Abstractions and Algorithms for Specializing Dynamic Program Analysis and Random
Fuzz Testing

by

Rohan Raju Padhye

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Koushik Sen, Chair
Professor Coye Cheshire
Assistant Professor Alvin Cheung
Assistant Professor Jonathan Ragan-Kelley

Summer 2020

Abstractions and Algorithms for Specializing Dynamic Program Analysis and Random
Fuzz Testing

Abstract

Abstractions and Algorithms for Specializing Dynamic Program Analysis and Random Fuzz Testing

by

Rohan Raju Padhye

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Koushik Sen, Chair

Software bugs affect the security, performance, and reliability of critical systems that much of our society depends on. In practice, the predominant method of ensuring software quality is via extensive testing. Software developers have considerable domain expertise about their own software, and are adept at writing functional tests. However, handcrafted tests often fail to catch corner cases. Further, it is far less common to find software projects that ship with handwritten tests that target non-functional software issues such as performance, concurrency, security, and privacy.

Dynamic program analysis techniques can be used to find potential software bugs by observing program execution. Such techniques are limited by the availability of quality inputs with which to execute the program. For example, although profilers can be used to diagnose performance issues when good stress tests are available, they are not very useful when provided with only small functional test cases. Researchers have also developed various algorithms to automatically generate test inputs. Techniques such random fuzzing are a promising approach for discovering unexpected inputs in a scalable manner. Coverage-guided fuzzing (CGF) tools that evolve a corpus of test inputs via random mutations and guided by test-execution feedback have recently become popular due to their success in crashing programs that process binary data. However, by relying solely on hard-coded heuristics, their effectiveness as push-button tools is limited when the test program, the input format, or the testing objective becomes complex.

This dissertation presents new abstractions and algorithms that empower software developers to specialize automated testing tools using their domain expertise.

First, we present two techniques to find algorithmic performance issues, such accidentally sub-optimal worst-case complexity, using only developer-provided functional tests: (1) TRAVI-OLI performs dynamic analysis of unit test executions to precisely identify program functions

that perform redundant data-structure traversals; (2) PERFFUZZ employs a novel algorithm based on CGF to automatically generate inputs that exercise worst-case complexity. These techniques have helped discover previously unknown asymptotic performance bugs in real-world software including the D3 visualization toolkit, the ExpressJS web server, and the Google Closure Compiler.

Second, we present ZEST+JQF, a technique and framework respectively to find semantic bugs in programs that process complex structured inputs in a multi-stage pipeline, such as compilers. This approach leverages domain knowledge about a program under test by allowing users to provide: (1) simple generator functions that sample syntactically valid inputs, and (2) predicate functions that determine whether a sampled input is also semantically valid. ZEST automatically guides the user-provided generator functions towards producing inputs that are likely to be semantically valid and also increase code coverage in the program under test. JQF allows researchers to plug-in custom algorithms for guiding such generators. Together, ZEST+JQF have enabled the discovery of 42 previously unknown software bugs in widely used Java projects such as OpenJDK, Apache Commons, Maven, Ant, and the Google Closure Compiler. Many of these bugs are far beyond the reach of conventional CGF or generator-based testing tools.

Finally, we present FUZZFACTORY, a framework for rapidly prototyping and composing domain-specific fuzzing applications. With FUZZFACTORY, new fuzzing applications can be created by defining a strategy for selecting which mutated inputs should be saved as the basis for subsequent mutations; such inputs are called *waypoints*. FUZZFACTORY provides a lightweight API for instrumenting programs such that they provide custom feedback during test execution; this feedback is used to determine if the corresponding test input should be considered a waypoint. We describe six domain-specific fuzzing applications created with FUZZFACTORY. We also show how two of these applications can be composed together to create a fuzzer that performs better than the sum of its parts.

To my parents and to my wife, Radhika.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I am tremendously grateful to my advisor, Koushik Sen, for his steadfast support and mentorship. I appreciate being given the flexibility to choose my own research direction and work on projects that I was passionate about. This was especially challenging when things did not go according to plan, but Koushik always believed in me and taught me how to fail fast when necessary. I am thankful for his advice on research, technical writing, and connecting with members of the broader research community. I always felt I could talk to Koushik about anything, professional or personal, without judgment; such confidence played a crucial role in my ability to succeed in graduate school. I hope I will be able to follow in his footsteps as a professor.

I probably wouldn't have started my journey towards a Ph.D. if it hadn't been for mentors such as Uday Khedker and Amitabha Sanyal at IIT Bombay, as well as Vibha Sinha and Senthil Mani formerly at IBM Research India. They played a pivotal role in sparking my interest in programming languages, software engineering, and in pursuing a career in research.

I would like to thank a number of faculty members at Berkeley for their time, support, and invaluable advice: Jonathan Bachrach, Coye Cheshire, Alvin Cheung, Susan Graham, Paul Hilfinger, Jonathan Ragan-Kelly, Sanjit Seshia, and Dawn Song.

I am indebted to my colleague and closest collaborator, Caroline Lemieux, with whom I co-authored a number of papers. I would also like to thank a number of people who have influenced my research direction and vision in various ways including advice, collaboration, and mentorship: Sarah Chasins, Daniel Fremont, Mike Grace, Lee Harrison, Eddie Kim, Miryung Kim, Yves Le Traon, Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, Tobias Ospelt, Mike Papadakis, Hayavardh Vijayakumar, and Andreas Zeller.

I am grateful that I could always rely on the amazing staff in the EECS Department including Jean Nguyen, Shirley Salanio, Audrey Sillers, Ria Briggs, Tami Chouteau, Roxana Infante, and Kostadin Ilov.

I am fortunate to have had fantastic colleagues and friends who made me look forward to making the trek up to campus every day: Rohan Bavishi, Benjamin Brock, Michael Chang, Wontae Choi, Michael Dennis, Rafael Dutra, Liang Gong, Giulia Guidi, Sagar Karandikar, Kevin Laeufer, Caroline Lemieux, Azad Salam, Stephanie Wang, Tyler Westenbroek, and Ed Younis.

Last but not least, I am grateful to my family and friends for always being there through the ups and downs and encouraging me to keep going.

Chapter 1

Introduction

Today, the vast majority of software is written by humans. Since much of our society depends on software systems, the consequences of inadvertently introduced software bugs can be devastating. With new application domains emerging faster than the mechanisms to produce provably good software automatically, software bugs are here to stay for the foreseeable future. The predominant form of ensuring quality in practice is via software testing: a 50 billion USD market by some estimates [69], which is expected to keep growing.

Now, software developers have considerable domain expertise and are adept at writing functional test suites. However, handcrafted test cases often fail to catch corner-case bugs. Such bugs leak into production software, and their effects can be devastating. *Software Fail Watch* [175] estimated that software failures have accounted for at least *1.7 trillion USD* of asset losses and impacted about 3.7 billion people—half the world's population—as of 2019.

Fortunately, numerous *program analysis* techniques have been developed to automatically reason about the run-time behavior of programs, with the goal of preventing or identifying software bugs. This dissertation almost exclusively focuses on applications of two of these techniques: *dynamic program analysis* and *random fuzz testing*. Although these approaches are well known, their effectiveness when implemented as push-button tools is limited to the availability of good test inputs or to discovering a narrow class of software faults respectively. We target scenarios where the test program, the input format, and the testing objective becomes complex. In such settings, the traditional approach of analyzing or automatically testing a program in isolation either does not scale or produces inadequate results. A *key insight* of this work is that we can specialize automated testing tools and make them smarter by drawing upon artifacts incorporating the domain expertise of software developers; thus, challenging testing problems can be made tractable.

Broadly, this dissertation addresses three related problems. First, we focus on automatically identifying *algorithmic performance bugs*; that is, software implementation issues that can cause programs to consume inordinate amounts of physical computing resources when presented with worst-case inputs. Second, we tackle the challenging problem of automatically testing programs that parse and transform complex structured inputs in a multi-stage pipeline, such as compilers. Third, we investigate mechanisms for rapid prototyping of au-

tomatic test-input generation tools specialized for achieving domain-specific test objectives. The solutions presented in this dissertation utilize a variety of data sources from existing unit tests to explicitly provided specifications.

The rest of this chapter provides an introduction to each of these three topics and outlines the research questions which prompted their development. Chapter 2 provides background material about dynamic program analysis and random fuzz testing. Chapters 3–6 describe four main contributions of this dissertation in detail: TRAVIOLI [144] identifies algorithmic complexity bottlenecks by *analyzing only functional unit tests* (Chapter 3); PERFFUZZ [107] automatically synthesizes program inputs that exercise worst-case algorithmic complexity (Chapter 4); JQF+ZEST [140, 141, 142] find *semantic bugs* deep within software that processes complex inputs with the help of QuickCheck-like *generator functions* and user-provided *validity predicates* (Chapter 5); FUZZFACTORY [143] enables rapid customization of fuzzing tools for *domain-specific testing objectives* (e.g. regression testing and finding memory consumption bugs) (Chapter 6). Chapter 7 discusses related work. Finally, Chapter 8 concludes with a summary of key takeaways and opportunities for future work.

1.1 Algorithmic Performance Bugs

Performance problems in software are notoriously difficult to detect and fix [93]. Unexpected performance issues can lead to serious project failures and create troublesome security issues. For example, a well-known class of Denial-of-Service (DoS) attacks target algorithmic complexity vulnerabilities [45] which cause a running program to exhaust computational resources when presented with worst-case inputs.

A large body of research has focused on diagnosing performance problems by observing or statistically analyzing dynamically collected performance profiles [79, 129, 15, 131, 176]. Almost all of these techniques assume the availability of test inputs with which to execute the candidate program for performance profiling. But where do these inputs come from? The most commonly chosen sources include (1) specially hand-crafted performance tests [131, 126], (2) standardized benchmark suites [15, 16, 41], (3) inputs that are commonly encountered in normal program usage (sometimes called *representative workloads*) [73, 200], or (4) inputs sent by users experiencing performance problems [176]. These sources of inputs either stress only average-case behavior, are subject to human bias and error, or can only be obtained when the damage is already done.

We want to be able to reason about worst-case program behavior, and identify any performance bottlenecks, without relying on the availability of worst-case inputs themselves. Test inputs that are designed specifically for analyzing a program's performance, such as the ones described above, are not readily available for the vast majority of software projects. However, most software projects ship with functional test cases that demonstrate various features and exercise a variety of code paths. Our first research question is thus:

| Can we identify algorithmic performance bottlenecks using only functional test cases? |

Chapter 3 presents TRAVIOLI, which is a first attempt at tackling this challenge. TRAVI-OLI performs dynamic analysis of a program's execution of functional unit tests. TRAVIOLI identifies program functions which traverse data structures, without relying on any knowledge of data-structure libraries or type definitions. The goal of TRAVIOLI is to identify a special class of algorithmic performance bottlenecks called *redundant data-structure traversals*. We have implemented TRAVIOLI for JavaScript and evaluated its analysis on five projects that are popular on GitHub and that make heavy use of data structures. In two of these projects, D3 and ExpressJS, TRAVIOLI was able to identify redundant traversals which correspond to algorithmic performance bottlenecks. Both of these issues were identified by the respective project developers as performance bugs, and one of them has since been patched with an optimization that provides asymptotic speedup.

Although these results are encouraging, TRAVIOLI's main limitation is that it can only identify program functions that are potential bottlenecks. A manual effort is still required to craft the worst-case input for such program functions before it can be confirmed as a performance bug or dismissed as a false positive.

We next investigated whether we could automatically generate program inputs that lead to pathological algorithmic performance. We decided to make use of coverage-guided fuzzing (CGF), which is a feedback-directed random test-input generation technique. Popular CGF tools such as AFL [196] perform an evolutionary search: test inputs are generated by performing random mutations on the binary representation of previously saved inputs (e.g. bit flips and splicing of multi-byte chunks). These tools are coverage-guided: new inputs are saved if their execution on the test program leads to a new program location being visited. Such code coverage is collected using lightweight program instrumentation.

Chapter 4 describes PERFFUZZ, a method to *automatically* synthesize pathological inputs using only functional test cases as a starting point. PERFFUZZ generates inputs by adapting the CGF algorithm, using developer-provided inputs that demonstrate the program's functionality as a set of *seeds* for mutation. A prior approach, SlowFuzz [149], used random fuzzing to find inputs that cause program execution to take really long execution paths. In contrast, PERFFUZZ uses multi-dimensional feedback and independently maximizes execution counts for all program locations. This enables PERFFUZZ to (1) find a variety of inputs that exercise distinct hot spots in a program and (2) generate inputs with higher total execution path length than previous approaches by escaping local maxima. PERFFUZZ is also effective at generating inputs that demonstrate algorithmic complexity vulnerabilities. We have implemented PERFFUZZ on top of AFL, a popular coverage-guided fuzzing tool. Chapter 4 also presents an experimental evaluation of PERFFUZZ on four widely used C libraries. We find that PERFFUZZ outperforms SlowFuzz by generating inputs that exercise the most-hit program branch $5\times$ to $69\times$ times more, and result in $1.9\times$ to $24.7\times$ longer total execution paths.
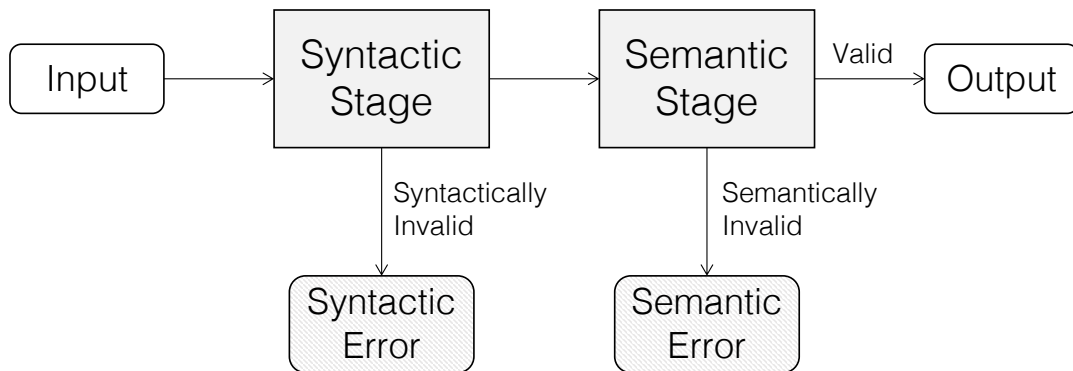
Figure 1.1: Architecture of a program having a structured-input processing pipeline.

1.2 Semantic Bugs in Input-Processing Pipelines

CGF tools such as PERFFUZZ rely on random mutations of inputs represented as sequences of bytes. These techniques are therefore effective at automatically generating inputs for programs that process binary data or parsers of simple text formats. However, the applicability of these methods is limited when testing programs that expect complex structured inputs. For example, a build system such as Apache Maven first parses its input as an XML document and checks its conformance to a schema before invoking the actual build functionality. Document processors, Web browsers, compilers and various other programs follow this same check-then-run pattern. In general, such programs have an input processing pipeline consisting of two stages: a syntax parser and a semantic analyzer. We illustrate this pipeline in Figure 1.1. The syntax parsing stage translates the raw input into an internal data structure that can be easily processed (e.g. an abstract syntax tree) by the rest of the program. The semantic analysis stage checks if an input satisfies certain semantic constraints (e.g. if an XML input fits a specific schema), and executes the core logic of the program. Inputs may be rejected by either stage if they are *syntactically* or *semantically invalid*.

Automatically testing such programs is challenging. The difficulty lies in synthesizing inputs that (1) satisfy complex constraints on their structure and (2) exercise a variety of code paths in the semantic analysis stages and beyond.

Conventional CGF tools that generate new test inputs via byte-level mutations easily destroy the syntax and semantics of previously valid inputs. Unsurprisingly, most of the bugs discovered by CGF when testing such software lie in their syntax parsing stages only.

To address this problem, we first observe that software developers are familiar with the input format of the programs they are testing. This is evidenced by the fact that software developers frequently write extensive unit tests, with hard-coded test inputs containing the expected structure and semantics. With this insight, we ask the following research question:

> How can we enable CGF tools to leverage the domain expertise of software developers, who are familiar with the structure and semantics of their programs' inputs? What abstractions can we provide to enable developers to encode their domain knowledge?

We propose to use an abstraction popularized by property testing tools like QuickCheck [37]. This abstraction consists of (1) a *generator function*, whose job is to randomly sample a syntactically valid input, and (2) a *validity predicate*, whose job is to check whether a syntactically valid input also satisfies certain semantic invariants. Due to the popularity of QuickCheck, which was originally developed for testing Haskell programs, as well as the proliferation of similar *property testing* tools in many other programming languages, this is a well-known abstraction. However, property testing tools themselves are not sufficient at testing complex programs such as compilers due to the very large space of inputs to sample from and a very large set of distinct program behaviors to exercise.

Chapter 5 describes ZEST, a technique which automatically guides QuickCheck-like random input generators to better explore the semantic analysis stage of test programs. ZEST treats random-input generators as deterministic functions of infinite bit sequences called *parameters*. We present the key insight that mutations in the untyped parameter domain map to structural mutations in the input domain. ZEST leverages program feedback in the form of code coverage and input validity using an algorithm we call *semantic fuzzing*. Chapter 5 also presents an evaluation of ZEST against AFL and QuickCheck on five Java programs: Maven, Ant, BCEL, Closure, and Rhino. ZEST covers $1.03\times$–$2.81\times$ as many branches within the benchmarks' semantic analysis stages as baseline techniques. Further, ZEST found 10 new previously unknown bugs in the semantic analysis stages of these widely used software projects. ZEST was the most effective technique in finding these bugs reliably and quickly, requiring at most 10 minutes on average to find each bug.

1.3 Domain-Specific Testing Objectives

Like PERFFUZZ and ZEST, many other researchers have adopted coverage-guided fuzzing technology to meet domain-specific testing objectives. Examples of such objectives include directed testing [23], differential testing [148], side-channel analysis [130], and discovering algorithmic complexity vulnerabilities [149].

Currently, the practice of developing domain-specific fuzzing applications is quite ad-hoc. For every new domain, researchers must find a way to tweak the fuzzing algorithm and produce a new variant of AFL or some other fuzzing tool. Each such solution can require non-trivial implementation effort. Further, these variants are independent and cannot be easily composed. We thus ask the following research question:

> How can we enable researchers to *create* and *compose* domain-specific fuzzing applications? What abstractions can we provide to support such specifications?

Chapter 6 describes FUZZFACTORY, a framework that enables researchers to rapidly pro-

totype new domain-specific fuzzing applications and compose them with each other. FUZ-ZFACTORY introduces the abstraction of *waypoints*: the intermediate inputs that are saved in a CGF-like fuzzing loop. In traditional CGF, the waypoints are any inputs that increase code coverage. FUZZFACTORY allows researchers to specify domain-specific criteria under which inputs should be considered waypoints. In particular, the FUZZFACTORY API allows researchers to (1) perform domain-specific instrumentation of test programs to collect custom feedback during test execution in the form of a key-value map, and (2) specify a function to aggregate this feedback across all inputs generated during a fuzzing loop. The result of this aggregation determines if a newly generated input in a fuzzing loop should be saved as a waypoint. We have identified some key properties of this aggregation function that can help provide formal guarantees that all waypoints make progress towards some domain-specific testing objective. Chapter 6 also describes six instantiations of domain-specific fuzzing applications that we developed using FUZZFACTORY, along with an experimental evaluation for each of them.

Chapter 2

Background

This chapter introduces some key concepts that are necessary for appreciating the novel contributions in this dissertation detailed in the subsequent chapters.

2.1 Software Bugs

The first recorded reference to the modern concept of a software bug appears in Ada Lovelace's notes [114] on the sketch of Charles Babbage's Analytical Engine, which would have been programmable using punch cards (emphasis original):

> "an analysing process must equally have been performed in order to furnish the Analytical Engine with the necessary *operative* data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the *cards* may give it wrong orders."

In this dissertation, we use the following broad definition of a software bug:

Definition 1 (Software Bug)**.** A *software bug* is any error or flaw in the implementation of a software system that causes it to produce incorrect results, exhibit undesirable behavior, or cause unintended consequences.

For example, consider an online shopping website that is designed to allow users to purchase items through a web browser. A software bug in the implementation of such a web-based application might: (a) lead to incorrect listings of item prices—a functional correctness issue, (b) take too long to return search results—a performance issue, (c) crash and exit when a button is clicked—a reliability issue, (d) let attackers steal a customer's credit card information—a security issue, (e) leak a customer's shopping activity—a privacy issue, etc.

Ideally, we want software to be free of any bugs. There are two overarching challenges to achieving this goal.

The first challenge stems from the definition of a software bug: how do we know what results are correct or what behavior is acceptable? This is the *specification problem*. Specifying desirable properties of software systems is challenging due to a number of reasons ranging from capturing the right requirements from stakeholders [133] to formally specifying them in an appropriate representation [104]. A related question that can be asked is: given a program's execution on some specific inputs, did it exhibit undesirable behavior? This is known as the *test oracle problem* [18]. Without good specifications or test oracles, it is impossible to identify software bugs even when they manifest at run-time.

The second challenge is that of automatically reasoning about software behavior. Assuming that we can specify some program property or behavior that is undesirable, can we determine if a given program is buggy?

This dissertation does not delve into the details of the specification or oracle problem. In some applications, such as when looking for semantic bugs, we assume that test oracles are available in the form of explicit `assert` statements inserted by developers, automatic checks inserted by sanitization tools [166, 167], or implied via the abnormal termination of a program by the operating system (e.g. aborts, segfaults, memory exhaustion, and other crashes). In other applications, such as when looking for performance issues, we assume that our goal is to generate a test inputs that exhibit pathological behavior (e.g. worst-case performance). Whether a program's behavior on such a pathological input is expected or unexpected is up to a human user to decide.

We next provide a background on research that addresses the second challenge, that of automatically reasoning about program behavior and/or identifying software bugs.

2.2 Program Analysis

Rice's theorem [159] states for any non-trivial property of partial functions—that is, properties that do not hold universally true or false for all partial computable functions—the question of whether a given algorithm computes a partial function with this property is undecidable. This theorem implies that all interesting questions about the behavior of programs are undecidable. With respect to our goal of determining if programs are buggy (for some non-trivial class of software bugs), it means that it is impossible to develop an algorithm that will precisely answer "yes" or "no" for every program. In practice, however, we are often satisfied with algorithms that can answer "yes", "no", or "maybe". The general field of computer science that deals with automated reasoning about software properties is called *program analysis*.

Program analyses can be broadly classified as *static* or *dynamic*. A *static analysis* examines a program's source code or other representation to simultaneously reason about all possible run-time behaviors under all possible program inputs. In contrast, a *dynamic analysis* observes one or more program executions on specific program inputs.

Since static analyses reason about all possible behaviors, they are often conservative; that is, they can prove the absence of a certain class of bugs in some cases and report that a

program *may* contain bugs in other cases. For example, a type checker [152] performs a static analysis to prevent programs written in a particular language execute operations that are not valid in the semantics of that language. A *sound* type checker can guarantee that well-typed programs never execute such invalid operations. Due to Rice's theorem, a sound type checker cannot also be *complete*; that is, it must necessarily reject some programs as ill-typed even though they may never execute invalid operations at run-time. Static analyses have been developed to find many classes of bugs, from null-pointer exceptions [52, 128, 117] and buffer overflow vulnerabilities [183, 105, 190] to data races [56, 97] and algorithmic performance issues [135]. Contemporary static analysis tools such as FindBugs [40], Error-Prone [76], and Infer [60], are actually extensive frameworks that support the addition of new analyses based on syntatic code smells, type analysis, and abstract interpretation respectively. Static bug finding tools attempt to minimize, but cannot escape from, *false positives*; that is, warnings of potential bugs or security vulnerabilities that would never manifest at run-time. Too many false warnings can pose a barrier to the adoption of static analysis tools [95]. Minimizing false warnings requires sophisticated algorithms which are ever more precise; unfortunately, an increase in precision almost always comes at the expense of scalability. Developing static analyses that are simultaneously sound, scalable, and highly precise is an active area of research. Closely related to static analysis is the use of formal methods to perform *software model checking* [92, 57]; that is, automatic verification of formally specified software properties.

Dynamic analyses attempt to uncover software faults by demonstrating the violation of some property on specific program inputs, but cannot prove the absence of bugs in general. As such, dynamic analyses suffer from *false negatives*. Dynamic analyses have been developed to target similar classes of bugs, including data races [134, 167], buffer overflows [160, 166], and algorithmic performance issues [131]. Dynamic analysis is often implemented by performing program instrumentation; that is, inserting code at various locations in the program being analyzed. The instrumentation code monitors program execution by collecting data about program values during execution and performing run-time checks. Although program instrumentation introduces a run-time overhead, dynamic analyses tools can be a more scalable solution than static analysis for large programs; for example, contrast a $10\times$ slowdown during program execution for dynamic analysis with an inter-procedural static analysis whose run-time complexity is a cubic function of program size.

This dissertation focuses exclusively on dynamic program analysis techniques. The effectiveness of dynamic analysis techniques is of course limited by the quality test inputs that a program is executed with. Much of this dissertation concerns a problem that is closely related to the field of program analysis: *test-input generation*.

2.3 Automatic Test-Input Generation

Test-input generation is the problem of synthesizing an input i for program p, such that the execution of p with i leads to some run-time property ϕ being satisfied. A run-time property

is usually, though not always, a predicate about program values at a certain program point; for example, *"is the value of variable `x` at line 6 greater than 0?"*. Such problems can be represented in a number of equivalent formulations, such as discovering inputs that cause a program to violate an `assert` statement, that cause a test suite to fail, that cause the program to crash or throw an exception, or simply that lead to a particular program point being visited. Some test-input generation problems consider run-time properties about program execution: for example, the number of times a program function is invoked, the amount of memory consumed by a program, or whether sensitive information (e.g. a password) is leaked (e.g. to a file).

The naïve approach of enumerating all program inputs and checking the run-time property dynamically quickly becomes impractical as the space of program inputs becomes very large or unbounded.

The field of *search-based software engineering* (SBST) [122, 83, 82, 194] recognizes that for certain classes of programs and properties, test inputs with desirable properties can be discovered via mathematical optimization or metaheuristic techniques such as hill climbing, simulated annealing, and genetic algorithms.

Symbolic execution techniques recognize that distinct program inputs which lead to the execution of the same program path—that is, the same control-flow sequence—belong to the same equivalence class for the purpose of exploring program behaviors. Instead of enumerating *inputs*, they enumerate *program paths*. A program path p is *feasible* if there is at least one program input i such that the program's execution on i takes path p. The goal now is to find a feasible program path that visits a program point of interest. The basic idea of symbolic execution dates back to 1976 [38, 99]: (1) programs can be executed by treating input variables as *symbols*, (2) expressions can be evaluated to formulas possibly involving symbolic input variables, (3) at every point during symbolic execution, a logical *path condition p* is maintained, which is initialized to `true` at the start of the program, (4) when a conditional branch involving a symbolic condition q is encountered, the path condition for the true branch is $p \wedge q$ and for the false branch is $p \wedge \neg q$, (5) a test-input for any program path with path condition p can be constructed by finding a solution to p (if it exists). Research on symbolic execution-based test-input generation exploded in the 2000s as the technology behind constraint solvers (such as SMT [19]) improved. A number of symbolic execution tools such as EXE [30], DART [72], CUTE [164], JPF-SE [5], KLEE [29], Pex [181], SAGE [25], and S2E [36] have been developed for various platforms and employing various performance optimizations. A key challenge for symbolic execution tools is the *path explosion problem* [31]: the number of program paths to explore grows exponentially with conditional branches, and can even be unbounded in the presence of symbolic loops. Later work on symbolic execution has explicitly targeted challenges with scalability [14, 165].

2.4 Random Fuzz Testing

Practioners have long known that simply generating test inputs at *random* is a scalable and surprisingly effective method for finding implementation faults in computer systems. Random test generation was first popularized for finding faults in hardware in the 1970s and 80s: random test-input generators were developed for sequential circuits [26], memories [64], ICs [50], floating point units [120], cache controllers [189], etc.

Random test-case generation as a methodology for finding software bugs was initially dismissed: Myers' 1979 book *The Art of Software Testing* [126] states "the least effective methodology of all is random-input testing". However, by the 1980s random testing was found to be "more cost effective" than systematic techniques and "a useful validation tool" that achieves "a very high degree of coverage" [53, 91]. Many of these results reflect experiences in testing software that operated on a fixed set of numeric inputs, such as computer simulations.

In 1990, Miller et al. [123] developed `fuzz`, a tool for testing the reliability of Unix utilities by generating random sequences of characters as input[1]. They were able to crash dozens of standard widely used Unix utilities including `vi`, `emacs`, `as`, `ftp`, `spell`, and `uniq` by simply feeding random input data generated by `fuzz`. A common cause of these crashes was segfaults; many of the tested programs had input-validation bugs such as missing size checks or improper format strings that could cause the programs to read/write memory out of bounds when presented with unexpected inputs. Such *buffer overflow* bugs were and remain serious security vulnerabilities[2].

Today, *fuzz testing*, or simply *fuzzing*, refers to any test-input generation technique that produces inputs using some randomized algorithm. The input generator is itself sometimes referred to as a *fuzzer*. In the three decades following Miller et al.'s work, fuzz testing has become a rich field of research for finding security vulnerabilities [118, 70].

The key advantage of fuzz testing over systematic techniques such as symbolic execution is scalability: a randomized search can explore many program behaviors quickly and can be easily parallelized. Possibly fueled by the increasing availability of cheap computing resources, fuzz testing has become one of the predominant automated testing methods used in practice. For example, Google's ClusterFuzz system has found more than 16,000 bugs in the Chrome web browser and over 11,000 bugs across 160+ open-source projects by January 2019 [74].

Modern fuzzers rarely generate inputs randomly from scratch: it is very unlikely that inputs constructed as purely random sequences of bytes will exercise a non-trivial fraction of a complex software system. The two broad approaches to smarter input generation include *model-based fuzzing* and *mutation-based fuzzing*.

---

[1]Apparently, one of the authors accidentally discovered fuzz testing when working from home one "dark and stormy night"; the rain introduced noise in the phone lines which were transmitting his commands to a remote Unix system and caused programs at the other end to crash [123].

[2]MITRE Corporation's Common Weakness Enumeration (CWE) list ranks buffer overflows as number 1 in the top 25 most dangerous software errors in 2019 [124].

Model-based fuzzers generate inputs based on some understanding of what kind of inputs a program expects. Although this might seem unintuitive—the goal of fuzzing is to generate *unexpected* inputs that reveal software bugs—the idea is that generating inputs having some basic structure or syntax will guarantee that certain parts of a test program's code logic are exercised. For example, grammar-based fuzzing [121, 43, 173, 71, 22] techniques use context-free grammar specifications to generate strings belong to a particular language. CSmith [192] generates C programs to test C compilers, using a combination of the knowledge of C's syntax and semantics. Several network-protocol fuzzers [17, 94, 146] generate input messages that belong to some specified format. Model-based fuzzers are popular approaches for testing software with graphical user interfaces [63, 193].

Mutation-based fuzzers generate inputs by performing random changes on valid *seed* inputs. The idea is that making small random changes to a valid input, such as flipping some bits in an input to a program that processes binary data (e.g. a media player), or inserting random keywords in a text input to a parser (e.g. in a database query processor) will correspond to subtle changes in the execution path of the test program through its control-flow graph. Random mutations will create new, previously unseen and possibly unexpected inputs, while retaining much of the syntax, structure, and other features of the valid seed input. This idea has been used extensively by security-oriented fuzzers such as honggfuzz [179], zzuf [85], and radamsa [137]. These tools have been used to find hundreds of security vulnerabilities in commonly used software such as Unix utilities, network protocol implementations, and C libraries.

Both model-based fuzzers and mutation-based fuzzers make use of some knowledge about what kind of inputs a program expects. Both techniques as described above are *black box*; that is, they do not analyze the test program's source code or collect any additional information during program execution. Such black box testing is incredibly efficient, especially when compared to *white box* techniques such as symbolic execution which need to collect path constraints for every execution. Black-box fuzzers are also embarassingly parallelizable since the generation of every input is independent from every other input. However, a direct consequence of this fact is that the probability of generating an input that reveals a bug is the exact same when generating the very first input as it is when generating say the hundred millionth input. Researchers have long known that some of the most important questions about the random testing strategy include *"'How long should it run?"* and *"Has it covered all the important cases?"* [189].

One way to measure the quality of a set of test inputs generated by a fuzzer is to use proxy metrics such as *code coverage*, which correspond to the amount or fraction of program code that is exercised by test inputs. Common granularities of code coverage include *line coverage*, *statement coverage*, *branch coverage*, *basic-block coverage*, and *edge coverage* (the latter two refer to nodes and edges in a program's control-flow graph respectively). A straightforward strategy for tracking the progress of a fuzzing session is to measure the code coverage achieved by all the inputs generated so far; fuzzing is no longer viable when the rate of increasing code coverage falls below a certain threshold. However, measuring code coverage requires test programs to be *instrumented* with code that tracks which parts of the program are being

---

Algorithm 1 The coverage-guided fuzzing algorithm

Input: an instrumented test program p, a set of initial seed inputs $\mathcal{I}$
Output: a corpus of automatically generated inputs $\mathcal{S}$, a set of failing test inputs $\mathcal{F}$
 1: $\mathcal{S} \leftarrow \mathcal{I}$
 2: $\mathcal{F} \leftarrow \emptyset$
 3: $totalCoverage \leftarrow \emptyset$
 4: **repeat**   ▷ Main fuzzing loop
 5:      **for** i in $\mathcal{S}$ **do**
 6:          **if** sample FUZZPROB(i) **then**
 7:              $i' \leftarrow$ MUTATE(i)   ▷ Generate new test input i'
 8:              $coverage, result \leftarrow$ EXECUTE(p, i')   ▷ Run test with new input i'
 9:              **if** $result =$ FAILURE **then**
10:                  $\mathcal{F} \leftarrow \mathcal{F} \cup \{i'\}$
11:              **else if** $coverage \cap totalCoverage \neq \emptyset$ **then**
12:                  $\mathcal{S} \leftarrow \mathcal{S} \cup \{i'\}$ ▷ Save i' if new code coverage achieved
13:                  $totalCoverage \leftarrow totalCoverage \cup coverage$
14:              **end if**
15:          **end if**
16:      **end for**
17: **until** given time budget expires
18: **return** $\mathcal{S}, \mathcal{F}$

---

exercised when executing test inputs. This instrumentation adds performance overhead, and can reduce the overall fuzzing efficiency.

Collecting code coverage during a fuzzing session does brings one very important advantage, at least to mutation-based fuzzers. The coverage information can be used to augment the set of *seed* inputs: automatically generated (i.e., fuzzed) inputs that exercise previously uncovered code can be used as the basis for subsequent mutation. In this way, fuzzing can become *feedback-directed* and test inputs can *evolve* over time. The vast majority of recent progress in fuzz testing [118], both in terms of new research and new discoveries of serious software bugs, has stemmed from the field of *coverage-guided fuzzing* (CGF). Chapters 4–6 describe new algorithms that improve upon or retarget the core ideas developed in CGF. We next describe how this technique works in detail.

2.5   Coverage-Guided Fuzzing (CGF)

Algorithm 1 describes how CGF works at a high level. The CGF algorithm takes as input an instrumented test program p and a set of user-provided *seed inputs* $\mathcal{I}$. CGF maintains three global states: (1) $\mathcal{S}$ is a set of saved inputs to be mutated by the algorithm, (2) $\mathcal{F}$ is a set of bug-revealing inputs corresponding to test failures, and (2) *totalCoverage* tracks the

cumulative coverage of the program on the inputs in $\mathcal{S}$. CGF can track any kind of coverage; in practice, branch coverage or edge coverage is commonly used. $\mathcal{S}$ is initialized to the set of user-provided seed inputs (Line 1) and *totalCoverage* is initialized to the empty set (Line 3). The main fuzzing loop of CGF (Line 4) keeps making passes over the set of inputs (Line 5), selecting an input i from the set $\mathcal{S}$. With some probability (Line 6) determined by an implementation-specific heuristic function FUZZPROB(i), CGF decides whether to mutate the input i or not. If i is selected for mutation, CGF randomly mutates i to generate i' (Line 7). The random mutation can be selected from a set of predefined mutations such as bit flipping, byte flipping, arithmetic increment and decrement of integer values, replacing of bytes with handpicked interesting values, etc. CGF then executes the program p with the newly generated input i' (Line 8). The coverage corresponding to this execution is collected into the variable *coverage*. The variable *result* whether the execution terminated normally or abnormally (e.g. with a crash or timeout). Inputs corresponding to test failures are added to a set $\mathcal{F}$ (Line 10). If the observed coverage *coverage* when executing a non-failing input contains some new coverage point that is not present in the global cumulative coverage *totalCoverage* (Line 11), then the new input i' is added to the set of saved inputs $\mathcal{S}$ (Line 12) and *totalCoverage* is updated to include the new coverage (Line 13). The input i' will then get mutated during a future iteration of the fuzzing loop. The fuzzing loop continues until a time budget has expired (Line 17). Finally, the generated test corpus $\mathcal{S}$ and the set of failing inputs $\mathcal{F}$ are returned as the result of fuzzing (Line 18).

2.5.1 Contemporary CGF Tools: AFL and libFuzzer

CGF was popularized by AFL [196]—which stands for *American Fuzzy Lop*—an open-source fuzzing tool developed by Michał Zalewski at Google. We next describe some implementation heuristics of AFL in detail, since many of these heuristics are inherited or borrowed by PERFFUZZ (Chapter 4), ZEST (Chapter 5), and FUZZFACTORY (Chapter 6).

AFL is designed to test standalone programs that process inputs either as a single file or via the standard-input stream. AFL therefore generates inputs as fixed size binary files. AFL uses inter-process communication (IPC) to transfer inputs to and receive code coverage feedback from an instrumented program under test. The IPC is done via fixed-size shared memory. Instead of restarting the test program for every test execution, which would be very slow, AFL uses a *fork server* [199] mechanism that uses the Unix `fork` system call to create copies of a partially initialized process.

AFL starts fuzzing using a user-provided set of seed input files, corresponding to set $\mathcal{I}$ in Algorithm 1. The mutations applied by AFL to generate new inputs include:

- Bitflips/byteflips at random locations.

- Setting bytes to random or interesting (0, `MAX_INT`) values at random locations.

- Deleting/cloning random blocks of bytes.

AFL also occasionally performs *splicing* mutations, more commonly called a *crossover* muta-tion. For a candidate input i, a splicing mutation chooses a random input i' in $\mathcal{S}$ and pastes a random sub-sequence from i' at a random offset in i. This stage runs only when AFL has not discovered new coverage in several cycles of the main fuzzing loop. AFL also allows users to specify a *dictionary* of keywords or *magic* byte sequenees that are then randomly inserted into mutated inputs.

AFL ships with wrappers for GCC and Clang that can instrument C/C++ programs at compile time. The instrumentation logic collects edge coverage in a 64KB data structure called the *bitmap* [197]. The basic strategy is as follows: (a) every basic block is assigned a pseudo-unique 16-bit random identifier at compile time, (b) during program execution, at the entry of each basic block, a 16-bit *edge identifier* is computed as a hash function of the identifiers of the current and previously visited basic block, (c) a one-byte entry in the bitmap at the offset corresponding to the edge identifier is incremented. Thus, the bitmap is essentially a hashmap of 2^{16} entries, which tracks the number of times (between 0–255) an edge is executed[3]. At the end of test program execution, for a single test input, AFL *buckets* the counter for every entry in the bitmap. Roughly, the highest order bit of the counter value is retained wheras the lower bits are set to zero. So, a count of 1000 would be bucketed to 512, a count of 35 to 32, etc. Finally, the set bits in the post-processed bitmap are returned as the set *coverage* in Algorithm 1. This allows AFL to save inputs that not only increase edge coverage, but also inputs whose edge execution count differs from the execution counts for the same edge by any previously saved inputs by an order of magnitude.

LibFuzzer is another widely used CGF tool that targets the LLVM platform. Since around 2016, libFuzzer [111] has been included as part of the LLVM project. libFuzzer borrows many ideas from AFL, but differs in two important ways. First, libFuzzer is designed to fuzz library functions instead of command-line programs. LibFuzzer repeatedly invokes a user-provided program function that accepts a variable-sized byte array with fuzzed input values. LibFuzzer runs the entire fuzzing session in one process; therefore, the throughput of libFuzzer—in terms of number of test executions per unit time—is much higher than that of AFL. Second, libFuzzer also provides some additional features such as user-defined mutators and other instrumentation hooks. LibFuzzer has therefore become a popular reusable component for use in fuzzing programs written in other applications, such as Rust [32].

Together, AFL and libFuzzer have been used to discover thousands of security vulner-abilities, mostly in C/C++ programs such as Google Chrome, OpenSSL, Mozilla Firefox, Adobe Flash, VLC Media Player, and others.

---

[3]Hash collisions and integer overflows can introduce inaccuracies in this measurement.

Chapter 3

TRAVIOLI: Dynamic Analysis of Data-Structure Traversals

We start by addressing the problem of identifying algorithmic performance bottlenecks in programs using only developer-provided functional test cases, as motivated in Section 1.1.

This chapter presents TRAVIOLI [144][1], a dynamic analysis for identifying a special class of algorithmic performance bottlenecks: *redundant data-structure traversals*. A redundant traversal occurs when a program traverses the same data structure (of size say n) multiple times (say m times) even though the data structure is not updated between each traversal, and where m and n are non-constant values derived from program input. A redundant traversal has worst-case algorithmic complexity of at least $\mathcal{O}(mn)$. Redundant traversals often indicate a "performance bug"; that is, a sub-optimal implementation choice of data structures or algorithms. Section 3.1 motivates the problem with an example and discusses prior work that identified this class of performance issues.

TRAVIOLI aims to identify redundant traversal bugs using dynamic program analysis; that is, by running a program with some inputs and analyzing its execution. In particular, TRAVIOLI is designed to analyze the execution of programs on readily available functional test cases. Functional tests, such as unit tests, rarely stress the run-time performance of a program, use very small input sizes, and almost never exercise worst-case behavior. TRAVIOLI therefore addresses two important challenges: (1) identifying where data structures are traversed, and (2) identifying which data-structure traversals are potentially *redundant*. Sections 3.2 and 3.3 describe TRAVIOLI's dynamic analysis.

In order to keep the technique as general as possible, we implement TRAVIOLI as an analysis of JavaScript programs. Unlike other popular languages such as Java, C++, and Python, there is no standard data-structure library in JavaScript. Moreover, JavaScript's exceedingly dynamic type system makes it very challenging to statically reason about which objects represent data structures or nodes of a data structure. Finally, the dynamic nature

---

[1]A portion of this chapter was previously published in a conference proceedings [144], which are under copyright © IEEE 2017. Reproduced with permission as per rights retained by the original author.

of JavaScript also makes it very hard to reason about a program's call graph, which further complicates analysis of recursive data-structure traversals. TRAVIOLI can (1) identify data-structures composed of a mix of arrays and objects connected by references, (2) identify traversals that involve a mix of loops and recursive function calls (including complex mutual recursion), (3) does not depend on any type information, and (4) can analyze program executions on very small input sizes.

We implemented TRAVIOLI and used it to analyze 5 popular JavaScript projects— `express`, `d3-collection`, `d3-hierarchy`, `immutable-js`, `mathjs`—by running off-the-shelf unit tests provided by the developers. Section 3.4 describes the results of our empirical evaluation. TRAVIOLI was able to identify two redundant traversal bugs, one each in `d3-hierarchy`—a redundant $\mathcal{O}(n^2)$ traversal that can be optimized to $\mathcal{O}(n \log n)$—and `express`—a redundant $\mathcal{O}(mn)$ traversal that can be optimized to $\mathcal{O}(n)$. Both bugs have been confirmed by their respective developers; D3 has also incorporated our proposed fix.

The implementation of TRAVIOLI and scripts to reproduce the experiments listed in this chapter are available at: `https://github.com/rohanpadhye/travioli`.

3.1 Motivation

Consider the JavaScript function `containsAll` shown in Figure 3.2. The function `containsAll` takes as input a linked list `list` and an array `arr` and returns `true` if an only if all items in the array are also present in the list, by repeatedly invoking the `contains` function defined in Figure 3.1. The list is traversed multiple times without any change to its data—this is a case of redundant traversal. If the list contains n elements and the array is of length m, then the worst-case complexity of `containsAll` is $\mathcal{O}(mn)$. This is an example of a *redundant data-structure traversal*. Such instances are often indicative of performance bugs and can be fixed by using different data structures (such as hashed sets) or caching.

This class of performance bugs was identified in prior work; we are aware of at least two program analysis techniques that have been proposed to identify such issues. First, Clarity [135] uses static analysis to detect program functions which perform $\mathcal{O}(n)$ data-structure traversals $\mathcal{O}(m)$ times without any changes to the data structure between the traversals. Functions containing such *redundant traversals* can often be modified to use different data structures that improve the function's performance. However, Clarity, being a static analysis technique, makes conservative assumptions and cannot capture fine grained information about traversals along different conditional branches in a program (e.g. traversal of a binary-search tree). Second, Toddler [131] uses dynamic analysis to detect similar performance issues by discovering statistical similarities in memory access patterns at a program location. However, Toddler requires specially constructed performance tests and does not capture an abstract notion of data-structure traversal. Moreover, both techniques primarily analyze program loops, and do not capture recursive data-structure traversals such as `contains` defined in Figure 3.1.

```
1 /* Check if a linked list contains a value */
2 function contains(list, x) {
3   if (list === null) {
4     return false;
5   } else if (list.data === x) {
6     return true;
7   } else {
8     var tail = list.next;
9     return contains(tail, x);
10  }
11 }
```

Figure 3.1: A recursive function containing a traversal.

```
1 /* Does 'list' contain everything in 'arr'? */
2 function containsAll(list, arr) {
3   for (var i = 0; i < arr.length; i++) {
4     var item = arr[i];
5     if (contains(list, item) == false) {
6       return false;
7     }
8   }
9   return true;
10 }
```

Figure 3.2: A function that redundantly traverses a list.

A key advantage of TRAVIOLI is that it can detect a traversal even if a program is executed on a small unit test—the program does not need to execute a program location many times to detect a traversal. Another key advantage of TRAVIOLI is that it can detect a traversal even if the traversal involves recursive function calls and loop iterations.

3.2   Identifying Data-Structure Traversals

Before we can identify *redundant data-structure traversals*, we must first be able to identify where a program actually performs a *data-structure traversal*. But what exactly constitutes a data structure traversal? Surprisingly, even though traversal is a fundamental concept in computer science, it is not easy to find a precise definition that distinguishes a *traversal* from simply an access of some records of a data structure. For example, Skiena [174] describes a traversal as an algorithm that systematically visits some or all of the data items of a data structure.

```
1 /* Sum values in records in array */
2 function sum(arr) {
3   var result = 0, record, i;
4   for(i = 0; i < arr.length; i++){
5     record = arr[i];
6     result += record.val;
7   }
8   return result;
9 }
```

Figure 3.3: A function that traverses an array.

```
1 /* Compute the length of linked list  */
2 function len(list) {
3   var count = 0;
4   while (list != null) {
5     count++;
6     list = list.next;
7   }
8   return count;
9 }
```

Figure 3.4: A function that traverses a linked list.

We note that the running time of program functions that perform traversals usually increases with the increase in the size of the input data structures. In contrast, the running time of a program function that simply accesses a bounded number of records of a data structure is not considered a traversal. With this insight, we identify program functions that operate on data structures and whose running time could be arbitrarily increased by increasing the size of the input data structure. We call such functions *traversing functions.*

3.2.1   Traversing Functions

Consider the function `sum` in Figure 3.3. The function iterates over an input array of objects, `arr`, and computes the sum of the `val` field of the objects it contains. The function is an example of a simple data-structure traversing function. The running time of the function can be increased by increasing the size of the input array.

Although in this example we could easily identify the input to the function (i.e. the array `arr`), this may be non-trivial for complex functions where inputs could be passed via global or static variables. We define *read footprint* to precisely capture the set of inputs to a function.

```
1 /* Add values from a pair of array elements. */
2 function addPair(arr) {
3   var rx = arr[0];
4   var ry = arr[1];
5   return rx.val
6        + ry.val;
7 }
```

Figure 3.5: A non-traversing function.

Definition 2 (Memory Location). A *memory location* is the address of a piece of memory which stores a program value that can be read by a program. A memory location is often denoted in a program by a variable, an element of an array, or a field of an object.

Definition 3 (Read Footprint). The *read footprint* of a function execution consists of all memory locations that are read during the function execution without any prior write to them during the same execution. Such memory locations could be treated as the input to the function.

For example, the read footprint of the `sum` function consists of the array `arr`, all its elements (accessed via `arr[i]`), the `length` field of the array (accessed via `arr.length`), and the field `val` of the objects stored in the array (accessed via `record.val`). In contrast, the memory locations denoted by the variables `i`, `record` and `result` are not part of the read footprint, because, in any execution of `sum`, `sum` first writes them before reading them.

Definition 4 (Traversing Function). We say that a program function is a *traversing function* if the size of its read footprint is unbounded. We say that a function *contains a traversal* if and only if it is a traversing function.

The function `sum` in Figure 3.3 contains a traversal because the size of the read footprint increases if the size of the input array `arr` is increased.

The function `len` in Figure 3.4 is another example of a traversing function. The read footprint of the `len` function consists of the memory location denoted by `list` and the memory locations denoted by the `next` field of all objects reachable from `list` by following the `next` field zero or more times. The read footprint of this function can be increased by increasing the size of the list passed as an argument.

In contrast, the function `addPair` in Figure 3.5 is not a traversing function. The function `addPair` adds the values of the first two elements of the input array. While this function also reads multiple elements of `arr`, it is not a traversing function because the size of its read footprint is always bounded regardless of the size of the input array or the values it contains.

3.2.2 Detecting Traversing Functions

The problem of determining if a function contains a traversal is undecidable in general (see Theorem 1 in Section 3.3.4). However, in many cases, one can determine whether a function has a traversal either by analyzing the source code or by analyzing an execution of the function. We now describe a dynamic analysis technique, called TRAVIOLI, to determine if a function contains a traversal. TRAVIOLI works by checking a set of conditions on an execution of the function—if the conditions are satisfied then we say that the function contains a *possible* traversal. Our technique is approximate in the sense that it can give both false positives and negatives. However, we have identified a set of conditions which, if satisfied, often accurately indicate the presence of a traversal. A key feature of TRAVIOLI is that we do not need to invoke the function on an input having a large read footprint—TRAVIOLI can detect a traversal by analyzing the execution of the function on a small test input.

TRAVIOLI uses program instrumentation to generate a trace of *events* corresponding to reads and writes of memory locations. In the following discussion, whenever an execution of a function reads a memory location that the function execution has not written before, we call it an *input-read event.* An input-read event contains the address of the memory location being read, the value being read, and the program location where the read is performed by the function. TRAVIOLI determines the input-read events during each function execution and analyzes them to determine if the function has a traversal.

From executions of `sum` and `len` in Figures 3.3 and 3.4, respectively, one can observe that different memory locations are read at the same program location: `sum` reads the elements of the array `arr` at line 5 and `len` reads the `next` field of the `list` objects at line 6. This observation suggests that a traversal should satisfy the following two conditions:

C1. At least two input-read events at some program location ℓ access different memory locations, and

C2. the memory locations involved in the input-read events either belong to the same object, or belong to different objects connected by a series of pointers.

Note that `addPair` in Figure 3.5 does not satisfy the first condition because the two elements of the array are read at different program locations—lines 3 and 4, respectively. Further, the above two conditions result in a false positive for the function `third` in Figure 3.6. The function `third` calls `n` twice, and line 8 accesses `next` field of objects connected by a pointer. Thus both conditions are satisfied. However, `third` does not contain a traversal, since its read footprint is bounded to at most two linked-list nodes. The imprecision stems from the first condition, which requires two input-read events to occur at similar execution points, where two execution points are deemed similar if they have the same program locations. This notion of similarity of two execution points is too coarse-grained. We can alleviate this problem if we say two execution points are similar if they are executing the same program location and have identical call stacks. We capture such a state of execution in a concept called execution contexts.

```
1 /* Get the third element of a linked list */
2 function third(list) {
3   var node = n(list);
4   node =  n(node);
5   return node.data;
6 }
7 function n(node) {
8   return node.next;
9 }
```

Figure 3.6: Another example of a non-traversing function.

Definition 5 (Execution Context). The *execution context* of an event with respect to an execution of a function f is a sequence $(f_1 : \ell_1)(f_2 : \ell_2) \ldots (f_n : \ell_n)$, where

- f_1 is the function f,

- for each i such that $1 \leq i < n$, ℓ_i is the program location within function f_i where f_{i+1} is invoked in the current execution, and

- the function f_n is currently executing the program location ℓ_n to generate the input-read event.

For example, in an execution of the function `third` in Figure 3.6, the two input-read events at line 8 have the execution contexts `(third:3)(n:8)` and `(third:4)(n:8)` with respect to the execution of the function `third`. Unless otherwise specified, we always refer to execution contexts with respect to the execution of the function being analyzed for traversals. In order to remove the false positive for `third`, we refine the first condition for traversal as follows:

C1. At least two input-read events at some execution context access different memory locations.

The revised condition gives no false positive for any of the previous examples. Unfortunately, this revision, which uses a fine-grained notion of similarity of execution points, introduces false negatives—it fails to detect data-structure traversals via recursive functions, such as the function `contains` defined in Figure 3.1.

In the function `contains`, a recursive traversal occurs at line 8, but its execution does not meet condition C1 because the execution contexts of the input-read events at this program location are different. In particular, the execution context is `(contains:8)` for the first input-read event, `(contains:9)(contains:8)` for the second input-read event, `(contains:9)(contains:9)(contains:8)` for the third input-read event, and so on. Such execution contexts become more complicated for more complex functions involving mutual recursion, such as the function `alt` in Figure 3.7.

```
 1 /* Alternately add and subtract from items. */
 2 function alt(obj) {
 3   return p(obj.items, true, 0);
 4 }
 5 function p(node, flag, total) {
 6   if (node != null) {
 7     var value = node.data;
 8     return flag ? q(node, flag, total + value)
 9                 : q(node, flag, total - value);
10   } else {
11     return total;
12   }
13 }
14 function q(node, flag, total) {
15   var tail = n(node);
16   return p(tail, !flag, total);
17 }
18 function n(node) {
19   return node.next;
20 }
```

Figure 3.7: Mutually recursive functions containing a traversal.

The function `alt` traverses the linked list rooted at `obj.items` and alternately adds and subtracts values of its nodes to the total. The boolean `flag` passed to function p at line 8 decides which operation to perform, and this flag is toggled by the function q at line 16. Here, `p` and `q` are mutually recursive, and the traversal of the linked list occurs at line 19 after q calls n at line 15. The first time program control reaches line 19, the execution context is `(alt:3)(p:8)(q:15)(n:19)`; the second-time a different branch is taken in p, and thus the context is `(alt:3)(p:8)(q:16)(p:9)(q:15)(n:19)`, and so on.

In TRAVIOLI, a key observation we make is that, despite the differences in the execution contexts of the input-read events involved in a traversal, the contexts are equivalent *modulo recursion* (i.e. after removing any cycles). Such reduced execution contexts, which we define next, are called *acyclic execution contexts* (AEC) and they are constructed as follows. For an execution context $(f_1 : \ell_1)(f_2 : \ell_2) \ldots (f_n : \ell_n)$, we first construct an *execution-context graph* consisting of a node for each unique function f_i and a special node `end`. Moreover, let `start` denote the node corresponding to f_1. For every consecutive pair $(f_i : \ell_i)(f_{i+1} : \ell_{i+1})$ in the execution context, we add a directed edge from f_i to f_{i+1} with label ℓ_i and weight i. Additionally, we add an edge from f_n to `end` with label ℓ_n and weight n. For the example in Figure 3.7, the execution-context graph for the second input-read event at line 19 is shown in Figure 3.8, where the edges are labeled by the program locations ℓ and weights w.

$$\ell=9,w=4$$

$$\ell=3,w=1 \quad \overbrace{\ell=8,w=2} \quad \ell=15,w=5 \quad \ell=19,w=6$$

$$\text{alt} \xrightarrow{} p \underset{\ell=16,w=3}{\overset{}{\rightleftarrows}} q \xrightarrow{} n \xrightarrow{} \text{end}$$
$$(\texttt{start})$$
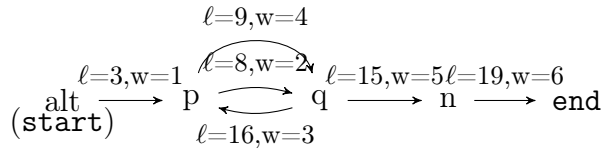
Figure 3.8: Execution-context graph for the execution context `(alt:3)(p:8)(q:16)(p:9)(q:15)(n:19)`.

Example	Execution contexts	AEC
Fig. 3.6,	(third:3)(n:8)	(third:3)(n:8)
Line 8	(third:4)(n:8)	(third:4)(n:8)
Fig. 3.4,	(len:6)	**(len:6)**
Line 6	(len:6)	**(len:6)**
Fig. 3.1,	(contains:8)	**(contains:8)**
Line 8	(contains:9) (contains:8)	**(contains:8)**
Fig. 3.7,	(alt:3)(p:8)(q:15)(n:19)	**(alt:3)(p:8)(q:15)(n:19)**
Line 19	(alt:3)(p:8)(q:16)(p:9)(q:15)(n:19)	**(alt:3)(p:8)(q:15)(n:19)**

Table 3.1: Execution contexts and AECs for first two read events.

Definition 6 (Acyclic Execution Context). The **acyclic execution context** (AEC) of an execution context is the sequence $(f_1 : \ell_1)(f_2 : \ell_2) \ldots (f_k : \ell_k)$ such that $f_1 \xrightarrow{\ell_1} f_2 \ldots f_k \xrightarrow{\ell_k} f_{k+1}$ is the shortest weighted path from `start` to `end` in its execution-context graph.

For the graph in Figure 3.8, the acyclic execution context is `(alt:3)(p:8)(q:15)(n:19)`. As the edge weights correspond to the position of the edge in the sequence, multiple edges between two nodes are disambiguated by choosing the edge corresponding to the least recent function invocation.

Two distinct execution contexts that have the same AEC are the recursive analog of distinct iterations of a single loop. Unlike execution contexts that can grow unboundedly, AECs are bounded because the number of permutations of distinct functions in a program is finite. We found AECs to be a useful abstraction for clustering execution contexts of input-read events involved in a traversal—such an abstraction helps us to merge execution points involved in a traversal in a precise way irrespective of whether the traversal involves recursive calls or loop iterations.

Table 3.1 lists, for some example functions and program locations (column 1), the execution contexts (column 2) and corresponding AECs (column 3) for the first two input-read events, when the functions are provided an input linked list containing at least two nodes. The first row shows that the AECs for input-read events at line 8 in the function `third` are distinct, since `third` does not contain a traversal. The last three rows show that for the functions `len`, `contains` and `alt`, multiple input-read events at the given locations have a common AEC; therefore, they are traversing functions.

We can now refine the conditions that a traversing function should satisfy in terms of AECs as follows:

C1. At least two input-read events having same the AECs access different memory locations, and

C2. the memory locations involved in the input-read events either belong to the same object, or belong to different objects connected by a series of pointers.

We call the AEC of such input-read events a *traversal point*. In general, a traversing function may contain more than one traversal point.

3.2.3 Detecting Redundant Traversals

In order to determine if a traversal in a function is redundant, we need to analyze the sequence of concrete memory locations (i.e. actual memory addresses) read at a traversal point of the function. If the sequence contains repeated contiguous subsequences, then we know that the memory locations in these contiguous subsequences are traversed repeatedly. We then say the function has a redundant traversal. Formally, if the sequence of memory locations read at a traversal point can be partitioned into the contiguous subsequences $\beta_1, \beta_2, \ldots, \beta_k$ where $k \geq 2$ and for each $1 \leq i, j \leq k$, either β_i is a prefix of β_j or β_j is a prefix of β_i, then the sequence of memory locations indicate a possibly redundant traversal.

For example, if a, b and c are concrete memory locations, then the sequence of reads *abcaba* can be partitioned into repeating contiguous subsequences $(abc)(ab)(a)$ indicating redundant traversals. On the other hand, the sequence *abcacab* is partitioned as $(abc)(ac)(ab)$ and does not indicate a redundant traversal because *ac* is not a prefix of *ab* and vice versa.

Consider the execution of `containsAll` (Fig. 3.2) on an input linked list `list` containing the elements `['a', 'b', 'c']` and an array `arr` containing `['c', 'b', 'a']`. If the memory locations corresponding to the first three nodes of the linked list `list` are denoted as a, b, and c respectively, then the sequence of reads at AEC (`containsAll:5`)(`contains:8`) is *abcaba*, which can be partitioned into repeating contiguous subsequences as above; therefore, TRAVIOLI will identify `containsAll` as containing a potentially redundant traversal.

In general, a redundant traversal can be detected by a memory location sequence as short as *aba* or *aab*; therefore, TRAVIOLI can detect redundant traversals from functional unit tests alone. Moreover, TRAVIOLI can detect redundant traversals in functions that use recursion, such as the example in Figure 3.2, which could not be detected using prior approaches [131, 135].

3.3 Dynamic Analysis Implementation

TRAVIOLI identifies the traversing functions in a program by analyzing an execution of the program. TRAVIOLI first instruments the program under analysis to generate run-time

events. The instrumented program is executed with a suitable set of inputs to generate a trace of run-time events. From the generated trace, TRAVIOLI determines the input-read events for every function execution. TRAVIOLI then analyzes each sequence of input-read events to detect traversals. We next describe each of these steps formally.

3.3.1 Events and Traces

TRAVIOLI tracks reads and writes of every memory location during an execution of a program. In a program, a memory location can be denoted by a local variable, a global variable, a field of an object, or an element of an array. A memory location is represented by a pair (*obj*, *fld*), where *obj* is the address of an object (or array), and *fld* is the name of a field (or index of an array element). Local variables are treated as fields of special *activation record* objects corresponding to the stack frames in which they are allocated. Global variables are treated as fields of a special *globals* object.

TRAVIOLI instruments a program to generate the following four kinds of events:

1. READ$\langle \ell, obj, fld, val \rangle$ denotes the read of a memory location (*obj*, *fld*) at program location ℓ. The result of the read, *val*, can be a scalar or the address of another object.

2. WRITE$\langle \ell, obj, fld, val \rangle$ denotes the write of a memory location (*obj*, *fld*) at program location ℓ. Here, *val* is the new value that is written to the memory location. At function calls, write-events are generated for each argument passed to the function, where each formal parameter is treated as a local variable.

3. CALL$\langle \ell, f, a \rangle$ is an event corresponding to the invocation of function f at the program location (i.e. call site) ℓ. Here, a is a freshly generated unique identifier for the newly created activation record object for this function invocation.

4. RET$\langle \ell, a \rangle$ is an event corresponding to a function returning to its caller. Here, ℓ is the program location of the return instruction and a is the identifier of the current activation record, which is about to be destroyed. Note that each unique value of a appears in exactly one call and one return event in the program execution.

The execution of an instrumented program generates a trace of events. We identify the *execution of a function* started by the event CALL$\langle \ell, f, a \rangle$ by the activation record identifier a. For a function execution denoted by a, we use TRACE(a) to denote the sequence of events generated by the function execution, including the call and return events that start and end the execution of the function, respectively. If a function f' is invoked during the execution of a function f with activation record a, and if this invocation creates an activation record a', then TRACE(a') is a subsequence of TRACE(a).

3.3.2 Read-Traces and Read-Footprints

To compute the read footprint of a function execution, we need to determine the set of memory locations that are read before being written during the execution. We define the *read trace* of a function execution a, denoted by $\text{RTRACE}(a)$, as the largest set of events e_i such that:

- $e_i = \text{READ}\langle *, obj, fld, * \rangle$

- $e_i \in \text{TRACE}(a)$

- $\forall j : (e_j = \text{WRITE}\langle *, obj, fld, * \rangle) \in \text{TRACE}(a) \Rightarrow j > i$

The third condition ensures that if there is a write to the memory location (obj, fld) in the trace, then it must occur *after* e_i. Then, the *read footprint* of a function execution a, denoted by $\text{FP}(a)$, is computed as:

$$\text{FP}(a) = \{(obj, fld, val) \mid \text{READ}\langle *, obj, fld, val \rangle \in \text{RTRACE}(a)\}$$

3.3.3 Traversing Functions

Let f_X denote the execution of a function f with input X, where X represents the state of the entire program memory before such an execution, including the state of any arguments passed to f as parameters. A function f is a *traversing function* (ref. Definition 4) if and only if the following condition holds:

$$\forall X_1 : f_{X_1} \text{ halts}, \exists X_2 : |\text{FP}(f_{X_2})| > |\text{FP}(f_{X_1})|$$

3.3.4 Detecting Traversals

Theorem 1. The problem of determining if an arbitrary function contains a traversal is undecidable.

Proof. Assume we have a function called `traverses(f)` that determines if an input function `f` is a traversing function. Now, we can construct another function `halts` that takes as input another function `p` and some input `x` that and returns true if an only if `p` halts when provided with the input `x`:

```
1 function halts(p, x) {
2   var f = function(arr) {
3     p(x); // Must halt for 'arr' to be traversed
4     for (var i = 0; i < arr.length; i++) {
5       print(arr[i]);
6     }
7   }
8   return traverses(f); // True iff p(x) halts
9 }
```

But we know that the halting problem is undecidable. Hence, the function `traverses` cannot exist.

□

TRAVIOLI therefore uses heuristics to determine whether a function is a traversing function.

For every function execution a and for each event e in $\text{TRACE}(a)$, we compute the execution context of e with respect to a, denoted by $\text{EC}(a, e)$ as follows:

- If e is the first event of $\text{TRACE}(a)$ and is of the form $\text{CALL}\langle \ell, f, a \rangle$, then $\text{EC}(a, e) = \epsilon$, i.e. the empty sequence.

- If e is not the first event of $\text{TRACE}(a)$ and is generated at program location ℓ, and if the latest call-event before e without a matching return event before e is $e' = \text{CALL}\langle \ell', f', a' \rangle$, then $\text{EC}(a, e) = \text{EC}(a, e').(f', \ell)$, where $s.(f, \ell)$ is the sequence obtained by appending the pair (f, ℓ) to the sequence s.

This is a formal version of Definition 5 given in Section 3.2.2.

Once we have computed the execution context of an event with respect to a function execution, we determine its acyclic execution context as per Definition 6. Let us denote the acyclic execution context of an event e with respect to a function execution a by $\text{AEC}(a, e)$.

We define a reachability relation $\overset{a}{\rightsquigarrow}$ between objects accessed in function execution a: $o_1 \overset{a}{\rightsquigarrow} o_n$ holds if and only if there exists a sequence $(o_1, f_1, o_2), (o_2, f_2, o_3), \ldots (o_n, f_n, val)$, such that each element of the sequence is in the read footprint $\text{FP}(a)$. This relation is reflexive and transitive.

We can now formalize the conditions we check to detect if an execution a of function f contains a traversal: if there exist two input-read events $e_i = \text{READ}\langle \ell, obj_i, fld_i, val_i \rangle$ and $e_j = \text{READ}\langle \ell, obj_j, fld_j, val_j \rangle$ such that:

- $e_i, e_j \in \text{RTRACE}(a)$

- $(obj_i, fld_i) \neq (obj_j, fld_j)$

- $\text{AEC}(a, e_i) = \text{AEC}(a, e_j) = \alpha$

- $obj_i \overset{a}{\rightsquigarrow} obj_j$ or $obj_j \overset{a}{\rightsquigarrow} obj_i$

then we mark the function f as a traversing function and the AEC α as a traversal point. There may be more than one acyclic execution context marked as a traversal point for a function f across one or more of the function's executions.

3.3.5   Detecting Redundant Traversals

Consider an AEC α that is marked as a traversal point during the execution a. If we observed r events in TRACE(a) having AEC α with respect to a, then let the sequence of memory locations read in these events be $M = m_1, m_2, \ldots m_r$. To determine if this AEC is the point of a possibly redundant traversal (cf. Section 3.2.3), we perform the following steps:

1. We find all indexes i in $[1..r]$ where $m_i = m_1$, the first memory location in the sequence. If we find k such positions, and arrange them in increasing order, let the resulting sequence of positions be $p_1, p_2 \ldots p_k$. Naturally, $p_1 = 1$. If $k = 1$, then we quit early as there is no repetition. Otherwise, let $p_{k+1} = r$, as an upper bound.

2. We divide the read sequence into k partitions $\pi_1 \ldots \pi_k$. For all j in $[1..k]$, the jth partition π_j is the subsequence $m_{p_j}, m_{p_j+1}, m_{p_j+2}, \ldots m_{p_{j+1}-1}$. At least one partition must have a length greater than one, because the traversal criteria insists on at least two distinct memory locations.

3. For all pairs of distinct partitions π_i and π_j, if π_i is a prefix of π_j or if π_j is a prefix of π_i, then the complete sequence M consists of repeating subsequences. In that case, we mark α as a point of redundant traversal.

3.3.6   Access Graphs

TRAVIOLI can discover traversal points in functions that traverse input data structures. In order to identify the data structure being traversed, and to visualize the traversal across one or more AECs, we develop the concept of access paths and access graphs.

A memory location in a read footprint, which we call an *input-memory location*, can be reached from a program variable via a series of one or more fields or array indices, called an *access path*.

Definition 7 (Access Path). An *access path* π in a function execution is a finite non-empty sequence of the form $v.k_1.k_2 \cdots k_n$, where $n \geq 0$, v is a variable name, and each k_i is either a field name or an array index. The access path v represents the value of the variable v before the function execution starts, and the access path $\pi.k$ represents the value stored in the field or array index k of the object whose access path is π.

For example, the input-memory locations read by the function `third` in Figure 3.6 can be represented by access paths `list`, `list.next`, `list.next.next`, and `list.next.next.data`. More than one access path may refer to the same memory location.

Since traversing functions have read footprints that are unbounded, we found it useful to represent the unbounded set of access paths involved in a data-structure traversal using a finite graph, called an *access graph*. Figure 3.9 lists access graphs for various examples used in this chapter.

Definition 8 (Access Graph). In an *access graph*, nodes represent a set of values, which may be scalars or object addresses. There are two types of nodes: variable nodes and AEC nodes. A variable node with label v represents the value stored in variable v at the beginning of the function execution. An AEC node with label α represents the values read by an input-read event at AEC α. There is an edge with label k from any node n to an AEC node α if the field k of an object denoted by the n-node is read in an input-read event at the AEC α. If more than one field of objects represented by node n are read at the AEC α, then the edge from the n node to the α node is labeled with $*$. This happens when multiple elements of an array or multiple fields of an object are read at the AEC α.

According to this definition, variable nodes do not have incoming edges. Moreover, all AEC nodes are reachable from at least one variable node. An AEC node is colored grey if the corresponding AEC is a traversal point.

An access graph concisely captures the access paths of all input-memory locations read at each AEC. In particular, a path in the graph from a variable node v to an AEC node α corresponds to an access path that begins with v and is followed by the sequence of edge labels along the path in the access graph. For example, in Fig. 3.9a, the access path of an input-memory location read at AEC (`third:5`) is `list.next.next.data`. In Fig. 3.9b, the access graph contains a cycle. Therefore, the access paths of the input-memory locations read at AEC (`contains:5`) are `list.data`, `list.next.data`, `list.next.next.data`, and so on. In this manner, an access graph provides a bounded representation of an unbounded number of access paths.

Figures 3.9b, 3.9c and 3.9d represent access graphs of three functions that traverse linked lists in different ways, but the access graphs provide similar abstractions, because, in each case, the input lists are traversed via the `next` field at a single AEC. Figures 3.9e and 3.9f depict access graphs for functions that read array elements. In `addPair`, array elements are read at two distinct AECs; therefore, the graph contains two branches starting from `arr`. On the other hand, `sum` traverses the array and this is captured by the wild-card $*$ that labels the edge from `arr` to the AEC (`sum:5`). The access paths that reach this AEC are `arr.*`, which indicate that more than one field (or in this case, more than one array index) of the variable `arr` is read at the AEC (`sum:5`). Similarly, the access paths that are read at AEC (`sum:6`) are `arr.*.val`, which represent the `val` fields of the elements contained in the array `arr`.

Access graphs were first used in a static liveness analysis [98] to represent an unbounded set of heap-memory locations that may be live at a program point. Our access graphs are similar in that a node can represent a regular pattern of access paths. However, we distinguish nodes based on AECs rather than program locations as in the original formulation; therefore, our access graphs are context-sensitive.

We can use access graphs to determine access paths that identify the data structure being traversed. We call such an access path the *root* of the data-structure traversal.

Definition 9 (Root of a Data-Structure Traversal). In a traversing function f, a data-structure traversal *root* is a minimal access path π such that in the access graph of f, the

(a) Access graph for `third` (Fig. 3.6)

(b) Access graph for `contains`
(Fig. 3.1)

(c) Access graph for `alt` (Fig. 3.7)

(d) Access graph for `len`
(Fig. 3.4)

(e) Access graph for `addPair`
(Fig. 3.5)

(f) Access graph for `sum`
(Fig. 3.3)
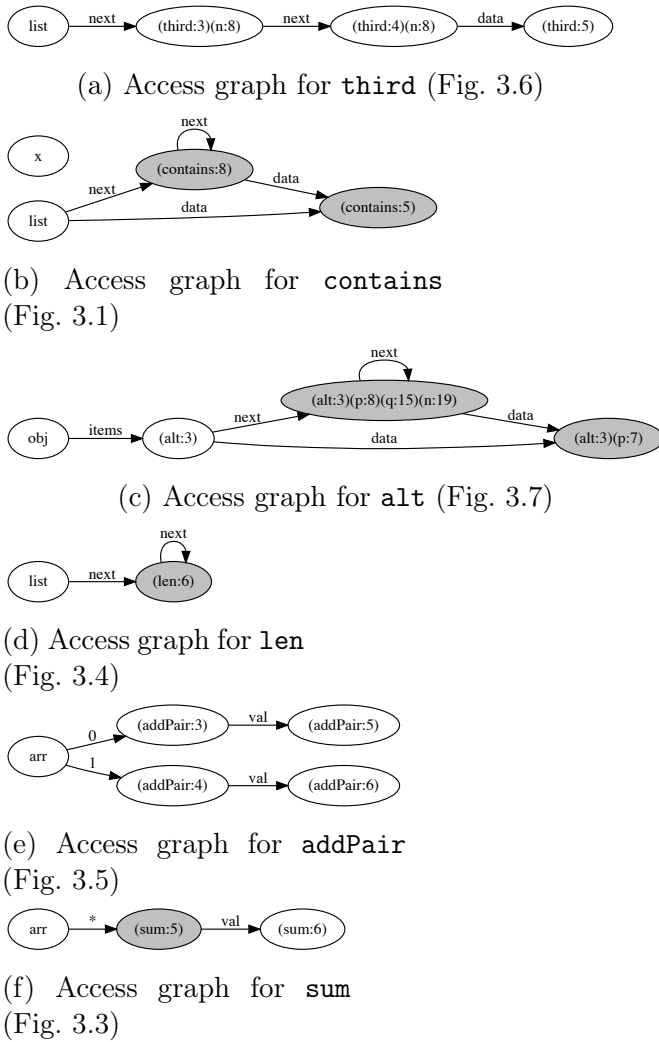
Figure 3.9: Access graphs for various examples presented in the chapter.

node n at the end of access path π satisfies the following two conditions: (1) there is an edge from n to a grey node and (2) there is no grey node along the path π.

Since π is a minimal access path that satisfies these conditions, no prefix of π is also a root. For example, the root of the data structure traversed in Fig. 3.9b is simply `list`, while in Fig. 3.9c the data structure being traversed is `obj.items`. Roots can be identified by determining the shortest path(s) from a variable node to an AEC node corresponding to a traversal point such that the path does not pass through any other traversal point.

Table 3.2: Overview of experiments conducted to evaluate TRAVIOLI.

Application	Test Suite			Analysis		
	Test Cases	Run. Time (normal)	Run. Time (w/instr)	Analysis Time	Function Invoca- tions	Unique Func.
(1)	(2)	(3)	(4)	(5)	(6)	(7)
d3-collection	233	0.21	7.95	3.88	1,340	37
immutable-js	418	0.65	81.12	149.69	260,642	513
d3-hierarchy	49	0.18	10.14	6.40	5,523	50
mathjs:matrix	357	0.59	43.80	44.33	26,931	282
express	696	2.12	81.09	91.52	53,382	158

Table 3.3: Results of experimental evaluation: traversals discovered by TRAVIOLI.

Application	All Traversals			Redundant Traversals		
	Unique Func.	Unique Roots	Unique AECs	Unique Func.	Unique Roots	Unique AECs
(1)	(8)	(9)	(10)	(11)	(12)	(13)
d3-collection	13	15	36	0	0	0
immutable-js	239	460	2,859	45	62	106
d3-hierarchy	20	23	88	1	1	1
mathjs:matrix	128	226	2,261	31	13	1,444
express	50	81	1,847	2	2	2

3.4 Evaluation

We have implemented TRAVIOLI using the Jalangi framework [163] for instrumenting JavaScript programs. We evaluate TRAVIOLI on a set of five open-source JavaScript projects. The projects were chosen because they are widely used, they have comprehensive unit tests that can be launched from command-line using Node.js [132], and they represent a variety of scenarios where data-structure performance may be important. The projects include `d3-collection` [47], a data-structure library used in the popular D3 [49] visualization toolkit, `immutable-js` [90], an immutable data-structure library developed by Facebook, `d3-hierarchy` [48], which provides algorithms for visualizing hierarchical data-sets, `express` [59], a server-side web framework, and `mathjs` [119], an extensive math library. We analyze the `matrix` module of `mathjs`.

Tables 3.2 and 3.3 provide an overview of experiments performed on a MacBook Pro with an Intel Core i7-4770HQ processor and 16GB RAM running OS X 10.10 and Node.js v4.4.0. All listed run-times are in seconds. In both tables, Column 1 lists the candidate projects studied. In Table 3.2, column 2 lists the number of unit tests in their test suites, column 3 reports the running time of the corresponding test suites, and column 4 reports the running time of the instrumented test suites, including the time to instrument the source files (project + dependencies) and the time to generate events. Columns 5–7 report the time required to analyze these events, the number of function executions analyzed for traversals, and the number of unique functions for which access graphs are generated respectively. Although

we compute the read trace for all function executions, we exclude analysis of functions from the projects' dependencies or test suites. In Table 3.3, columns 8–10 report the results of traversal detection: the number of traversing functions, the number of distinct access paths identified as roots of data structures (cf. Section 3.3.6), and the number of distinct AECs marked as traversal points. Columns 11–13 repeat this information for redundant traversals.

To evaluate the quality of results provided by TRAVIOLI, we perform the following manual investigation. For each candidate project, we randomly sample up to 10 access paths reported as roots of data structures being traversed, and randomly pick one reported traversal point for each access path. If a reported traversal point does not correspond to a traversing function within the library, we classify it as a *false positive*. In all other cases, the traversal point lies within a traversing function as per Definition 4, and is thus a true positive.

Of the 50 traversal points that were randomly sampled across all candidate projects, we found only two false positives: one in `immutable-js` and another in `express`. In `immutable-js`, an array data structure was incorrectly reported to be traversed. The false positive resulted from a related traversal of a hash-map that mapped strings to integer values; the resulting integers were used as indices to access a single element of different arrays. The array accesses occurred within the same loop that traversed the hash-map, and in at least two iterations a common array was accessed at the same AEC; therefore, the conditions that TRAVIOLI checks for detecting traversals were satisfied. In `express`, one traversal point was in the test suite itself, in a function that was supplied as a callback parameter to `express`. Since the traversal was not really part of `express` we marked this as a false positive.

Similarly, we manually evaluate a random subset of the traversal points that are reported as *redundant*. If the reported traversal point was not really redundant, we mark it as a *false positive*. All other redundant traversals are *true positives*. Now, a traversing function may be private to the library to which it belongs, and this library may use domain-specific constraints to ensure that the function receives inputs of only a bounded size. We classify true positives that belong to this category as *restricted traversals*. In such instances, even though a program function may appear to contain a redundant traversal, it will likely not become a performance bottleneck due to the constraints under which it is used. Finally, we classify the remaining true positives as either *bugs*—when the implementation performs more work than an optimal algorithm—or *necessary redundancies*—when the optimal algorithm necessarily requires repeated traversals of a data structure (e.g. matrix multiplication).

From the reported redundant traversals, we sampled 10 data structures and one corresponding traversal point from both `immutablejs` and `mathjs`. `d3-hierarchy` and `express` contained fewer than 10 reports of redundant traversals and we analyzed all of those cases. No redundant traversal was reported for `d3-collection`. We manually analyzed a total of 23 redundant traversals. Table 3.4 outlines the result of this manual evaluation.

All false positives were in `immutable-js`. The sequence of memory locations read at the reported traversal points did contain repeated contiguous subsequences, but this was specific to the particular inputs in the test suites. The corresponding traversing functions do not perform redundant computations in general.

Of the 19 sampled redundant traversals that were true positives, we found 10 to be

Table 3.4: Evaluation of sampled redundant traversal points reported by TRAVIOLI.

Application	False Positives	Restricted Traversals	Necessary Redundancies	Perf. Bugs	Total
d3-collection	0	0	0	0	**0**
immutable-js	4	6	0	0	**10**
d3-hierarchy	0	0	0	1	**1**
mathjs:matrix	0	3	7	0	**10**
express	0	1	0	1	**2**
Total	**4**	**10**	**7**	**2**	**23**

restricted traversals. For example, the implementation of maps in `immutable-js` uses `ArrayMap` with linear-time lookup only when the number of elements is less than 8; for larger maps the implementation switches to using hash-tables with constant-time lookup. In `express`, one reported redundant traversal was restricted because the traversing function can only ever be invoked internally with a list of HTTP methods (e.g. `GET`, `POST`), of which only 26 are supported; therefore, this function does not lead to performance issues. In `mathjs`, all reported redundant traversals belonged to algorithms that required repeated traversals, such as matrix multiplication, and thus were not classified as bugs.

Two of the reported redundant traversals were real performance bugs—they were confirmed by the developers. In `d3-hierarchy`, TRAVIOLI found a bug in the implementation of binary tree-maps, which are a visualization of hierarchical data as rectangles that are repeatedly partitioned into two sets. The implementation partitions an array of numbers by computing an index such that the sums of the left and right sub-arrays are approximately equal. This process is recursively repeated for each partition, resulting in a binary tree. We detected, from a simple unit test, that the algorithm to find the index to partition the array performed redundant traversals at each step to compute the sums of the sub-arrays. We were able to show that in the worst-case the implementation had complexity $\mathcal{O}(n^2)$. We reported and fixed this bug (see `https://github.com/d3/d3-hierarchy/issues/44`), by computing the sums of all prefixes of the input array ahead-of-time, and using a binary search to find the partition index at each step. The fixed implementation is $\mathcal{O}(n \log n)$ in the worst-case, and provides about a 20× speed-up for a binary tree-map with 1,000 nodes.

The second bug was found in `express`. When an `express` application is configured to support m URL patterns with n handlers using a particular API, the list of URL patterns is redundantly traversed once per handler to construct a regular expression that combines all patterns. As regex compilation is expensive, this implementation may lead to longer start-up times for some applications. We reported this as a performance issue, which was subsequently acknowledged by the developers (see `https://github.com/expressjs/express/issues/3065`).

3.5 Summary

In this chapter, we presented TRAVIOLI, a dynamic analysis technique to find algorithmic performance bottlenecks using only functional test cases. TRAVIOLI identifies potentially redundant data-structure traversals. TRAVIOLI's key innovation is *acyclic execution contexts*, which allows it to precisely identify complex ad-hoc data-structure traversals from analyzing the execution of small unit tests. TRAVIOLI was able to identify two algorithmic performance bugs in widely used JavaScript projects D3 and ExpressJS.

TRAVIOLI does suffer from several limitations. For example, TRAVIOLI reported several potentially redundant traversals that turned out not to be performance bugs either because they were not really traversals, because the program restricted traversal complexity by enforcing certain preconditions, or because the program uses algorithms that necessarily require super-linear complexity (e.g. matrix multiplication). Further, since TRAVIOLI uses dynamic analysis, it can only detect and report traversals if they occur during program execution. We cannot precisely evaluate TRAVIOLI for *false negatives*, because it is not possible to statically determine all traversal points, and it is not feasible to manually evaluate all candidate acyclic execution contexts. Finally, TRAVIOLI only points to program locations that may be performance bottlenecks due to potentially redundant data-structure traversals. Test cases corresponding to worst-case behavior must still be handcrafted in order to confirm or dismiss such reports. The next chapter presents a solution that overcomes this limitation.

Chapter 4

PERFFUZZ: Automatically Generating Pathological Inputs

The previous chapter described a dynamic analysis to identify likely algorithmic performance bottlenecks. Although the results were promising, the main limitation of this approach is that it does not automatically produce the inputs that trigger worst-case algorithmic complexity.

This chapter presents PERFFUZZ [107], a method to automatically generate *pathological inputs*; that is, inputs that exercise worst-case behavior, potentially revealing algorithmic performance bottlenecks. PERFFUZZ generates inputs via coverage-guided fuzzing (CGF). To recap Section 2.5, CGF generates new inputs by mutating a previously saved input, and new inputs are saved for future mutation if they execute a new program location (i.e. they increase code coverage in a program under test). The *key idea* in PERFFUZZ is to associate each program location with an input that exercises that location the most. Inputs that exercise some program location more than any previous input are saved and prioritized for subsequent mutation. This enables PERFFUZZ to find a variety of inputs that exercise distinct *hot spots* in a program, i.e., program locations that are frequently executed.

Our experimental evaluation demonstrates the ability of PERFFUZZ to find hot spots in four real-world C programs commonly used in the fuzzing literature. The inputs generated by PERFFUZZ exercise the most-frequently executed program branch $2\times$–$39\times$ times more often than the inputs generated by conventional coverage-guided fuzzing. We also compare PERFFUZZ with SlowFuzz [149], a prior work on discovering algorithmic complexity vulnerabilities. PERFFUZZ outperforms SlowFuzz in discovering inputs exercising worst-case algorithmic complexity in micro-benchmarks. PERFFUZZ is also better at generating pathological inputs in macro-benchmarks, finding inputs that exercise the most-frequently executed program branch $5\times$–$69\times$ times more and have $1.9\times$–$24.7\times$ longer execution paths. Unlike SlowFuzz, which tries to maximize only the total execution path length, PERFFUZZ uses multi-dimensional feedback and independently maximizes the number of times each program location is executed. However, the performance response of a program is not necessarily a convex function of its input characteristics. We believe the multi-dimensional objective helps PERFFUZZ escape local maxima, which explains its better performance.

```
1  // Hash-map entry node, with ptr        21  // Increments a word count
2  // to resolve hash collisions           22  void add_word(char* word) {
3  typedef struct entry_t {                 23    // access appropriate hash bucket
4    char* key;                             24    int bucket = compute_hash(word);
5    int value;                             25    entry* e = hashtable[bucket];
6    struct entry_t* next;                  26
7  } entry;                                 27    // find matching entry
8                                           28    while (e != NULL) {
9  // Fixed-size table of hash-map entries. 29      if (strcmp(e->key, word) == 0) {
10 const int TABLE_SIZE = 1001;             30        // increment count
11 entry* hashtable[TABLE_SIZE] = {0};      31        e->value++;
12                                          32        return;
13 // Computes a hash value for a word.     33      } else {
14 unsigned int compute_hash(char* str) {   34        // traverse linked list
15   unsigned int hash = 0;                 35        e = e->next;
16   for (char* p = str; *p != '\0'; p++){  36      }
17     hash = 31 * hash + (*p);             37    }
18   }                                      38    // If no entry found, create one
19   return hash % TABLE_SIZE;              39    hashtable[bucket] = new_entry(word,
20 }                                        40        1, hashtable[bucket]);
                                            40 }
```

Figure 4.1: Extract from a C program that counts the frequency of words in an input string.

4.1 A Motivating Example

The C program in Figure 4.1 is a simplified version of `wf` [185], a simple word frequency counting tool that is packaged in the Fedora 27 RPM repository. The main program driver (omitted from the figure for brevity) takes as input a string, splits the string into words at whitespaces, and counts how many times each word occurs in the input. To map words to integer counts, the program uses a simple hashtable (defined at Line 11) with a fixed number of buckets. Each bucket is a linked list of entries holding counts for distinct words that hash to the same bucket. As each word is scanned from the input, the program invokes the `add_word` function (Lines 22–40). This function first computes a hash value for that word—implemented in `compute_hash` (Lines 14–20)—and then attempts to find an existing entry for that word (Lines 28–37). If such an entry is found, its count is incremented (Line 31). Otherwise, a new entry is created with a count of `1` (Line 39).

When this program is used to compute word frequencies for an input containing English text, the program does not exhibit any performance bottlenecks. This is because English text usually contains words of short length (about 5 characters on average) and the number of distinct words is not very large (less than 10,000 in a typical novel). However, there are at least two performance bottlenecks that can be exposed by pathological inputs.

First, if the input contains very long words (e.g., nucleic acid sequences, a common genomics application), the program will spend most of its time in the `compute_hash` function. This is because the `compute_hash` function iterates over each character in the word irrespective of its length. For most applications, it is sufficient to compute a hash based on a

bounded subset of the input, such as a prefix of up to 10 characters.

Second, if the input contains many distinct words (e.g., e-mail addresses from a server log), the frequency of hash collisions in the fixed-size hashtable increases dramatically. For such an input, the program will spend most of its time in the function `add_word`, traversing the linked list of entries in the loop at lines 28–37. In the worst-case, the run-time of `wf` increases quadratically with the number of words. This bottleneck can be alleviated by replacing the linked list with a balanced binary search tree whenever the number of entries in a bucket becomes very large.

Now, how does the developer of this program identify these performance bottlenecks? If the inputs that exercised the behaviors outlined above were available, then they could run the program through a standard profiling tool such as GProf [79] or Valgrind [129] and observe the source locations where the program spends most of its time. They could also use a statistical debugging tool [176] to compare runs of inputs that take a long time to process versus inputs that are processed quickly. Alternatively, they could use an algorithmic profiling tool [200] to estimate the run-time complexity by varying the size of pathological inputs. But how does the developer acquire such inputs in the first place? Our performance fuzzing technique addresses exactly this concern.

Our goal is to generate inputs that independently maximize the execution count of each edge in the control-flow graph (CFG) of a program. We assume that we have one or more *seed inputs* to start with. These seeds are test inputs designed for verifying functional correctness of the program, and need not expose worst-case behavior. In our experiments, we use at most 4 seeds, but usually only 1. In the absence of such seeds, we can also simply start with arbitrary inputs such as an empty string or randomly generated sequences. The basic outline of our input-generation algorithm, called PERFFUZZ, is as follows:

1. Initialize a set of inputs, called the *parent inputs*, with the given *seed inputs*.

2. Pick an input from the parent inputs that maximize the execution count for some CFG edge.

3. From the chosen parent input, generate many more inputs, called *child* inputs, by performing one or more *random mutations*. These mutations include randomly flipping input bytes, inserting or removing byte sequences, or extracting random parts of another input in the set of parent inputs and splicing it at a randomly chosen location in the parent.

4. For each child input, run the test program and collect execution counts for each CFG edge. If the child executes some edge more times than any other input seen so far (i.e., it maximizes the execution count for that edge), then add it to the set of parent inputs.

5. Repeat from step 2 until a time limit is reached.

We walk through an execution of the PERFFUZZ algorithm for the word frequency counting program `wf` shown in Figure 4.1.

Suppose the seed input provided by a developer is the string `"the quick brown fox jumps over the lazy dog"`. This input consists of 9 words. This input does not have any special characteristics that exhibit worst-case complexity. All of the 8 distinct words in this input map to distinct buckets in the hashtable, and none are very long. PERFFUZZ first runs the program with this input and collects data about which CFG edges were executed. For example, the function `add_word` is invoked 8 times, whereas the `true` branch of the condition on Line 29 is executed only once to increment the count for the word `"the"`.

In step 2, PERFFUZZ picks this input and mutates it several times. Let us walk through a few sample mutations to observe the outcome of each mutation.

1. The character at position 18 is changed from `o` to `i`, yielding the string `"the quick brown fix jumps over the lazy dog"`. Running the program with this input does not increase the execution count for any CFG edge. Therefore, this input is discarded. This is the most common outcome of mutation.

2. The character at position 7 (the `i` in `quick`) is replaced with a space, yielding the string `"the qu ck brown fox jumps over the lazy dog"`. This operation increases the number of words, so running `wf` with this input leads to an additional execution of the function `add_word`. As no previous input has executed the CFG edge that invokes this function 10 or more times, the input is saved for subsequent fuzzing.

3. The character at position 16 (the space between `brown` and `fox`) is replaced with an underscore, yielding the string `"the quick brown_fox jumps over the lazy dog"`. The words `brown_fox` and `dog` have the same hash value of 545, causing a collision-resolving linked-list traversal at line 35. As this branch is executed for the first time, this input is also saved.

Note that the last mutation, (3), actually *reduces* the total number of words, and therefore the total end-to-end execution path length. This is important, and we will return to this point later.

Newly saved inputs will be picked in the future as the *parent* for subsequent mutations, and the process repeats. Inputs that maximize the execution count of at least one CFG edge are *favored*; that is, they are picked for fuzzing with higher probability. A favored input may cease to be favored if newer inputs are found with higher execution counts for the same edge. The number of favored inputs at any time is much smaller than the number of CFG edges in the program due to correlations between execution counts of various edges in the program—the same favored input may maximize the execution counts of correlated CFG edges.

Most mutated inputs will not increase execution counts. However, executing a program with a single input is a very fast operation, even in the presence of lightweight instrumentation for collecting profiling data. So, PERFFUZZ can make steady progress in a reasonable amount of time. For example, with our experimental setup, `wf` can be executed more than

6,000 times per second on average. Thus in one hour, PERFFUZZ can go through over 20 million inputs.

After a predefined time budget expires, PERFFUZZ outputs the current favored program inputs and the execution counts for the CFG edges that they maximize (see Table 4.1 for an example). For the running example, PERFFUZZ outputs strings including

"tvÇ1PFEj??A4A+v!^?^AE!§^?MPttð8dg80ÿ(8mrÿÿÿÿ",

a single long word which maximizes the execution count of Line 17 in `compute_hash`, as well as

"t t t t i nv t X t 1 9 t l t l t t t t t",

a string containing many short words which exercises repeated executions of the function `add_word()`, and

"t <81>v ^?@t <80>!^?@t <80>!t t^Rn t t t t t t t t t",

which contains many words that hash to the same bucket as the word "t", exposing the worst-case complexity due to repeated traversals of a long linked list. Section 4.3.1.2 describes in detail the results of running PERFFUZZ on `wf-0.41`.

An important feature of PERFFUZZ is that it saves mutated inputs if they maximize the execution count for any CFG edge, even if the mutation reduces the total execution path length. This is in contrast to previous tools which use a greedy approach and consider only increases in total path length [149]. This feature helps PERFFUZZ find inputs exercising worst-case behavior even when the performance response of the program is non-convex. For example, finding inputs with many hash collisions in the example above usually requires reducing the total path length when discovering the first few collisions—refer to mutation 3. But, the total path length becomes much larger once multiple collisions are found due to the quadratic increase in the number of linked-list node traversals. Empirical results in Section 4.3.1.2 support the importance of this multi-objective approach.

4.2 The PERFFUZZ Algorithm

Algorithm 2 describes the high-level outline of PERFFUZZ. The goal of PERFFUZZ is to generate inputs which achieve high *performance values* associated with some *program components*. To generate inputs exhibiting high computational complexity, we take the program components to be CFG edges and the values to be their execution counts. Chapter 6 describes a generalization of the ideas developed in this section that enable developing new domain-specific fuzzing applications.

Algorithm 2 is a modification of the CGF algorithm, with important changes from Algorithm 1 highlighted in grey. PERFFUZZ is given a program, p, and a set of initial seed inputs $\mathcal{I}$. These seed inputs are used to initialize a set $\mathcal{S}$ (Line 2). Inputs in set $\mathcal{S}$ form

the base from which new inputs are generated via mutation. PERFFUZZ also initializes two mutable data structures: *totalCoverage* (Line 3), an initially empty set that will track the code coverage achieved by fuzzed inputs, and *cumulmap* (Line 4), an initially zero-valued map that will track the maximum performance values observed for each program component.

PERFFUZZ then considers each input from the set $\mathcal{S}$ (Line 6) and probabilistically decides whether or not to select that input for mutational fuzzing (Line 7). The selection probability FUZZPROB is 1 for an input that is currently *favored* because it maximizes a performance value (see Definition 13 below) and a very small number otherwise.

PERFFUZZ then executes the program under test with every newly generated input (Line 9). During the execution, PERFFUZZ collects two types of feedback: (1) *coverage*, which includes code coverage information (e.g., *which* CFG edges were executed), and (2) *perfmap*, which maps numeric values to program components of interest (e.g., *how many times* each CFG edge was executed). If an execution results in new code coverage (Line 10) or if it maximizes the value for some component (Line 14), then the corresponding input is added to the set of inputs $\mathcal{S}$ for future fuzzing (Lines 11 and 15 respectively). Saving inputs which explore new coverage is key to exploring different program behavior when the program component, performance value pairs to be maximized are not simply CFG edges and their hit counts. Finally, the *cumulmap* map is updated to reflect the element-wise maximum of performance values observed for each program component, across all inputs generated so far (Line 18). Once PERFFUZZ completes a full cycle through the set $\mathcal{S}$, it simply repeats this process until a given time budget expires (Line 21).

We now define a series of concepts that are required to precisely describe what it means for an input to maximize a value associated with a program component (i.e., satisfy NEWMAX) and for an input to be *favored*.

Definition 10. A *performance map* is a function *perfmap* : $\mathcal{K} \rightarrow \mathcal{V}$, where $\mathcal{K}$ is a set of keys corresponding to program components and V is a set of ordered values ($\leq$) corresponding to performance values at these components.

Given a $\mathcal{K}$ and $\mathcal{V}$, *perfmap$_i$* is the performance map derived from the execution of input i on program p. The sets $\mathcal{K}$ and $\mathcal{V}$ have deliberately been left abstract to make the algorithm flexible; in our implementation, $\mathcal{K}$ is the set of edges in the program's control-flow graph and $\mathcal{V}$ is the set of non-negative integers that represent execution counts for each program location.

Definition 11. The *cumulative maximum map* at time step t is a function *cumulmax$_t$* : $\mathcal{K} \rightarrow \mathcal{V}$. It maps each program component to the maximum performance value observed for that component across all inputs generated up to time t. Precisely, if $\mathcal{I}_t$ is the cumulative set of inputs executed up to time step t, then:

$$\forall k \in \mathcal{K} : cumulmax_t(k) = \max_{i \in \mathcal{I}_t} perfmap_i(k).$$

Algorithm 2 The PERFFUZZ algorithm. Changes to Algorithm 1 are highlighted in grey.

Input: an instrumented test program p, a set of initial seed inputs $\mathcal{I}$
Output: a corpus of automatically generated inputs $\mathcal{S}$, a set of failing test inputs $\mathcal{F}$

1: $t \leftarrow 0$
2: $\mathcal{S} \leftarrow \mathcal{I}$
3: $totalCoverage \leftarrow \emptyset$
4: $cumulmax_t \leftarrow \lambda k.0$
5: **repeat** $\triangleright$ Main fuzzing loop
6: **for** i in $\mathcal{S}$ **do**
7:   **if** sample FUZZPROB(i) **then**
8:    $i' \leftarrow$ MUTATE(i) $\triangleright$ Generate new test input i'
9:    $coverage, perfmap \leftarrow$ RUN(p, i')
10:    **if** $coverage \cap totalCoverage \neq \emptyset$ **then**
11:     $\mathcal{S} \leftarrow \mathcal{S} \cup \{i'\}$ $\triangleright$ Save i' if new code coverage achieved
12:     $totalCoverage \leftarrow totalCoverage \cup coverage$
13:    **end if**
14:    **if** NEWMAX($perfmap, cumulmax_t$) **then**
15:     $\mathcal{S} \leftarrow \mathcal{S} \cup \{i'\}$ $\triangleright$ Save i' if it maximizes a perf value
16:    **end if**
17:    $t \leftarrow t + 1$
18:    $cumulmax_t \leftarrow \lambda k. \max(cumulmax_{t-1}(k), perfmap(k))$
19:   **end if**
20: **end for**
21: **until** given time budget expires

The first key to the PERFFUZZ algorithm is saving inputs which achieve a new maximum compared to previously observed values (Lines 14–15 in Algorithm 2). The function NEWMAX is defined as follows:

Definition 12. The function NEWMAX will return true for a newly generated input i at time step t if and only if the following condition holds:

$$\exists k \in \mathcal{K} \text{ s.t. } perfmap_i(k) > cumulmax_t(k).$$

The second key to the PERFFUZZ algorithm is the selection of inputs from $\mathcal{S}$ to mutate. To define the selection probability of an input, FUZZPROB, we must first define the concept of *favoring*.

Definition 13. An input i *maximizes* a performance value for some component k if and only if its performance profile registers the maximum value observed for that component so far:

$$maximizes_t(i, k) \Leftrightarrow perfmap_i(k) = cumulmax_t(k).$$

An input i is *favored* by PERFFUZZ at time step t if and only if it maximizes a performance value for some component. The favoring mechanism is a heuristic that allows PERFFUZZ to prioritize fuzzing those inputs that maximize the performance value of some program component. The intuition behind this is that these inputs contain some characteristics that lead to expensive resource usage in some program components. Thus, new inputs derived from them may be more likely to contain the same characteristics. With this, we can define the probability that an input will be selected as a parent for fuzzing:

Definition 14. The *selection probability* of an input i at time t is:

$$\text{FUZZPROB}_t(i) = \begin{cases} 1 & \text{if } \exists k \in \mathcal{K} \text{ s. t. } maximizes_t(i,k) \\ \alpha & \text{otherwise} \end{cases} .$$

That is, favored inputs are always selected, and α is the probability of selecting a non-favored input. In our experiments we use $\alpha = 0.01$.

4.2.1 Implementation

PERFFUZZ is implemented as a fork of AFL (ref. Section 2.5.1). In our implementation, the *performance map* sent back to the program has $\mathcal{K} = \mathcal{E} \cup \{total\}$ and $\mathcal{V} = \mathbb{N}$, where $\mathcal{E}$ is the program's set of CFG edges and *total* is an additional key. For an input i, for each $e \in \mathcal{E}$, $perfmap_i(e)$ is the total number of times the program executes e when run on input i, and $perfmap_i(total) = \sum_{e \in \mathcal{E}} perfmap_i(e)$. The purpose of the *total* key is to save inputs which have high total path length.

To produce this performance map, we simply augmented AFL's *LLVM-mode* instrumentation, which inserts the coverage instrumentation described above into LLVM IR. Our augmented instrumentation still creates the usual coverage map, whose keys are in $\mathcal{E}$ and whose values are their 8-bit hit counts. Additionally, our augmented instrumentation creates the performance map outlined above, with values as 32-bit integers.

The implementation of PERFFUZZ is open-source and available at the following URL: `https://github.com/carolemieux/perffuzz`.

4.3 Evaluation

In our evaluation of PERFFUZZ, we seek to answer the following research questions:

RQ1. How does PERFFUZZ compare to single-objective complexity fuzzing techniques such as SlowFuzz [149]?

RQ2. Is PERFFUZZ more effective at finding pathological inputs than fuzzing techniques guided only by coverage?

RQ3. Does the multi-dimensional objective of PERFFUZZ help find a range of inputs that exercise distinct hot spots?

We chose four widely used C libraries (with appropriate entry points) as benchmarks for our main evaluation: (1) `libpng-1.6.34`, (2) `libjpeg-turbo-1.5.3`, (3) `zlib-1.2.11`, and (4) `libxml2-2.9.7`. We chose these benchmarks as they are (a) common benchmarks in the coverage-guided fuzzing literature (b) fairly large—from 9k LoC for `zlib` and 30k LoC for `libpng` and `libjpeg`, to 70k LoC for `libxml`—and (c) had readily-available drivers for `libFuzzer`, an LLVM-based fuzzing tool [111]. The availability of good `libFuzzer` drivers was key to being able to fairly compare PERFFUZZ to SlowFuzz [149] in Section 4.3.1. While AFL-based tools need only a program that accepts standard input or an input-file name, `libFuzzer`-based tools rely on a specialized driver that directly takes in a byte array, does not depend on global state, and never exits on any input. Creating drivers with this second characteristic from command-line programs is especially tricky. The particular drivers we chose (from the OSS-fuzz project [2]) exercised the PNG read function, the JPEG decompression function, the ZLIB decompression function, and the XML read-from-memory function.

For each of these benchmarks, we ran PERFFUZZ (and the tools with which we compare it) for 6 hours on a maximum file size of 500 bytes. AFL ships with sample seed inputs in formats including PNG, JPEG, GZIP and XML; we simply used the same inputs as seeds for our evaluation. We chose the maximum size of 500 bytes as it was an upper bound on all the seeds that we considered. As the fuzzing algorithm used by PERFFUZZ as well as other tools is non-deterministic, we repeated each 6-hour run 20 times to account for variability in the results.

For our evaluation on discovering worst-case algorithmic complexity as a function of varying input sizes (Section 4.3.1.2), we used three micro-benchmarks: (1) insertion sort (because it was provided as the default example in the SlowFuzz repository), (2) matching an input string to a URL regex [28] using the PCRE library, and (3) `wf-0.41` [185], a simple word-frequency counting tool found in the Feodra Linux repository.

To evaluate PERFFUZZ against other techniques, we measure one or both of the *maximum path length* and the *maximum hot spot*, where appropriate. More precisely, if $\mathcal{E}$ is the set of CFG edges in the program under test, and $\mathcal{I}_t$ is the set of inputs generated by a fuzzing tool up to time t, then:

Definition 15. The *maximum path length* is the longest execution path across all inputs generated so far.

$$\text{max. path length} = \max_{i \in \mathcal{I}_t} \sum_{e \in \mathcal{E}} perfmap_i(e).$$

Definition 16. The *maximum hot spot* is the highest execution count observed for any CFG edge across all inputs generated so far.

$$\text{max. hot spot} = \max_{i \in \mathcal{I}_t} \max_{e \in \mathcal{E}} perfmap_i(e).$$

These two values allow us to get a grasp of the overall computational time complexity of generated inputs (the path length) as well as whether it is driven by a particular program component (the hot spot) without having to look at the entire distribution of execution counts of CFG edges, which is not practical to do over time.

4.3.1 Comparison with SlowFuzz

SlowFuzz [149] is a fuzz testing tool whose main goal is to produce inputs triggering algorithmic complexity vulnerabilities. Like PERFFUZZ, SlowFuzz is also an input-format agnostic fuzzing tool for C/C++ programs; therefore, we believe it is the most closely related work to practically compare against.

The objective of SlowFuzz is one-dimensional: to maximize the total execution path length for a program. As such, it serves as an important candidate for evaluating the coverage-guided multi-objective maximization of PERFFUZZ against a traditional single-objective technique.

There are two other main algorithmic differences between SlowFuzz and PERFFUZZ. First, while PERFFUZZ prioritizes inputs to fuzz using the concept of *favored* inputs (Line 7 of Algorithm 1), SlowFuzz randomly selects a parent input to fuzz. Second, PERFFUZZ applies AFL's *havoc* mutations to the input. SlowFuzz learns which mutations were successful in producing slow inputs in the past, and applies these more often.

Finally, SlowFuzz is built on top of on `libFuzzer` (ref. Section 2.5.1). In practice, `libFuzzer` is faster than AFL, running more inputs through the program per second; therefore, SlowFuzz usually produces more inputs than PERFFUZZ in the same time span. Nonetheless, in our evaluation, we run both PERFFUZZ and SlowFuzz for the same amount of time.

We compare PERFFUZZ with SlowFuzz on two fronts. First, we evaluate PERFFUZZ and SlowFuzz on their ability to maximize total execution path lengths as well as the maximum hot spot on the four macro-benchmarks described above. Second, we compare the ability of PERFFUZZ and SlowFuzz to find inputs that demonstrate worst-case algorithmic complexity in micro-benchmarks which are known to have worst-case quadratic complexity.

In all runs of SlowFuzz, we used the arguments provided in the `example` directory, except that we used the "hybrid" mutation selection strategy. This was the strategy used in SlowFuzz's own evaluation [149], and we found that it performed best on a selection of micro-benchmarks in our initial experiments.

4.3.1.1 Maximizing Execution Counts

Figure 4.2 shows the progress made by PERFFUZZ and SlowFuzz during 6-hour runs in maximizing total path length (on the left) and the maximum hot spot (on the right). The lines in the plot represent average values over 20 repeated 6-hour runs, while the shaded areas represent 95% confidence intervals, calculated with *Student's t-distribution*.

(a) libpng - max. path length

(b) libpng - max. hot spot

(c) libxml2 - max. path length

(d) libxml2 - max. hot spot

(e) libjpeg - max. path length

(f) libjpeg - max. hot spot

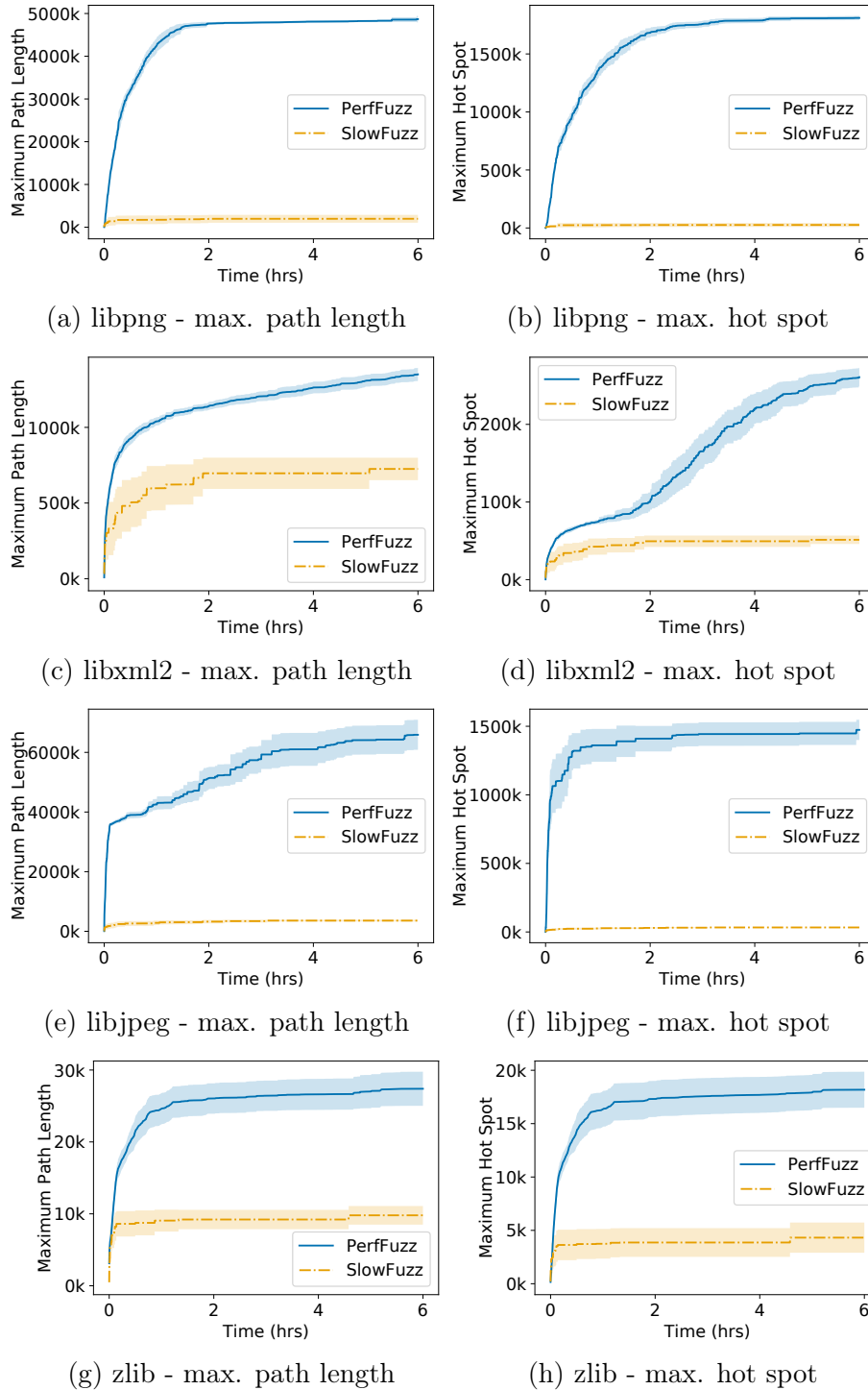(g) zlib - max. path length

(h) zlib - max. hot spot

Figure 4.2: PERFFUZZ vs. SlowFuzz on macro-benchmarks: maximum path length and maximum hot spot found throughout the duration of the 6-hour fuzzing runs. Lines and bands show averages and 95% confidence intervals across 20 repetitions; higher is better.

It is clear from Figure 4.2 that PERFFUZZ consistently finds inputs that are significantly worse-performing than SlowFuzz's by both the evaluated metrics—the maximum path lengths found by PERFFUZZ are $1.9\times$–$24.7\times$ higher and the maximum hot spots are $5\times$–$69\times$ higher. This is in spite of the fact that SlowFuzz produces more inputs in each of this 6-hour runs (from $1.7\times$ more for `libxml2` to $17.7\times$ more for `libjpeg-turbo`).

The results show that not only is PERFFUZZ better than SlowFuzz at finding hot spots, for which the PERFFUZZ algorithm is tailored, but that PERFFUZZ is superior to SlowFuzz even for finding inputs that maximize total path length, for which SlowFuzz is tailored. Intuitively, we believe that this is because the total path length is not a convex function of input characteristics; a greedy approach to maximizing total path length is likely to get stuck in local maxima. In contrast, PERFFUZZ saves newly generated inputs even if the total path length is lower than the maximum found so far, as long as there is an increase in the execution count for some CFG edge. Thus, the multi-dimensional objective of PERFFUZZ allows it to perform better global maximization of total path lengths.

4.3.1.2 Algorithmic Complexity Vulnerabilities

SlowFuzz was designed to find algorithmic complexity vulnerabilities, where programs exhibit worst-case behavior that is asymptotically worse than their average-case behavior. Such programs pose a security risk if they process untrusted inputs: an attacker can send carefully crafted inputs that exercise worst-case complexity and exhaust the victim's computational resources, resulting in a Denial-of-Service (DoS) attack [45]. We now show that PERFFUZZ can also address this use case, and in fact can out-perform SlowFuzz in some cases.

We considered three micro-benchmarks: (1) insertion sort on an array of 8-bit integers, which is the only benchmark provided in the SlowFuzz repository, (2) matching an input string against a regular expression to validate URLs using the PCRE library, and (3) `wf-0.41`, the word-frequency counting program from the Fedora Linux repository. These benchmarks are very similar to those used to evaluate SlowFuzz. Each of these micro-benchmarks have an average-case run-time complexity that is linear in the size of the input, and a worst-case complexity that is quadratic.

For each of these benchmarks, we varied the upper bound on the input size between 10 and 60 bytes with 10-byte intervals. We then ran each tool on the micro-benchmarks for a fixed duration: 10 minutes for insertion sort and 60 minutes for PCRE and `wf`. In all cases, we provided a single input seed: a sequence of zero-valued bytes of maximum length for insertion sort and PCRE (these represent trivial base cases), and (truncations of) the string "`the quick brown fox jumps over the lazy dog`" for `wf`, as it leads to average-case performance. For each input length, we performed 20 runs to account for variability. Finally, we measured the maximum path length observed over all the inputs produced in these runs.

Figure 4.3 shows the results of these runs: points plot the average maximum path length, while lines show 95% confidence intervals.
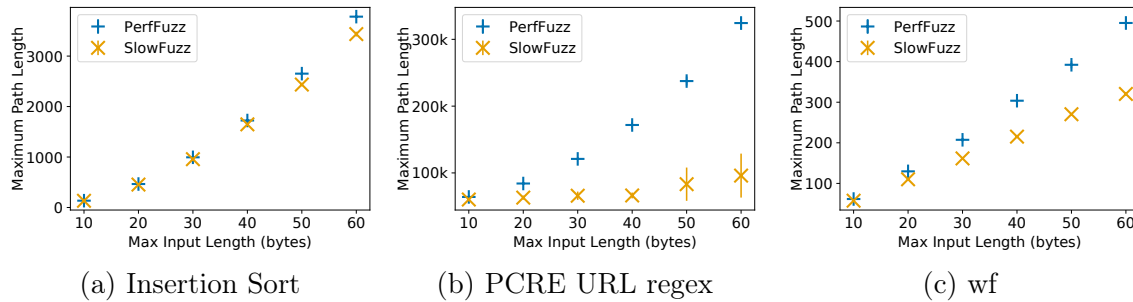
Figure 4.3: PERFFUZZ vs. SlowFuzz on micro-benchmarks: maximum path length found with given time budget, for varying input sizes; higher is better.

For insertion sort, for all input lengths, PERFFUZZ found a significantly (at 95% confidence) longer maximum path length, but as Figure 4.3a shows, the difference is minimal for small input lengths. For input lengths 10 and 20, PERFFUZZ consistently found the worst-case—a reverse-sorted list—while SlowFuzz had non-zero variance in its results. Figure 4.3a also shows that for larger input sizes, PERFFUZZ finds lists that require more comparisons to sort than SlowFuzz. Overall, both tools discover the worst-case quadratic time complexity for this benchmark.

However, in Figure 4.3b we see a major difference between the worst-case inputs found by PERFFUZZ and SlowFuzz on the PCRE URL benchmark. PERFFUZZ finds inputs that lead to worst-case quadratic complexity, while SlowFuzz finds only a slight super-linear curve. An example of a 50-byte input found by PERFFUZZ in one of the runs was:

<div align="center">

`fhftp://ftp://ftp://ftp://f.m.m.m.m.m.m.m.m.m.m.`

</div>

This is remarkable because the seed input was an empty string and PERFFUZZ was not provided any knowledge of the syntax of URLs. On the other hand, SlowFuzz has difficulty in automatically discovering substrings such as `ftp` in the input string. We suspect that this is because of its one-dimensional objective function, which does not allow it to make incremental progress in the regex matching algorithm unless there is an increase in *total* path length. Additionally, Figure 4.3b shows that there is much more variance in SlowFuzz's performance (see large confidence intervals for length 50 and 60) on this benchmark, indicating that any such progress likely relies on a sequence of improbable random mutations.

`wf` is a much harder benchmark, as the worst-case behavior is only triggered when distinct words in the input string map to the same hash-table bucket (ref. Section 4.1). Figure 4.3c shows that PERFFUZZ clearly finds inputs closer to worst-case time complexity in the given time budget. We noticed that in nearly all runs (i.e., 19 of the 20 runs for 60-byte inputs), PERFFUZZ produced inputs with a very peculiar structure: first a few distinct words with the same hash code, then a single 1-letter word repeated multiple times. For example, PERFFUZZ generated this input in one of its runs:

<div align="center">

`t <81>v ^?@t <80>!^?@t <80>!t t^Rn t t t t t t t t`

</div>

Table 4.1: A snapshot of the output of PERFFUZZ after one 6-hour run on `libpng`. For each of 3 favored inputs, the table shows the top 3 CFG edges—represented by start and end line numbers—by their execution count.

Input #9189		Input #10520		Input #10944	
Count	CFG edge	Count	CFG edge	Count	CFG edge
2,071,824	`pngrutil.c:3715->3715`	289,536	`pngrutil.c:3842->3842`	225,489	`pngread.c:387->396`
274,212	`pngrutil.c:3715->3712`	144,536	`pngrutil.c:3416->3419`	225,489	`pngread.c:405->456`
274,178	`pngrutil.c:3712->3715`	144,536	`pngrutil.c:3419->3404`	225,489	`pngread.c:456->459`

What is amazing about this input is how precisely it exercises worst-case complexity. First, a small word is inserted into some hash bucket. Then, the next few words have the exact same hash code and are inserted at the front of the linked list in that bucket; the first word is now the last node in this linked list. Finally, the repeated occurrences of the first word cause `wf` to traverse the entire linked list multiple times. The worst inputs produced by SlowFuzz had some hash collisions, but still had several different hash codes and no traversal-stressing structure like the input above.

Overall, we see that in the same time constraints, PERFFUZZ is able to find inputs with significantly longer paths than SlowFuzz, and can out-perform SlowFuzz in discovering inputs exercising near worst-case algorithmic complexity.

4.3.2 Comparison with Coverage-Guided Fuzzing

With the insight that PERFFUZZ's efficacy is in part due to its multi-objective. coverage-guided progress, we ask whether PERFFUZZ performs better than just AFL off-the-shelf. To evaluate this aspect, we ran AFL on our four C macro-benchmarks. Like PERFFUZZ, AFL was configured to use only havoc mutations (`-d` option), because this configuration has been shown to result in faster program coverage [198]. This experiment tests the value-add of PERFFUZZ's performance maps and maximizing-input favoring heuristics.

We begin by looking at the evolution of the maximum hot spot found by each technique through time, shown in Figure 4.4. For the `libpng`, `libjpeg-turbo`, and `zlib` benchmarks (Figures 4.4a, 4.4c, 4.4d), we see that PERFFUZZ rapidly finds a hot spot with a significantly higher execution count. For the `libxml2` benchmark (Figure 4.4b), AFL initially finds a hot spot with higher execution count, but quickly plateaus. On the other hand, PERFFUZZ finds a hot spot with over $2\times$ higher execution count after 6 hours. Overall, Figure 4.4 demonstrates that PERFFUZZ's performance-map feedback has a significant effect on its ability to generate pathological inputs, exercising hot spots with $2\times$–$18\times$ higher execution counts.

Figure 4.4 shows only the execution counts for the maximum hot spot, as this is easy to visualize through time. However, we were curious as to whether the maximum execution
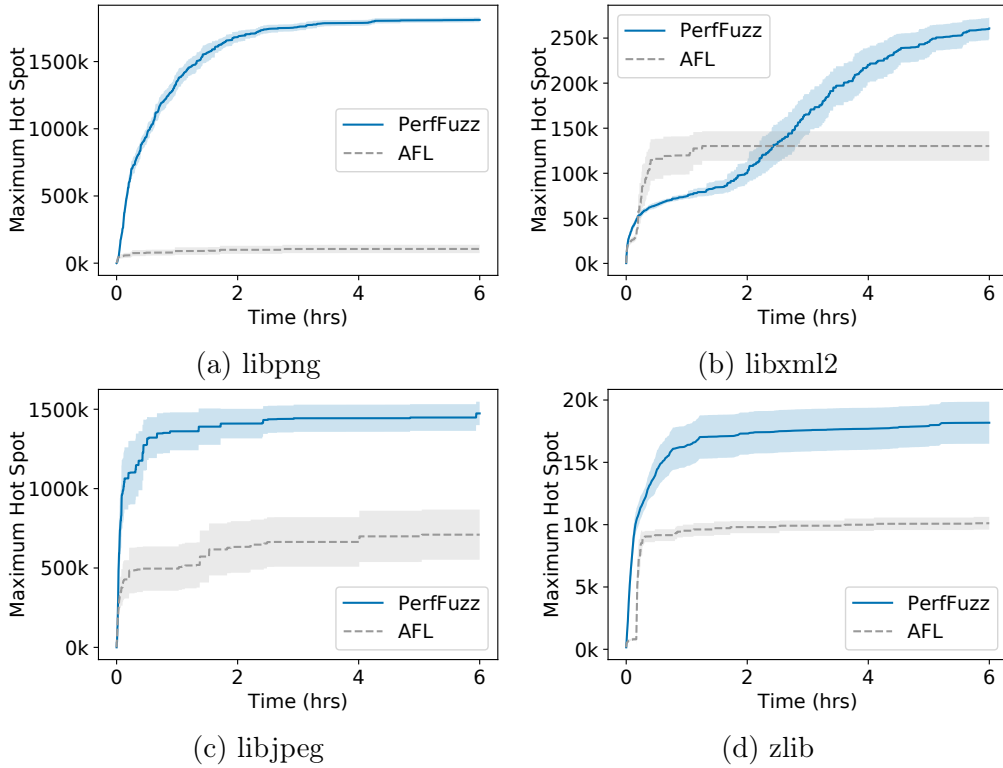
Figure 4.4: PERFFUZZ vs. AFL: Time evolution of the maximum hot spot through the 6-hour runs. Lines and bands show averages and 95% confidence intervals across 20 repetitions. Higher is better.

counts found by PERFFUZZ are significantly higher than those found by AFL over all hot spots in the program. Figure 4.5 provides this information.

In particular, Figure 4.5 shows the maximum execution count per CFG edge found by each technique at the end of the 6 hour runs. We plot the median of this measure across the 20 repeated runs. For clarity, we sort the CFG edges by the counts achieved by PERFFUZZ and truncate the data to show only those edges with execution counts within 2 orders of magnitude of the maximum hot spot found by PERFFUZZ. The omitted tails of the distributions are indistinguishable. Figure 4.5 confirms that PERFFUZZ's gains are not limited to only the maximum hot spot in the program. Across the four benchmarks, there are 453 of the plotted edges which PERFFUZZ-generated inputs exercise over 2x more times than AFL-generated inputs, and 238 edges which PERFFUZZ-generated inputs exercise over 10x more times.

(a) libpng

(b) libxml2

(c) libjpeg-turbo

(d) zlib
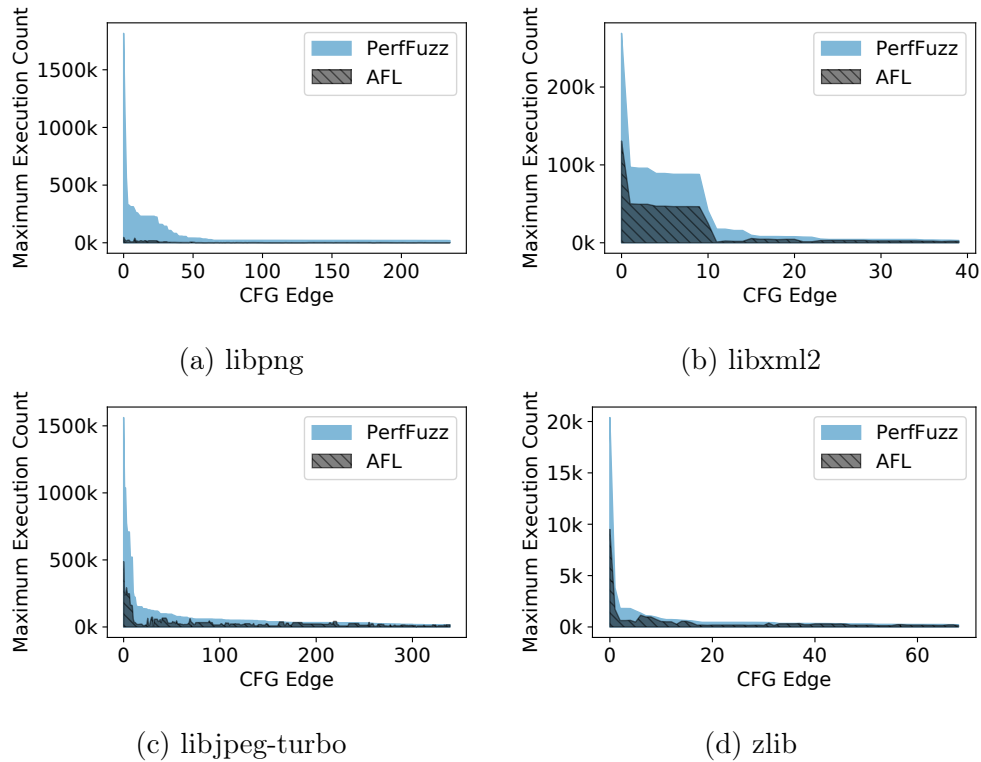
Figure 4.5: Distribution of maximum execution counts across CFG edges, as found by by PERFFUZZ and AFL after 6-hour runs. Plots show median values of this measurement across 20 repetitions.

4.3.3 Case Studies

PERFFUZZ is designed to generate inputs that demonstrate pathological behavior in programs across different program components (in this evaluation, CFG edges). In Section 4.3.1.2 we saw that the inputs generated by PERFFUZZ exercised close-to worst-case algorithmic complexity on micro-benchmarks. We decided to manually analyze the inputs generated by PERFFUZZ—in a single run each—on the four macro-benchmarks to see where the hot spots were located and how different input characteristics affected these hot spots.

At the end of each run, PERFFUZZ outputs its set of favored inputs—those that maximize the execution count of at least one CFG edge—as well as the execution counts for each CFG edge that it maximizes. Table 4.1 shows an example of this output: it is a snippet from the results obtained from one run of PERFFUZZ on the `libpng` benchmark, showing the top 3 CFG edges by execution count for the top 3 favored inputs.

```
           void png_do_read_interlace(png_row_infop row_info, ...) {
              ...
              switch (row_info -> pixel_depth) {
                  case 1:
                  {
                     for (i = 0; i < row_info ->width; i++)
 3715:                   for (j = 0; j < jstop; j++)
                            ...
                  }
                  ...
                  case 4:
                  {
                     for (i = 0; i < row_info ->width; i++)
 3842:                   for (j = 0; j < jstop; j++)
                            ...
                  }
              }
           }
```

Figure 4.6: Snippet from `pngrutil.c` showing hot spots which can only be exercised by inputs with distinct features.

4.3.4 `libpng`

From Table 4.1, we can directly look at the source code locations to see which features each input exercises. This alone already highlights different hot spots in the code. For illustration, we look at a snippet from `pngrutil.c` in Figure 4.6, which shows an excerpt from a function that performs PNG interlacing. The argument `row_info` contains data parsed from the input file. This snippet of code shows two distinct hot spots—sets of input-dependent nested loops—guarded by a `switch` on an input characteristic. Therefore, these hot spots can only be exercised by distinct inputs. As illustrated in Table 4.1, input #9189 maximizes the number of executions of the inner loop when pixel depth is 1 (Line 3715 of Figure 4.6), corresponding to a monochrome image. Input #10520, on the other hand, maximizes executions of the inner loop for a pixel depth of 4 (Line 3842 of Figure 4.6), corresponding to an image segment with 16 color-palette entries. Other inputs stress completely different parts of the code. For example, input #10944 from Table 4.1 maximizes execution counts for CFG edges in a loop whose bounds are proportional to the height of the PNG image, as declared in the PNG header: each iteration processes one row of pixels at a time.

From a quick glance at just three favored inputs, we can see that PERFFUZZ has enabled us to discover some of the key features which have an effect on the performance of parsing a PNG image independent of the file size, such as the image's geometric dimensions and color depths declared in the header. We repeat this exercise for the other benchmarks, but omit the actual outputs and code snippets for brevity.

4.3.5 `libjpeg-turbo`

In the `libjpeg` benchmark, we saw a similar distribution of inputs where the hot spots were related to JPEG image properties. For example, one input's hot spot was in processing for an image with $4:4:0$ chroma sub-sampling; the input also had a huge number of columns. Other inputs stressed various points in the arithmetic decoding algorithms. PERFFUZZ discovered inputs that stressed processing for both one-pass and multi-pass images.

4.3.6 `zlib`

Compared to image formats, the functionality of the `zlib` decompressor is relatively straight-forward. This was reflected by the fact that there were very few edges exercised a huge number of times; that is, there were fewer hot spots. Nonetheless, PERFFUZZ discovered an input with a compression factor of nearly $126\times$, whose processing lead to a long execution path.

4.3.7 `libxml`

The inputs produced by PERFFUZZ for the `libxml2` benchmark revealed what appears to be quadratic complexity in the parsing process. The largest hot spot was the traversal of the characters of a string in a string-duplication function. For a 500 byte input, there were 226,512 iterations of this loop. By running the input, it was quickly apparent that the source of this quadratic complexity came from repeatedly printing out the context of errors in the input. Naturally, inputs generated by random mutation are not well-formed XML files. In fact, these inputs had so many errors that they caused the same work—printing the error context—to be done over and over again. PERFFUZZ also stressed error handling code that repeatedly traversed the input backwards to check whether a parent tag had a given name-space; essentially, PERFFUZZ learned to produce errors deep in the XML tree, causing pathological behavior.

These case studies indicate that the inputs generated by PERFFUZZ lead to non-trivial hot spots being uncovered. The inputs generated for `libxml2` also reveal potential inefficiencies in the program performance. Overall, this analysis suggests that PERFFUZZ successfully produces inputs that stress various program functionalities, and may be useful by themselves or as references for creating performance tests on these benchmarks.

4.3.8 Google Closure Compiler

We have also implemented a version of PERFFUZZ for testing Java programs using the JQF framework (described in Section 5.3). Using PERFFUZZ, we were able to generate pathological inputs for the Google Closure Compiler [78], which optimizes JavaScript programs. Some of the inputs generated by PERFFUZZ resulted in timeouts; parsing a small 60-byte JavaScript program required more than 10 seconds. Upon further investigation,

we found that the Closure Compiler's parser had worst-case exponential complexity, which was certainly suboptimal. We reported the performance bug to the developers, who soon acknowledged the issue[1].

4.4   Threats to Validity

Like many other input generation techniques founded on evolutionary search, PERFFUZZ relies solely on heuristics to produce inputs that achieve its testing goal, which is to exercise pathological program behaviors. In combination with the fact that PERFFUZZ is a dynamic technique, this means that PERFFUZZ is not guaranteed to find all hot spots in a program or the absolute worst-case behavior for each hot spot it discovers.

In this chapter, we focused on discovering bottlenecks due to increase in computational complexity; therefore, we measure execution counts of CFG edges instead of total running time. This helps ensure that our measurements are accurate and deterministic, but also means that the identified bottlenecks may not be the points in which the program spends the most time. This gap could be mitigated by using a different cost model for CFG edges, i.e. to find bottlenecks due to other factors such as I/O operations. Chapter 6 presents a framework that makes it easy to prototype such fuzzing applications.

Finally, we believe that the reason that PERFFUZZ outperforms greedy techniques such as SlowFuzz is due to the ability to overcome local maxima in a non-convex performance space. Although we have anecdotal evidence to back this intuition, such as the observations with the `wf` tool described in Section 4.1, we have not mapped the performance spaces of our benchmarks to measure their convexity. Doing this would require searching through all possible mutations from each generated input, which is infeasible.

4.5   Summary

In this chapter, we presented PERFFUZZ, an algorithm for generating inputs that exercise pathological behavior in various program components, using only functional test cases as a starting point. Like fuzz testing tools such as AFL, PERFFUZZ is input-format agnostic. We empirically evaluated the efficacy of PERFFUZZ in generating pathological inputs in comparison with that of SlowFuzz [149], a similar feedback-directed fuzzing tool designed to find algorithmic complexity vulnerabilities. PERFFUZZ's multi-dimensional feedback allows it to escape local maxima and generate inputs that exercise the most-frequently executed program branch $5\times$–$69\times$ times more and have $1.9\times$–$24.7\times$ longer execution paths than those generated by SlowFuzz. We also empirically evaluated the efficacy of PERFFUZZ in generating pathological inputs in comparison with that of AFL, a conventional coverage-guided fuzzing tool. The inputs generated by PERFFUZZ exercise the most-frequently executed program branch $2\times$–$39\times$ times more often than the inputs generated by AFL. Finally, we performed

---

[1]`https://github.com/google/closure-compiler/issues/3173`

a manual analysis of the hot spots discovered by PERFFUZZ. We were able to identify how varying input features affect the performance of different program components.

Together, TRAVIOLI and PERFFUZZ have demonstrated that the information embedded in functional test cases can be sufficient to find algorithmic performance bottlenecks when combined with smart heuristics for dynamic program analysis and fuzz testing respectively.

Chapter 5

JQF and ZEST: Coverage-Guided Generator-Based Fuzzing

In the previous chapter, we saw the effectiveness of using a feedback-directed random fuzzing algorithm for automatically generating test inputs. However, CGF and CGF-like algorithms (e.g. PERFFUZZ) still rely on random mutations of inputs represented as sequences of bytes; therefore, they are limited in scope to testing programs that process binary data or parsers of simple text formats.

This chapter concerns automatic testing for complex input-processing pipelines, as shown in Figure 1.1 in Section 1.2. Off-the-shelf CGF tools such as AFL mostly find bugs only in the syntax parsing stages of such programs. Automatic generation of test inputs that have the required structure and semantics to exercise the main logic of the program is a challenging problem. We motivate this with a detailed example in Section 5.1.

We observe that the software developers who otherwise write manual test cases for such software have knowledge about the expected syntax and semantics of test programs. This is evidenced by the fact that in the absence of fuzzing tools, developers hand-craft functional test cases with hard-coded inputs having this structure. We therefore investigate whether we can leverage the knowledge of such domain experts to capture information about input structure and/or semantics in a manner that can be used for coverage-guided fuzzing.

A well-known abstraction for sampling complex inputs—such as XML documents and abstract syntax trees–is that of *generators*; these are simply functions whose job is to return a randomly constructed object of a user-defined type. Popularized by QuickCheck [37], this approach has been adopted by many generator-based testing tools [55, 43, 11, 68, 145, 87, 4, 103]. QuickCheck-like test frameworks are now available in many programming languages such as Java [88], PHP [58], Python [89], JavaScript [96], Scala [162], and Clojure [180]. Many commercial black-box fuzzing tools, such as Peach [146], beSTORM [21], Cyberflood [46], and Codenomicon [39], also leverage generators for network protocols or file formats. However, in order to effectively exercise the semantic analyses in the test program, the generators need to be tuned to produce inputs that are not only *syntactically* valid but also *semantically* valid. For example, the developers of CSmith [192], a tool that generates random C programs for

testing compilers, spent significant effort manually tuning their generator to reliably produce valid C programs and to maximize code coverage in the compilers under test.

In this chapter, we present ZEST [141] in Section 5.2, a technique for *automatically* guiding simple QuickCheck-like input generators to exercise various code paths in the semantic analysis stages of programs. ZEST incorporates feedback from the test program in the form of *semantic validity* of test inputs and the *code coverage* achieved during test execution. The feedback is then used to generate new inputs via mutations. ZEST adapts the algorithm used by coverage-guided fuzzing (CGF) tools in order to quickly explore the *semantic analysis* stages of test programs.

ZEST treats QuickCheck-like random-input generators as deterministic *parametric generators*, which map a sequence of untyped bits, called the "parameters", to a syntactically valid input. The key insight in ZEST is that *bit-level* mutations on these parameters correspond to *structural* mutations in the space of syntactically valid inputs. ZEST then applies a CGF algorithm on the domain of parameters, in order to guide the test-input generation towards semantic validity and increased code coverage in the semantic analysis stages.

Effectively, ZEST combines the user-provided domain knowledge that can be encoded into generator-based testing tools with the proven effectiveness of the feedback-driven approach popularized by CGF tools.

This chapter also describes the JQF framework [140] in Section 5.3. JQF was developed to implement parametric generators and the ZEST algorithm. JQF has been open-sourced and is available at the following URL: `https://github.com/rohanpadhye/JQF`. JQF builds on `junit-quickcheck` [88], which is a port of QuickCheck [37] to the popular JUnit testing framework for Java programs. JQF converts off-the-shelf `junit-quickcheck` generators to parametric generators. JQF also allows customizing the *guidance*; that is, the algorithm that determines which parameter sequence to use for each test execution in the fuzzing loop. ZEST is implemented in JQF along with several other guidances (see Section 5.3.4).

Finally, in Section 5.4, we describe an experimental evaluation of ZEST on five real-world Java benchmarks and compare it to AFL and `junit-quickcheck`. Our results show that the ZEST technique achieves significantly higher code coverage in the semantic analysis stage of each benchmark. Further, during our evaluation, we find 10 new bugs in the semantic analysis stages of our benchmarks. We find ZEST to be the most effective technique for reliably and quickly triggering these *semantic bugs*. For each benchmark, ZEST discovers an input triggering every semantic bug in at most 10 minutes on average. ZEST complements AFL, which is best suited for finding syntactic bugs.

5.1   Problem Motivation

5.1.1   Generator-Based Testing

Generator-based testing tools [37, 55, 43, 68, 145, 192, 87, 4] allow users to write generator programs for producing inputs that belong to a specific type or format. These random-input

generators are *non-deterministic*, i.e., they sample a new input each time they are executed. Figure 5.1 shows a generator for XML documents in the `junit-quickcheck` [88] framework, which is a Java port of QuickCheck [37]. When `generate()` is called, the generator uses the Java standard library XML DOM API to generate a random XML document. It constructs the root element of the document by invoking `genElement` (Line 4). Then, `genElement` uses repeated calls to methods of `random` to generate the element's tag name (Line 9), any embedded text (Lines 19, 20, and in `genString`), and the number of children (Line 13); it recursively calls `genElement` to generate each child node. We omitted code to generate attributes, but it can be done analogously.

Figure 5.2 contains a sample test harness method `testProgram`, identified by the `@Property` annotation. This method expects a test input `xml` of type `XMLDocument`; the `@From` annotation indicates that inputs will be randomly generated using the `XMLGenerator.generate()` API. When invoked with a generated XML document, `testProgram` serializes the document (Line 3) and invokes the `readModel` method (Line 9), which parses an input string into a domain-specific model. For example, Apache Maven parses `pom.xml` files into an internal Project Object Model (POM). The model creation fails if the input XML document string does not meet certain syntactic and semantic requirements (Lines 11, 13). If the model creation is successful, the check at Line 4 succeeds and the test harness invokes the method `runModel` at Line 5 to test one of the core functionalities of the program under test.

An XML generator like the one shown in Figure 5.1 generates random syntactically valid XML inputs; therefore Line 11 in Figure 5.2 will never be executed. However, the generated inputs may not be *semantically* valid. That is, the inputs generated by the depicted XML generator do not necessarily conform to the schema expected by the application. In our example, the `readModel` method could throw a `ModelException` and cause the assumption at Line 4 to fail. If this happens, QuickCheck simply discards the test case and tries again. Writing generators that produce semantically valid inputs by construction is a challenging manual effort.

When we tested Apache Maven's model reader for `pom.xml` files using a generator similar to Figure 5.1, we found that only 0.09% of the generated inputs were semantically valid. Moreover, even if the generator manages to generate semantically valid inputs, it may not generate inputs that exercise a variety of code paths in the semantic analysis stage. In our experiments with Maven, the QuickCheck approach covers less than one-third of the branches in the semantic analysis stage than our proposed technique does. Fundamentally, this is because of the lack of coupling between the generators and the program under test.

5.1.2 Coverage-Guided Fuzzing

Coverage-guided fuzzing (CGF) tools (ref. Section 2.5) instrument programs under test and utilize feedback from each test execution in the form of code coverage. However, a key limitation of existing CGF tools is that they work without any knowledge about the syntax of the input. State-of-the-art CGF tools [196, 111, 24, 155, 108, 34] treat program inputs as sequences of bytes. This choice of representation also influences the design of their mutation

```
1 class XMLGenerator implements Generator<XMLDocument> {
2     @Override // For Generator<XMLDocument>
3     public XMLDocument generate(Random random) {
4       XMLElement root = genElement(random, 1);
5       return new XMLDocument(root);
6     }
7     private XMLElement genElement(Random random, int depth) {
8       // Generate element with random name
9       String name = genString(random);
10       XMLElement node = new XMLElement(name);
11       if (depth < MAX_DEPTH) { // Ensures termination
12           // Randomly generate child nodes
13           int n = random.nextInt(MAX_CHILDREN);
14           for (int i = 0; i < n; i++) {
15             node.appendChild(genElement(random, depth+1));
16           }
17       }
18       // Maybe insert text inside element
19       if (random.nextBool()) {
20         node.addText(genString(random));
21       }
22       return node;
23     }
24     private String genString(Random random) {
25       // Randomly choose a length and characters
26       int len = random.nextInt(1, MAX_STRLEN);
27       String str = "";
28       for (int i = 0; i < len; i++) {
29         str += random.nextChar();
30       }
31       return str;
32     }
33 }
```

Figure 5.1: A simplified XML document generator.

```
1 @Property
2 void testProgram(@From(XMLGenerator.class) XMLDocument xml) {
3   Model model = readModel(xml.toString());
4   assume(model != null); // validity
5   assert(runModel(model) == success);
6 }
7 private Model readModel(String input) {
8     try {
9         return ModelReader.readModel(input);
10    } catch (XMLParseException e) {
11        return null; // syntax error
12    } catch (ModelException e) {
13        return null; // semantic error
14    }
15 }
```

Figure 5.2: A junit-quickcheck property that tests an XML-based component.

operations, which include bit-flips, arithmetic operations on word-sized segments, setting random bytes to random or "interesting" values (e.g. 0, `MAX_INT`), etc. These mutations are tailored towards exercising various code paths in programs that parse inputs with a compact syntax, such as parsers for media file formats, decompression routines, and network packet analyzers. CGF tools have been very successful in finding memory-corruption bugs (such as buffer overflows) in the syntax analysis stage of such programs due to incorrect handling of unexpected inputs.

Unfortunately, this approach often fails to exercise the core functions of software that expects highly structured inputs. For example, when AFL (ref. Section 2.5.1) is applied on a program that processes XML input data, a typical input that it saves looks like:

<div align="center"><a b>ac&#84;a></div>

which exercises code paths that deal with syntax errors. In this case, an error-handling routine for unmatched start and end XML tags. It is very difficult to generate inputs that will exercise new, interesting code paths in the semantic analysis stage of a program via these low-level mutations. Often, it is necessary to run CGF tools for hours or days on end in order to find non-trivial bugs, making them impractical for use in a continuous integration setting.

5.2   Semantic Fuzzing with ZEST

ZEST adds the power of coverage-guided fuzzing to generator-based testing. ZEST treats a random-input generator as an equivalent deterministic *parametric generator*. ZEST then searches through the parameter space using an algorithm we call *semantic fuzzing*. This technique augments the CGF algorithm by keeping track of code coverage achieved by valid inputs. This enables it to guide the search towards deeper code paths in the semantic analysis stage.

5.2.1   Parametric Generators

Before defining parametric generators, let us return to the random XML generator from Figure 5.1. Let us consider a particular path through this generator, concentrating on the calls to `nextInt`, `nextBool`, and `nextChar`. The following sequence of calls will be our running example (some calls ommitted for space):

Call → result	Context
`random.nextInt(1, MAX_STRLEN)` → 3	Root: `name` length (Line 26)
`random.nextChar()` → 'f'	Root: `name[0]` (Line 29)
`random.nextChar()` → 'o'	Root: `name[1]` (Line 29)
`random.nextChar()` → 'o'	Root: `name[2]` (Line 29)
`random.nextInt(MAX_CHILDREN)` → 2	Root: # children (Line 13)
`random.nextInt(1, MAX_STRLEN)` → 3	Child 1: `name` length (Line 26)

$$\vdots$$

`random.nextBool()` → False	Child 2: embed text? (Line 19)
`random.nextBool()` → False	Root: embed text? (Line 19)

The XML document produced when the generator makes this sequence of calls looks like:

$$x_1 = \texttt{<foo><bar>Hello</bar><baz /></foo>}.$$

In order to produce random typed values, the implementations of `random.nextInt`, `random.nextChar`, and `random.nextBool` rely on a pseudo-random source of *untyped* bits. We call these untyped bits "*parameters*". The parameter sequence for the example above, annotated with the calls which consume the parameters, is:

$$\sigma_1 = \underbrace{\texttt{0000 0010}}_{\texttt{nextInt(1,...)}\to 3} \quad \underbrace{\texttt{0110 0110}}_{\texttt{nextChar()}\to\text{'f'}} \quad \dots \quad \underbrace{\texttt{0000 0000}}_{\texttt{nextBool()}\to\text{False}} .$$

For example, here the function `random.nextInt(`a, b`)` consumes eight bit parameters as a byte, n, and returns $n \% (b - a) + a$ as a typed integer. For simplicity of presentation, we show each `random.nextXYZ` function consuming the same number of parameters, but they can consume different numbers of parameters.

We can now define a *parametric generator*. A parametric generator is a function that takes a sequence of untyped parameters such as σ_1—the *parameter sequence*—and produces a structured input, such as the XML x_1. A parametric generator can be implemented by simply replacing the underlying implementation of `Random` to consult not a pseudo-random source of bits but instead a fixed sequence of bits provided as the parameters.

While this is a very simple change, making generators deterministic and explicitly dependent on a fixed parameter sequence enables us to make the following two key observations:

1. *Every untyped parameter sequence corresponds to a syntactically valid input*—assuming the generator only produces syntactically valid inputs.

2. *Bit-level mutations on untyped parameter sequences correspond to high-level structural mutations in the space of syntactically valid inputs.*

Observation (1) is true by construction. The `random.nextXYZ` functions are implemented to produce correctly-typed values no matter what bits the pseudo-random source–or in our case, the parameters—provide. Every sequence of untyped parameter bits correspond to

some execution path through the generator, and therefore every parameter sequence maps to a syntactically valid input. We describe how we handle parameter sequences that are longer or shorter than expected with the example sequences σ_3 and σ_4, respectively, below.

To illustrate observation (2), consider the following parameter sequence, σ_2, produced by mutating just a few bits of σ_1:

$$\sigma_2 = 0000\ 0010\ \underbrace{01\underline{01}\ 011\underline{1}}_{\texttt{nextChar()}\rightarrow\text{`W'}}\ \ldots\ 0000\ 0000.$$

As indicated by the annotation, all this parameter-sequence mutation does is change the value returned by the second call to `random.nextChar()` in our running example from 'f' to 'W'. So the generator produces the following test-input:

$$x_2 = \ \texttt{<Woo><bar>Hello</bar><baz /></Woo>}.$$

Notice that this generated input is still syntactically valid, with "Woo" appearing both in the start and end tag delimiters. This is because the XML generator uses an internal DOM tree representation that is only serialized after the entire tree is generated. We get this syntactic-validity-preserving structural mutation for free, *by construction*, and without modifying the underlying generators.

Mutating the parameter sequence can also result in more drastic high-level mutations. Suppose that σ_1 is mutated to influence the first call to `random.nextInt(MAX_CHILDREN)` as follows:

$$\sigma_3 = 0000\ \ldots\ \underbrace{0000\ 00\underline{01}}_{\texttt{nextInt(MAX\_CHILDREN)}\rightarrow 1}\ \ldots\ 0000.$$

Then the root node in the generated input will have only one child:

$$x_3 = \ \texttt{<foo><bar>Hello</bar>}\blacksquare\texttt{</foo>}$$

($\blacksquare$ designates the absence of `<baz />`). Since the remaining values in the untyped parameter sequence are the same, the first child node in x_3—`<bar>Hello</bar>`—is identical to the one in x_1. The parametric generator thus enables a structured mutation in the DOM tree, such as deleting a sub-tree, by simply changing a few values in the parameter sequence. Note that this change results in fewer `random.nextXYZ` calls by the generator; the unused parameters in the tail of the parameter sequence will simply be ignored by the parametric generator.

For our final example, suppose σ_1 is mutated as follows:

$$\sigma_4 = 0000\ 0011\ \ldots\ \underbrace{0000\ 000\underline{1}}_{\texttt{nextBool()}\rightarrow\text{True}}\ \underbrace{0000\ 0000}_{\texttt{nextInt(1,...)}\rightarrow 1}\ .$$

Notice that after this mutation, the last 8 parameters are consumed by `nextInt` instead of by `nextBool` (ref. σ_1). But, note that `nextInt` still returns a valid typed value even though the parameters were originally consumed by `nextBool`.

At the input level, this modifies the call sequence so that the decision to embed text in the second child of the document becomes True. Then, the last parameters are used by `nextInt` to choose an embedded text length of 1 character. However, one problem remains: to generate the content of the embedded text, the generator needs more parameter values than σ_4 contains. In ZEST, we deal with this by appending pseudo-random values to the end of the parameter sequence on demand. We use a fixed random seed to maintain determinism. For example, suppose the sequence is extended as:

$$\sigma_4' = 0000 \ \ldots \ \ 000\underline{1} \ 0000 \ 0000 \ \underbrace{0100 \ 1100}_{\texttt{nextChar()}\to\text{`H'}} \ \underbrace{0000 \ 0000}_{\texttt{nextBool()}\to\text{False}}$$

Then the parametric generator would produce the test-input:

$$x_4 = \ \texttt{<foo><bar>Hello</bar><baz>\underline{H}</baz></foo>}.$$

5.2.2 The ZEST Algorithm for Semantic Fuzzing

Algorithm 3 shows the ZEST algorithm, which guides parametric generators to produce inputs that get deeper into the semantic analysis stage of programs. We call this technique *semantic fuzzing*. The ZEST algorithm resembles Algorithm 1, but acts on parameter sequences rather than the raw inputs to the program. It also extends the CGF algorithm by keeping track of the coverage achieved by *semantically valid inputs*. We highlight the differences between Algorithms 3 and 1 in grey.

Like Algorithm 1, ZEST is provided a program under test p. Unlike Algorithm 1 which assumes seed inputs, the set of parameter sequences is initialized with a random sequence (Line 1). Additionally, ZEST is provided a generator q, which is automatically converted to a parametric generator g (Line 5). In an abuse of notation, we use $g(S)$ to designate the set of inputs generated by running g over the parameter sequences in S, i.e. $g(S) = \{g(s) : s \in S\}$.

Along with *totalCoverage*, which maintains the set of coverage points in p covered by *all* inputs in $g(\mathcal{S})$, ZEST also maintains *validCoverage*, the set of coverage points covered by the (semantically) valid inputs in $g(\mathcal{S})$. This is initialized at Line 4.

New parameter sequences are generated using standard CGF mutations at Line 9. The program p is executed on inputs that are generated by running the sequences through the parametric generator (Line 10). During the execution, in addition to code-coverage and failure feedback, the algorithm records in the variable *result* whether the input is valid or not. In particular, *result* can be one of {VALID, INVALID, FAILURE}. An input is considered invalid if it leads to a violation of any assumption in the test harness (e.g. Figure 5.2 at Line 4), which is how we capture application-specific semantic validity.

As in Algorithm 1, a newly generated parameter sequence is added to the set $\mathcal{S}$ at Lines 14–16 of Algorithm 3 if the corresponding input produces new code coverage. Further, if the corresponding input is *valid* and covers a coverage point that has not been exercised by *any previous valid input*, then the parameter sequence is added $\mathcal{S}$ and the cumulative valid coverage variable *validCoverage* is updated at Lines 18–20. Adding the parameter sequence

---

Algorithm 3 The ZEST algorithm for semantic fuzzing. Changes to Algorithm 1 highlighted in grey.

Input: an instrumented test program p, a user-provided generator q
Output: a set of test inputs and failing inputs
 1: $\mathcal{S} \leftarrow$ {RANDOM }
 2: $\mathcal{F} \leftarrow \emptyset$
 3: $totalCoverage \leftarrow \emptyset$
 4: $validCoverage \leftarrow \emptyset$
 5: $g \leftarrow$ MAKEPARAMETRIC(q)
 6: **repeat**
 7:    **for** i in $\mathcal{S}$ **do**
 8:       **if** sample FUZZPROB(i) **then**
 9:          $i' \leftarrow$ MUTATE$(i, \mathcal{S})$
10:          $coverage, result \leftarrow$ RUN$(p,$ $g(i')$ $)$
11:          **if** $result =$ FAILURE **then**
12:            $\mathcal{F} \leftarrow \mathcal{F} \cup \{i'\}$
13:          **else**
14:            **if** $coverage \cap totalCoverage \neq \emptyset$ **then**
15:               $\mathcal{S} \leftarrow \mathcal{S} \cup \{i'\}$
16:               $totalCoverage \leftarrow totalCoverage \cup coverage$
17:            **end if**
18:            **if** $result =$ VALID and $coverage \cap validCoverage \neq \emptyset$ **then**
19:               $\mathcal{S} \leftarrow \mathcal{S} \cup \{i'\}$
20:               $validCoverage \leftarrow validCoverage \cup coverage$
21:            **end if**
22:          **end if**
23:       **end if**
24:    **end for**
25: **until** given time budget expires
26: **return** $g(\mathcal{S}),$ $g(\mathcal{F})$

---

to $\mathcal{S}$ under this new condition ensures that ZEST keeps mutating valid inputs that exercise core program functionality. We hypothesize that this biases the search towards generating even more valid inputs and in turn increases code coverage in the semantic analysis stage.

As in Algorithm 1, the testing loop repeats until a time budget expires. Finally, ZEST returns the corpus of generated test inputs $g(\mathcal{S})$ and failing inputs $g(\mathcal{F})$.

```
1  @RunWith(JQF.class)
2  class TrieTest {
3    @Fuzz /* Arguments are generated randomly by JQF */
4    public void testMap2Trie(String key, Map<String,Integer> map){
5      assumeTrue(map.containsKey(key));
6      Trie trie = new PatriciaTrie(map); // Map2Trie
7      assertTrue(trie.containsKey(key));
8    }
9  }
```

Figure 5.3: A sample property test using JQF that checks the construction of a `Trie` data structure in Apache Commons from an input JDK `Map`. Fires an assertion violation (bug COLLECTIONS-714) when fuzzing with `ZestGuidance`.

5.3   The JQF Framework

In order to implement ZEST, we need a way to take existing QuickCheck-like generators and control the parameter sequences that drive the pseudo-random choices they make.

To this end, we developed a framework called JQF on top of `junit-quickcheck` [88], which itself is a Java port of the popular QuickCheck [37] tool. JQF enables the controlled guidance of `junit-quickcheck` generators using feedback from test execution in the form of code coverage. JQF has been made available at: `https://github.com/rohanpadhye/JQF`. ZEST is implemented as a special case of a guidance that implements Algorithm 3.

Practitioners can use JQF to automatically generate test inputs for parameterized test methods using coverage-guided fuzzing. Figure 5.3 shows an example of a JQF test driver written in Java, which aims to check a basic property of the class `PatriciaTrie` from Apache Commons Collections. A trie data structure can be constructed from a pre-existing mapping of strings in a JDK `Map` object.

The test method `testMap2Trie` checks the following property: Given an arbitrary string `key` and a JDK `map` whose keys are strings, if `key` exists within `map`, then a trie constructed from this map should also contain the same key. The `@Fuzz` annotation on the test method enables JQF to *automatically generate random instances* of `map` and `key` to verify this property. The JUnit `Assume` API allows the user to specify preconditions on the generated inputs (e.g. Line 5). Test generation can be launched via JQF's Apache Maven plugin[1]:

<div align="center">

`mvn jqf:fuzz -Dclass=TrieTest -Dmethod=testMap2Trie`

</div>

By default, JQF uses the ZEST algorithm to generate test inputs. Fuzzing continues either until it is explicitly stopped, until a user-specified timeout expires, or until a test failure is encountered.

For the test in Figure 5.3, the ZEST fuzzing engine often finds a test failure in about 5 seconds, after executing about 5,000 test inputs (of which over 1,700 satisfy the precondition on Line 5). The failing test case leads to an assertion violation at Line 7 due to a very special

---

[1]Non-Maven users can launch JQF programatically or via command-line scripts.

```
1 public interface Guidance {
2     boolean hasInput();
3     InputStream getInput();
4     void handleResult(Result result, Throwable error);
5     Consumer<TraceEvent> generateCallBack(Thread thread);
6 }
```

Figure 5.4: The `Guidance` interface provided by JQF.

corner case, which reveals a bug[2] in Apache Commons Collections v4.3. If the input `map` contains two distinct keys that differ only in a trailing null character, say `"x"` and `"x\u0000"`, then the `trie` cannot distinguish between them and ends up storing only one of the two keys. If the input `key` is also `"x"`, then the bug is revealed.

JQF was specifically designed to enable practitioners to write test methods in the familiar style of property-based testing. Thus, the test driver in Figure 5.3 can still be run with vanilla `junit-quickcheck`, which randomly generates test inputs without using code coverage feedback. However, random-from-scratch input generation is exceedingly unlikely to generate inputs fitting precise bug-revealing conditions, like those described above. Pure random generation does not find a failing test case for Figure 5.3 even after 30 minutes (over 7 million executions).

We next explain how JQF generates random inputs, such as `map` and `key` in Figure 5.3, using coverage-guided algorithms called *guidances.*

5.3.1 The `Guidance` Interface

Figure 5.4 shows the `Guidance` interface. Researchers can implement this interface to specify a coverage-guided fuzzing algorithm. `Guidance` instances are stateful objects whose methods are invoked by the JQF framework in a fuzzing loop (depicted in Figure 5.5).

The `Guidance` method `hasInput()` returns whether a new input is available; the return value `false` ends fuzzing. The `getInput()` method returns the next input generated by the `Guidance`, as an `InputStream`. This stream is used to generate structured inputs such as `Map` objects (see Section 5.3.2). The structured inputs, called `args` in Figure 5.5, are then used to execute the test method via JUnit (Line 7). Test execution generates `TraceEvent`s, whose handling is described in Section 5.3.3. At the end of test execution, the `Guidance.handleResult()` method is invoked. The result can either be `SUCCESS`, `INVALID`, or `FAILURE`, depending on whether the test method returned normally (Line 8), due to a violation of `assume` (Line 10), or due to an exception/assertion violation (Line 12), respectively. The `Guidance` instance updates its internal state based on the handling of code coverage events and the test result. The internal state is then used to generate new inputs in subsequent iterations of the fuzzing loop.

---

[2]`https://issues.apache.org/jira/browse/COLLECTIONS-714`

```
1  TestMethod test   = ...; // @Fuzz test driver
2  Guidance guidance = ...; // Fuzzing algorithm
3  while (guidance.hasInput()) {
4      // Generate args for test method
5      Object[] args = JQF.gen(test, guidance.getInput());
6      try {
7          JUnit.run(test, args); // fires TraceEvent(s)
8          guidance.handleResult(SUCCESS, null);
9      } catch (AssumptionViolatedException e) {
10         guidance.handleResult(INVALID, e);
11     } catch (Throwable t) {
12         guidance.handleResult(FAILURE, e);
13     }
14 }
```

Figure 5.5: Pseudo-code of JQF's fuzzing loop.

5.3.2 Parametric Generators

The arguments to a test method—such as `map` and `key` in Figure 5.3—are generated using the same mechanisms as supported by `junit-quickcheck`. In general, inputs of type `T` are generated by a backing `Generator<T>`, which provides a method to *randomly sample a new instance of T*. `junit-quickcheck` can either (1) implicitly pick a suitable generator from a library that it provides, (2) be directed to synthesize such a generator automatically, e.g. using the constructors or public fields of class `T`, or (3) be provided with a hand-written `Generator<T>`.

In all cases, the generator uses a `SourceOfRandomness` object, which provides an API for making non-deterministic decisions such as: choosing from a list of alternatives (e.g. whether to instantiate a `TreeMap` or `HashMap` for `map` in Figure 5.3), picking random sizes (e.g. how many entries to insert in `map`), or populating primitives (e.g. what keys and values to insert in `map`). In `junit-quickcheck`, the default `SourceOfRandomness` is backed a pseudo-random stream of bytes. JQF overrides this source to use the stream returned by `Guidance.getInput()` instead (ref. Line 5 in Figure 5.5), thereby making the generators deterministically dependent on the guidance.

5.3.3 Code Coverage Events

When coverage-guided fuzzing is launched (e.g. via `mvn jqf:fuzz`), the test program's classes are instrumented on-the-fly using the ASM bytecode manipulation library [138]. The instrumentation adds logic to generate `TraceEvent`s during test execution. For example, a `BranchEvent` is generated when a test program executes a conditional branch, a `CallEvent` accompanies a method invocation, and an `AllocEvent` signals the creation of a new object or array on the heap. These event objects contain information about their source program locations as well other event-specific data. When a trace event `e` is generated in thread `t`, JQF invokes the function `handle_t(e)`, where `handle_t` is the callback returned by

`Guidance.generateCallBack(t)`. The guidance must choose how to update its internal state based on this coverage information, which will presumably be used to generate subsequent inputs.

5.3.4 Guidances

JQF currently ships with the following `Guidance` implementations.

5.3.4.1 No Guidance

The most trivial guidance, called `NoGuidance`, returns an infinite stream of random values every time `getInput()` is called. This guidance completely ignores code coverage events. This guidance is almost equivalent to using vanilla `junit-quickcheck`.

5.3.4.2 ZEST Guidance

JQF's default guidance implements the ZEST algorithm (ref. Section 5.2.2), which is specifically designed for coverage-guided property testing. The `ZestGuidance` returns dynamically sized *parameter sequences* via the `getInput()` method, which are generated randomly for the first iteration of the fuzzing loop. Dynamic sizing allows the parameter sequences to be lazily extended (if the `Generator` needs to make more choices than expected) or to be efficiently truncated (if the `Generator` makes fewer choices).

ZEST maintains a set of saved parameter sequences. The `ZestGuidance` generates new inputs by randomly mutating previously saved parameter sequences. Byte-level mutations on these parameter sequences correspond to structural mutations in the generated test inputs. For example, a random mutation in the parameter sequence for `map` in Figure 5.3 may lead to the corresponding `Generator<Map>` to produce the next map with an additional entry. Since `java.util.Random` polls byte-sized chunks from its underlying stream of pseudo-random bits, ZEST performs mutations on the parameter sequences (Algorithm 3, Line 9) at the *byte-level*. The basic mutation procedure is as follows: (1) choose a random number m of mutations to perform sequentially on the original sequence, (2) for each mutation, choose a random length ℓ of bytes to mutate and an offset k at which to perform the mutation, and (3) replace the bytes from positions $[k, k + \ell)$ with ℓ randomly chosen bytes. The random numbers m and ℓ are chosen from a geometric distribution, which mostly provides small values without imposing an upper bound. We set the mean of this distribution to 4, since 4-byte `int`s are the most commonly requested random value.

Further, ZEST separately tracks code coverage achieved by *all* test executions and code coverage by *valid* test executions (i.e., those whose result is `SUCCESS`). If a mutated parameter sequence leads to new code coverage overall, or if it leads to a valid test that covers code which has not been covered by any previous *valid* test, then the sequence is saved for subsequent mutation. See Algorithm 3 for details.

5.3.4.3 AFL Guidance

JQF supports input generation using the popular AFL [196] tool, unmodified. This is possible because AFL, which is designed to fuzz C/C++ programs and x86 binaries, communicates with instrumented test programs via inter-process messages and a code coverage map in shared memory. The `AFLGuidance` in JQF implements this communication protocol via a proxy program. The proxy mocks an AFL-instrumented test target that reads input from a specific file. `AFLGuidance.getInput()` simply returns the contents of this file, which is continuously updated by AFL. During test execution, `AFLGuidance` collects code coverage information by handling `TraveEvent`s. When `AFLGuidance.handleResult()` is invoked, the coverage information is written to AFL's shared memory region via the proxy. Calls to `AFLGuidance.hasInput()` block until AFL is ready with the next input.

AFL's mutation strategy uses various heuristics that are applicable to programs that parse fixed-size binary files (e.g. media players). Further, AFL does not explicitly distinguish between `INVALID` and `FAILURE` results. Due to these reasons, JQF's `AFLGuidance` is most effective when used with test methods that take only one argument of type `InputStream` (since `Generator<InputStream>` returns the guidance-generated input stream as-is), and that do not use any `assume` statements. For example, `AFLGuidance` has been used to fuzz OpenJDK's ImageIO library that reads PNG and JPEG files[3], as well as Apache PDFBox's processing of PDF documents[4].

5.3.4.4 PERFFUZZ Guidance

PERFFUZZ (ref. Chapter 4) is a fork of AFL that extends its code coverage map with performance feedback in the form of $\langle k, v \rangle$ pairs where v is a value to be maximized for every key k. PERFFUZZ saves a mutated input either if it leads to new code coverage, or if it maximizes the value of v for some key k.

JQF's `PerfFuzzGuidance` is a sub-class of `AFLGuidance` which overrides `handleResult()` to communicate this additional performance map via the proxy program. `PerfFuzzGuidance` can be configured either to find hot spots (where keys are branch locations and values are execution counts for the corresponding branch) or to find memory consumption issues (where keys are allocation sites and values are number of bytes allocated at the corresponding site). For example, we used `PerfFuzzGuidance` to find an algorithmic complexity bug in the Google Closure Compiler, where reporting a specific case of syntax error in a JavaScript program can take time that is exponential in the size of the input program[5]. With the memory allocation feedback, we found an issue in OpenJDK's handling of PNG images that specify very large dimensions[6].

---

[3]`https://bugs.openjdk.java.net/browse/JDK-8191073`
[4]`https://issues.apache.org/jira/browse/PDFBOX-4333`
[5]`https://github.com/google/closure-compiler/issues/3173`
[6]`https://bugs.openjdk.java.net/browse/JDK-8190332`

Table 5.1: Number of new bugs discovered using JQF.

Project	Bugs Found	Bugs Fixed
OpenJDK - ImageIO	9	9
OpenJDK - DateTime	2	1
Apache Commons - Lang	1	1
Apache Commons - Compress	2	2
Apache Commons - Collections	1	0
Apache Maven	3	3
Apache Ant	1	1
Apache BCEL	8	0
Apache PDFBox	4	4
Apache Tika	2	2
Google Closure Compiler	4	1
Mozilla Rhino	5	0
Total	42	24

5.3.4.5 Repro Guidance

Finally, the `ReproGuidance` is a trivial guidance whose `getInput()` method returns the contents of a given file on disk, and then ends the loop. This guidance enables debugging of saved test failures.

5.3.5 New Software Bugs Uncovered

Table 5.1 summarizes the impact that JQF has had in discovering previously unknown bugs in widely used Java software. These bugs were found over the course of various experiments performed throughout 2017–2019.

Of the total 42 bugs found using JQF, 11 semantic bugs were found using `ZestGuidance` with appropriate generators, 29 syntax parsing bugs were found using `AFLGuidance` without using generators, and 2 bugs were found using `PerfFuzzGuidance`. 24 of the 42 reported bugs have been fixed at the time of writing, while the rest await patches.

Notably, 7 of the 42 bugs (including 4 security vulnerabilities with assigned CVEs) were discovered by two independent practitioners not affiliated with this author. We were made aware of JQF's success via social media [3] and blog posts [136]. All 7 of these bugs have been fixed. We are encouraged by these findings, and believe that they provide evidence to support JQF's usefulness to the software testing community at large.

5.4 Evaluation of ZEST

We evaluate ZEST by measuring its effectiveness in testing the semantic analysis stages of five benchmark programs. We compare the implementation of `ZestGuidance` in JQF with two baseline techniques: AFL, via `AFLGuidance` in JQF, and vanilla `junit-quickcheck`, via `NoGuidance` in JQF (referred to as simply QuickCheck hereon). AFL is known to excel in exercising the syntax analysis stage via coverage-guided fuzzing of raw input strings. We use version 2.52b, with "FidgetyAFL" configuration, which was found to match the performance of AFLFast [198]. QuickCheck uses the same generators as ZEST but only performs random sampling without any feedback from the programs under test. Specifically, we evaluate the three techniques on two fronts: (1) the amount of code coverage achieved in the semantic analysis stage after a fixed amount of time, and (2) their effectiveness in triggering bugs in the semantic analysis stage.

Benchmarks We use the following five real-world Java benchmarks as test programs for our evaluation:

1. Apache Maven [9] (99k LoC): The test reads a `pom.xml` file and converts it into an internal `Model` structure. The test driver is similar to the one in Figure 5.2. An input is valid if it is a valid XML document conforming to the POM schema.

2. Apache Ant [7] (223k LoC): Similar to Maven, this test reads a `build.xml` file and populates a `Project` object. An input is considered valid if it is a valid XML document and if it conforms to the schema expected by Ant.

3. Google Closure [78] (247k LoC) statically optimizes JavaScript code. The test driver invokes the `Compiler.compile()` on the input with the `SIMPLE_OPTIMIZATIONS` flag, which enables constant folding, function inlining, dead-code removal, etc.. An input is valid if Closure returns without error.

4. Mozilla Rhino [125] (89k LoC) compiles JavaScript to Java bytecode. The test driver invokes `Context.compileString()`. An input is valid if Rhino returns a compiled script.

5. Apache's Bytecode Engineering Library (BCEL) [8] (61k LoC) provides an API to parse, verify and manipulate Java bytecode. Our test driver takes as input a `.class` file and uses the `Verifier` API to perform 3-pass verification of the class file according to the Java 8 specification [110]. An input is valid if BCEL finds no errors up to Pass 3A verification.

Experimental Setup We make the following design decisions:

Table 5.2: Description of benchmarks with prefixes of class/package names corresponding to syntactic and semantic analyses.

Name	Version	Syntax Analysis Classes	Semantic Analysis Classes
Maven	3.5.2	`org/codehaus/plexus/util/xml`	`org/apache/maven/model`
Ant	1.10.2	`com/sun/org/apache/xerces`	`org/apache/tools/ant`
Closure	v20180204	`com/google/javascript/jscomp/parsing`	`com/google/javascript/jscomp/[A-Z]`
Rhino	1.7.8	`org/mozilla/javascript/Parser`	`org/mozilla/javascript/(optimizer|CodeGenerator)`
BCEL	6.2	`org/apache/bcel/classfile`	`org/apache/bcel/verifier`

- **Duration**: We run each test-generation experiment for *3 hours*. Researchers have used various timeouts to evaluate random test generation tools, from 2 minutes [139, 65] to 24 hours [24, 100].

  Our experiments justify this choice, as we found that semantic coverage plateaued after 2 hours in almost all experiments. Specifically, the number of semantic branches covered by ZEST increased by less than 1% in the last hour of the runs.

- **Repetitions**: Due to the non-deterministic nature of random testing, the results may vary across multiple repetitions of each experiment. We therefore run each experiment 20 times and report statistics across the 20 repetitions.

- **Seeds and Dictionaries**: To bootstrap AFL, we need to provide some initial seed inputs. There is no single best strategy for selecting initial seeds [157].

  Researchers have found success using varying strategies ranging from large seed corpora to single empty files [100]. In our evaluation, we provide AFL one valid seed input per benchmark that covers various domain-specific semantic features. For example, in the Closure and Rhino benchmarks, we use the entire React.JS library [156] as a seed.

  We also provide AFL with *dictionaries* of benchmark-specific strings (e.g. keywords, tag names) to inject into inputs during mutation. The generator-based tools ZEST and QuickCheck do not rely on meaningful seeds.

- **Generators**: ZEST and QuickCheck use hand-written input generators. For Maven and Ant, we use an XML document generator similar to Figure 5.1, of around 150 lines of Java code. It generates strings for tags and attributes by randomly choosing strings from a list of string literals scraped from class files in Maven and Ant. For Closure and Rhino, we use a generator for a subset of JavaScript that contains about 300 lines of Java code. The generator produces strings that are syntactically valid JavaScript programs. Finally, the BCEL generator (~500 LoC) uses the BCEL API to generate `JavaClass` objects with randomly generated fields, attributes and methods with randomly generated bytecode instructions. All generators were written in less than two hours each. Although these generators produce syntactically valid inputs, no

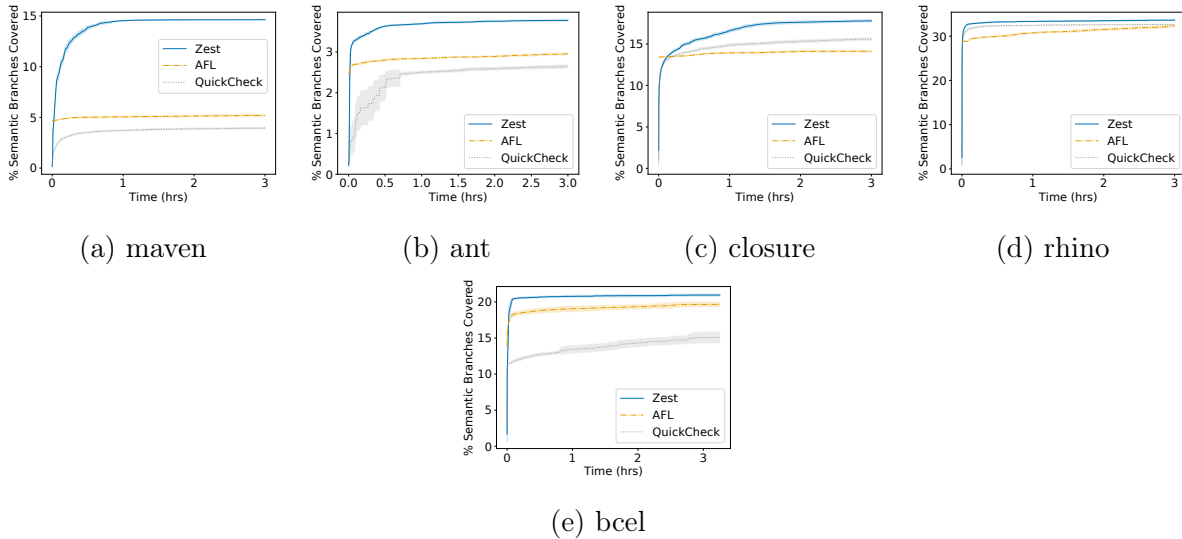(a) maven   (b) ant   (c) closure   (d) rhino

(e) bcel

Figure 5.6: Percent coverage of *all* branches in semantic analysis stage of the benchmark programs. Lines designate means and shaded regions 95% confidence intervals.

effort was made to produce semantically valid inputs; doing so can be a complex and tedious task [192].

All experiments are conducted on a machine with Intel(R) Core(TM) i7-5930K 3.50GHz CPU and 16GB of RAM running Ubuntu 18.04. A replication package that includes all the generators, seeds, and dictionaries has been made publicly available at `https://doi.org/10.1145/3339069`.

Syntax and Semantic Analysis Stages in Benchmarks ZEST is specifically designed to exercise the semantic analysis stages of programs. To evaluate ZEST's effectiveness in this regard, we manually identify the components of our benchmark programs which correspond to syntax and semantic analysis stages. Table 5.2 lists prefix patterns that we match on the fully-qualified names of classes in our benchmarks to classify them in either stage. Section 5.4.1 evaluates the code coverage achieved within the classes identified as belonging to the semantic analysis stage. Section 5.4.2 evaluates the bug-finding capabilities of each technique for bugs that arise in the semantic analysis classes. Section 5.5 discusses some findings in the *syntax* analysis classes, whose testing is outside the scope of ZEST.

5.4.1 Coverage of Semantic Analysis Classes

Instead of relying on our own instrumentation, we use a third party tool, the widely used Eclemma-JaCoCo [86] library, for measuring code coverage in our Java benchmarks. Specif-

ically, we measure *branch coverage* within the semantic analysis classes from Table 5.2; we refer to these branches as *semantic branches* for short.

To approximate the coverage of the semantic branches covered via the selected test drivers, we report the percentage of total semantic branches covered. Note, however, that this is a *conservative*, i.e. low, estimate. This is because the total number of semantic branches includes some branches not reachable from the test driver. We make this approximation as it is not feasible to statically determine the number of branches reachable from a given entry point, especially in the presence of virtual method dispatch. We expect the percent of semantic branches reachable from our test drivers to be much lower than 100%; therefore, the relative differences between coverage are more important than the absolute percentages.

Figure 5.6 plots the semantic branch coverage achieved by each of ZEST, AFL, and QuickCheck on the five benchmark programs across the 3-hour-long runs. In the plots, solid lines designate means and shaded areas designate 95% confidence intervals across the 20 repetitions. Interestingly, the variance in coverage is quite low for all techniques except QuickCheck. Since AFL is initialized with valid seed inputs, its initial coverage is non-zero; nonetheless, it is quickly overtaken by ZEST, usually within the first 5 minutes.
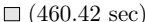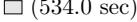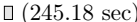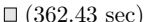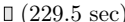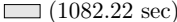
ZEST significantly outperforms baseline techniques in exercising branches within the semantic analysis stage, achieving statistically significant increases for all benchmarks. ZEST covers as much as $2.81\times$ as many semantic branches covered by the best baseline technique for Maven (Figure 5.6a). When looking at our Javascript benchmarks, we see that ZEST's advantage over QuickCheck is more slight in Rhino (Figure 5.6b) than in Closure (Figure 5.6c). This may be because Closure, which performs a variety of static code optimizations on JavaScript programs, has many input-dependent paths. Rhino, on the other hand, directly compiles valid JavaScript to JVM bytecode, and thus has fewer input-dependent paths for ZEST to discover via semantic fuzzing.

Note that in some benchmarks AFL has an edge in coverage over QuickCheck (Figure 5.6a, 5.6b, 5.6e), and vice-versa (Figure 5.6c, 5.6d). For BCEL, this may be because the input format is a compact syntax, on which AFL generally excels. The difference between the XML and JavaScript benchmarks may be related to the ability of randomly-sampled inputs from the generator to achieve broad coverage. It is much more likely for a random syntactically valid JavaScript program to be semantically valid than a random syntactically valid XML document to be a valid POM file, for example. The fact that ZEST dominates the baseline approaches in all these cases suggests that it is more robust to generator quality than QuickCheck.

5.4.2 Bugs in the Semantic Analysis Classes

Each of ZEST, AFL, and QuickCheck keep track of generated inputs which cause test failures. Ideally, for any given input, the test program should either process it successfully or reject the input as invalid using a documented mechanism, such as throwing a checked `ParseException` on syntax errors. Test *failures* correspond either to assertion violations or to undocumented

Table 5.3: The 10 new bugs found in the semantic analysis stages of benchmark programs. The tools ZEST, AFL, and QuickCheck (QC) are evaluated on the *mean time to find* (MTF) each bug across the 20 repeated experiments of 3 hours each as well as the *reliability* of this discovery, which is the percentage of the 20 repetitions in which the bug was triggered at least once. For each bug, the highlighted tool is statistically significantly more effective at finding the bug than unhighlighted tools.

Bug ID	Exception	Tool	Mean Time to Find (shorter is better)	Reliability
ant Ⓑ	IllegalStateException	(ZEST)	(99.45 sec)	100%
		AFL	(6369.5 sec)	10%
		QC	(1208.0 sec)	10%
closure Ⓒ	NullPointerException	(ZEST)	(8.8 sec)	100%
		AFL	(5496.25 sec)	20%
		(QC)	(8.8 sec)	100%
closure Ⓓ	RuntimeException	(ZEST)	(460.42 sec)	60%
		AFL	✗	0%
		QC	✗	0%
closure Ⓤ	IllegalStateException	(ZEST)	(534.0 sec)	5%
		AFL	✗	0%
		QC	✗	0%
rhino Ⓖ	IllegalStateException	(ZEST)	(8.25 sec)	100%
		AFL	(5343.0 sec)	20%
		(QC)	(9.65 sec)	100%
rhino Ⓕ	NullPointerException	ZEST	(18.6 sec)	100%
		AFL	✗	0%
		(QC)	(9.85 sec)	100%
rhino Ⓗ	ClassCastException	(ZEST)	(245.18 sec)	85%
		AFL	✗	0%
		QC	(362.43 sec)	35%
rhino Ⓙ	VerifyError	(ZEST)	(94.75 sec)	100%
		AFL	✗	0%
		QC	(229.5 sec)	80%
bcel Ⓞ	ClassFormatException	ZEST	(19.5 sec)	100%
		(AFL)	(5.85 sec)	100%
		QC	(142.1 sec)	100%
bcel Ⓝ	AssertionViolatedException	(ZEST)	(19.32 sec)	95%
		AFL	(1082.22 sec)	90%
		QC	(15.0 sec)	5%

run-time exceptions being thrown during test execution, such as a NullPointerException. Test failures can occur during the processing of either valid or invalid inputs; the latter can lead to failures within the syntax or semantic analysis stages themselves.

Across all our experiments, the various fuzzing techniques generated over 95,000 failing inputs that correspond to over 3,000 unique stack traces. We manually triaged these failures

by filtering them based on exception type, message text, and source location, resulting in a corpus of what we believe are 20 unique bugs. We have reported each of these bugs to the project developers. At the time of writing: 5 bugs have been fixed, 10 await patches, and 5 reports have received no response. See Section 5.3.5 for a summary of all new bugs found using the JQF framework.

We classify each bug as *syntactic* or *semantic*, depending on whether the corresponding exception was raised within the syntactic or semantic analysis classes, respectively (ref. Table 5.2). Of the 20 unique bugs we found, 10 were syntactic and 10 were semantic.

Here, we evaluate ZEST in discovering *semantic bugs*, for which it is specifically designed. Section 5.5 discusses the syntactic bugs we found, whose discovery was not ZEST's goal.

Table 5.3 enumerates the 10 semantic bugs that we found across four of the five benchmark programs. The bugs have been given unique IDs—represented as circled letters—for ease of discussion. The table also lists the type of exception thrown for each bug. To evaluate the effectiveness of each of the three techniques in discovering these bugs, we use two metrics. First, we are interested in knowing whether a given technique reliably finds the bug across repeated experiments. We define *reliability* as the percentage of the 20 runs (of 3-hours each) in which a given technique finds a particular bug at least once. Second, we measure the *mean time to find* (MTF) the first input that triggers the given bug, across the repetitions in which it was found. Naturally, a shorter MTF is desirable. For each bug, we circle the name of the technique that is the most effective in finding that bug. We define *most effective* as the technique with either the highest reliability, or if there is a tie in reliability, then the shortest MTF.

The table indicates that ZEST is the most effective technique in finding 8 of the 10 bugs; in the remaining two cases (Ⓕ and Ⓞ), ZEST still finds the bugs with 100% reliability and in less than 20 seconds on average. In fact, ZEST finds all the 10 semantic bugs in *at most 10 minutes on average*; 7 are found within the first 2 minutes on average. In contrast, AFL requires more than one hour to find 3 of the bugs (Ⓑ, Ⓒ, Ⓖ), and simply does not find 5 of the bugs within the 3-hour time limit. This makes sense because AFL's mutations on the raw input strings do not guarantee syntactic validity; it generates much fewer inputs that reach the semantic analysis stage. QuickCheck discovers 8 of the 10 semantic bugs, but since it relies on random sampling alone, its reliability is often low. For example, QuickCheck discovers Ⓑ only 10% of the time, and Ⓝ only 5% of the time; ZEST discovers them 100% and 95% of the time, respectively. Overall, ZEST is clearly the most effective technique in discovering bugs in the semantic analysis classes of our benchmarks.

Case studies   In Ant, Ⓑ is triggered when the input `build.xml` document contains both an `<augment>` element and a `<target>` element inside the root `<project>` element, but when the `<augment>` element is missing an `id` attribute. This incomplete semantic check leads to an `IllegalStateException` for a component down the pipeline which tries to configure an Ant task. Following our bug report, this issue has been fixed starting Ant version 1.10.6.

In Rhino, Ⓙ is triggered by a semantically valid input. Rhino successfully validates the

input JavaScript program and then compiles it to Java bytecode. However, the compiled bytecode is corrupted, which results in a `VerifyError` being generated by the JVM. AFL does not find this bug at all. The Rhino developers confirmed the bug, though a fix is still pending.

In Closure, Ⓒ is an NPE that is triggered in its dead-code elimination pass when handling arrow functions that reference undeclared variables, such as `"x => y"`. The generator-based techniques always find this bug and within just 8.8 seconds on average, while AFL requires more than 90 minutes and only finds it in 20% of the runs. The Closure developers fixed this issue after our report.

Ⓓ is a bug in Closure's semantic analysis of variable declarations. The bug is triggered when a new variable is declared after a `break` statement. Although everything immediately after a `break` statement is unreachable code, variable declarations in JavaScript are hoisted and therefore cannot be removed. ZEST is the only technique that discovered this bug. A sample input ZEST generated is:

```
while ((l_0)){
  while ((l_0)){
    if ((l_0)) { break;;var l_0;continue }
    { break;var l_0 }
  }
}
```

Ⓤ was the most elusive bug that we encountered. ZEST is the only technique that finds it and it does so in only one of the 20 runs. An exception is triggered by the following input:

```
((o_0) => (((o_0) *= (o_0))
   < ((i_1) &= ((o_0)(((((undefined)[(((i_1, o_0, a_2) => {
      if ((i_1)) { throw ((false).o_0) }
  })((y_3)))])((new (null)((true)))))))))))
```

The developers acknowledged this bug but have not yet published a fix. These complex examples demonstrate both the power of ZEST's generators, which reduce the search space to syntactically valid inputs, as well as the effectiveness of the semantic fuzzing technique.

5.5   Discussion and Limitations

ZEST and QuickCheck make use of generators for synthesizing inputs that are syntactically valid by construction. By design, these tools do not exercise code paths corresponding to parse errors in the syntax analysis stage. In contrast, AFL performs mutations directly on raw input strings. Byte-level mutations on raw inputs usually lead to inputs that do not parse. Consequently, AFL spends most of its time testing error paths within the syntax analysis stages.

In our experiments, AFL achieved the highest coverage within the syntax analysis classes of our benchmarks (ref. Table 5.2), 1.1×-1.6× higher than ZEST's syntax analysis coverage. Further, AFL discovered 10 syntactic bugs in addition to the bugs enumerated in Table 5.2:

3 in Maven, 6 in BCEL, and 1 in Rhino. These bugs were triggered by syntactically invalid inputs, which the generator-based tools do not produce. ZEST does not attempt to target these bugs; rather, it is complementary to AFL-like tools.

ZEST assumes the availability of QuickCheck-like generators to exercise the semantic analysis classes and to find semantic bugs. Although this is no doubt an additional cost, the effort required to develop a structured-input generator is usually no more than the effort required to write unit tests with hand-crafted structured inputs, which is usually an accepted cost. In fact, due to the growing popularity of generator-based testing tools like Hypothesis [89], ScalaCheck [162], PropEr [145], etc. a large number of off-the-shelf or automatically synthesized type-based generators are available. The ZEST technique can, in principle, work with any such generator. When given a generator, ZEST excels at exercising semantic analyses and is very effective in discovering semantic bugs.

We did not evaluate how ZEST's effectiveness might vary depending on the quality of generators, since we hand-wrote the simplest generators possible for our benchmarks. However, our results suggest that ZEST's ability to guide generation towards paths deep in the semantic analysis stage make its performance less tied to generator quality than pure random sampling as in QuickCheck.

The effectiveness of CGF tools like AFL is usually sensitive to the choice of seed inputs [100]. Although the relative differences between the performance of ZEST and AFL will likely vary with this choice, the purpose of our evaluation was to demonstrate that focusing on feedback-directed search in the space of syntactically valid inputs is advantageous. No matter what seed inputs one provides to conventional fuzzing tools, the byte-level mutations on raw inputs will lead to an enormous number of syntax errors. We believe that approaches like ZEST complement CGF tools in testing different components of programs.

5.6 Summary

In this chapter, we presented ZEST, a technique for generating inputs to test complex input-processing pipelines. ZEST leverages the domain knowledge in handwritten QuickCheck-like generator functions, which can be used to sample syntactically valid inputs, as well as predicates that specify whether an input is semantically valid. ZEST introduces *semantic fuzzing*, a technique to automatically guide QuickCheck-like generators towards producing inputs that are both likely to be semantically valid and increase code coverage in the program under test. We also presented an experimental evaluation of ZEST on five real-world Java programs. Our evaluation showed that ZEST outperforms vanilla `junit-quickcheck` as well as AFL in generating inputs that exercise code paths within the semantic analysis stages of our test programs. ZEST was also able to find previously unknown bugs in these programs more quickly and reliably than either `junit-quickcheck` or AFL.

We also presented JQF, a framework for controlling the pseudo-random choices made in `junit-quickcheck` generators. JQF allows researchers to develop new coverage-guided algorithms for generating structured inputs for Java programs. ZEST is implemented in JQF,

along with several other guidance algorithms. JQF has been used to find 42 previously unknown bugs in widely used open-source Java software, of which more than half have already been fixed.

Together, JQF+ZEST have shown that a combination of domain expertise—provided by software developers in the form of input generators and validity predicates—with sophisticated coverage-guided fuzzing algorithms can effectively perform automated testing of large complex programs that process highlight structured inputs.

Chapter 6

FUZZFACTORY: Domain-Specific Fuzzing with Waypoints

Fuzz testing has applications beyond finding program crashes. In Chapter 4, we saw how PERFFUZZ can be used to find performance hot spots. In Chapter 5, we saw how JQF and ZEST can be used to perform property-based testing. Fuzz testing can also be used for directed testing [23], differential testing [148], side-channel analysis [130], discovering algorithmic complexity vulnerabilities [149]. In each case, researchers have modified the original fuzzing algorithm to produce a specialized solution. Similarly, researchers have tweaked the original CGF algorithm to leverage domain-specific information from programs in order to improve code coverage, such as the use of magic bytes in file formats [102, 155, 109] or measures of input validity [142, 101, 150].

In this chapter, we present FUZZFACTORY [143], a framework for implementing domain-specific fuzzing applications. Our framework is based on the following observation: many domain-specific fuzzing problems can be solved by augmenting the coverage-guided fuzzing algorithm to selectively save newly generated inputs for subsequent mutation, beyond those that only improve code coverage. We call these intermediate inputs *waypoints*, inspired by the corresponding term in the field of navigation. These waypoints give the fuzzing algorithm steps towards a domain-specific goal. A *domain-specific fuzzing application* for domain d is specified via a predicate: *is_waypoint*$(i, \mathcal{S}, d)$. This predicate answers the following question: given a newly generated input i and a set of previously saved inputs $\mathcal{S}$, should we save input i to $\mathcal{S}$? FUZZFACTORY provides a simple mechanism for defining *is_waypoint*, based on *domain-specific feedback* that can be dynamically collected during test execution. A domain-specific fuzzing application can instrument programs under test to collect such custom feedback via a small set of APIs provided by FUZZFACTORY.

FUZZFACTORY enables development of domain-specific fuzzing applications without requiring changes to the underlying search algorithm. We were able to easily re-implement three algorithms from prior work and evaluate their strengths and weaknesses: SlowFuzz [149], PERFFUZZ [107], and validity fuzzing [142]. We also used FUZZFACTORY to prototype three novel applications: for smoothing hard comparisons, for generating inputs that allocate ex-

```
1  void* Test(int16_t a, int16_t b) {
2    if (a % 3 == 2) {
3      if (a > 0x1000) {
4        if (b >= 0x0123) {
5          if (a == b) {
6            abort();
7          } else {
8            return malloc(a);
9          }
10       }
11     }
12   }
13 }
```
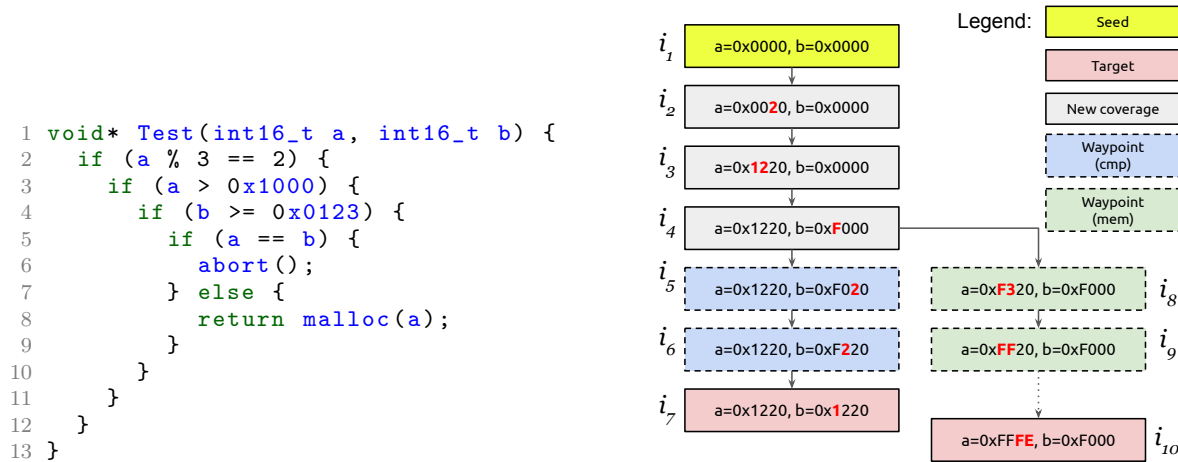
(a) Sample function in the test program. Parameters a and b are the test inputs.

(b) Sample fuzzed inputs starting with initial seed $a = 0$, $b = 0$. Arrows indicate mutations.

Figure 6.1: A motivating example

cessive amounts of memory, and to perform incremental fuzzing following code changes. We describe these six domain-specific fuzzing applications as well as our experimental results on six real-world benchmark programs from a test suite released by Google [77].

A key advantage of FUZZFACTORY is that domain-specific feedback is naturally composable. We combine our domain-specific fuzzing applications for exacerbating memory allocations and for smoothing hard comparisons to produce a composite application that performs better than each of its constituents. The composite application automatically generates LZ4 bombs and PNG bombs: tiny inputs that lead to dynamic allocations of 4GB in `libarchive` and 2GB in `libpng` respectively.

FUZZFACTORY has been made publicly available at: `https://github.com/rohanpadhye/FuzzFactory`.

6.1 Motivation

Consider the sample test program in Figure 6.1a. The function `Test` takes as input two 16-bit integers, a and b. A common test objective is to generate inputs that maximize code coverage in this program. We apply Algorithm 1 to perform CGF on this test program. Let us assume that we start with the *seed input*: a=0x0000, b=0x0000. The seed input does not satisfy the condition at Line 2. The CGF algorithm randomly mutates this seed input and executes the test program on the mutated inputs while looking for new code coverage. Figure 6.1b depicts in grey boxes a series of sample inputs which may be saved by CGF, starting with the initial seed input i_1 in an yellow box. A solid arrow between two inputs, say

i and i', indicates that the input i is mutated to generate i'. After some attempts, CGF may mutate the value of `a` in i_1 to a value such as 0x0020, which satisfies the condition at Line 2. Since such an input leads to new code being executed, it gets saved to $\mathcal{S}$. In Fig. 6.1b, this is input i_2. Small, byte-level mutations enable CGF to subsequently generate inputs that satisfy the branch condition at Line 3 and Line 4 of Fig. 6.1a. This is because there are many possible solutions that satisfy the comparisons `a > 0x1000` and `b >= 0x0123` respectively; we call these *soft* comparisons. Fig. 6.1b shows the corresponding inputs in our example: i_3 and i_4. However, it is much more difficult for CGF to generate inputs to satisfy comparisons such as `a == b` at Line 5; we call these *hard* comparisons. Random byte-level mutations on inputs i_1–i_4 are unlikely to produce an input where `a == b`. Therefore, the code at Line 6 may not be exercised in a reasonable amount of time using conventional CGF.

Now, consider another test objective, where we would like to generate inputs that maximize the amount of memory that is dynamically allocated via `malloc`. This objective is useful for generating stress tests or to discover potential out-of-memory related bugs. The CGF algorithm enables us to generate inputs that invoke `malloc` statement at Line 8, such as i_4. However, this input only allocates `0x1220` bytes (i.e., just over 4KB) of memory. Although random mutations on this input are likely to generate inputs that allocate larger amount of memory, CGF will never save these because they have the same coverage as i_4. Thus, it is unlikely that CGF will discover the *maximum* memory-allocating input in a reasonable amount of time.

6.1.1 Waypoints

Both of the challenges listed above can be addressed if we save some useful intermediate inputs to $\mathcal{S}$ regardless of whether they increase code coverage. Then, random mutations on these intermediate inputs may produce inputs achieving our test objectives. We call these intermediate inputs *waypoints.* For example, to overcome hard comparisons such as `a == b`, we want to save intermediate inputs if they maximize the number of common bits between `a` and `b`. Let us call this strategy `cmp`. The blue boxes in Fig. 6.1b show inputs that may be saved to $\mathcal{S}$ when using the `cmp` strategy for waypoints. In such a strategy, the inputs i_5 and i_6 are saved to $\mathcal{S}$ even though they do not achieve new code coverage. Now, input i_6 can easily be mutated to input i_7, which satisfies the condition `a == b`. Thus, we easily discover an input that triggers `abort` at Line 6 of Fig. 6.1a. Similarly, to achieve the objective of maximizing memory allocation, we save waypoints that allocate more memory at a given call to `malloc` than any other input in $\mathcal{S}$. Fig. 6.1b shows sample waypoints i_8 and i_9 that may be saved with this strategy, called `mem`. The dotted arrow from i_9 to i_{10} indicates that, after several such waypoints, random mutations will eventually lead us to generating input i_{10}. This input causes the test program to allocate the maximum possible memory at Line 8, which is almost 64KB.

Now, consider a change to the condition at Line 4 of Figure 6.1a. Instead of an inequality, suppose the condition is `b == 0x0123`. To generate inputs that invoke `malloc` at Line 8, we first need to overcome a hard comparison of `b` with `0x0123`. We can combine the two

strategies for saving waypoints as follows: save a new input i if *either* it increases the number of common bits between operands of hard comparisons *or* if it increases the amount of memory allocated at some call to `malloc`. In Section 6.4, we demonstrate how a combination of these strategies allows us to automatically generate PNG bombs and LZ4 bombs, i.e. tiny inputs that allocate 2–4 GB of memory, when fuzzing `libpng` and `libarchive` respectively.

We propose a framework, called FUZZFACTORY, which enables users to implement strategies for choosing waypoints. To do so, the user specifies what custom feedback they need to collect from the execution of a program under test in addition to coverage information. The user also specifies a function for aggregating such feedback across a collection of inputs; the aggregated feedback is used to decide whether an input should be considered a waypoint.

We next describe the framework and its underlying algorithm. The framework has enabled us to rapidly implement three existing strategies in the literature and four new strategies, including a composite strategy.

6.2 The FUZZFACTORY Framework

Our goal is to construct a framework which allows users to build a domain-specific fuzzing application d by simply defining a custom predicate: $is\_waypoint(i, \mathcal{S}, d)$. The predicate tells the fuzzer whether a new input i is a *waypoint*; that is, whether i should to be saved to the set of saved inputs $\mathcal{S}$ so that later on it can be mutated to generate new inputs.

In the conventional CGF algorithm, the decision of whether to save an input is defined in terms of the dynamic behavior of the program on the input i. Specifically, if the coverage of the program on the input i includes a coverage point that is not present in the coverage cumulatively attained by the program on the inputs in $\mathcal{S}$, then CGF deems i as interesting and saves it to $\mathcal{S}$. The decision is based on a specific kind of feedback (i.e. coverage) from the execution of the program on i. The feedback is directly related to the goal of CGF, which is to increase the coverage of the program.

Although improving code coverage is important for discovering new program behavior, we believe that a fuzzer could be made more effective and diverse if it was guided by other testing goals, such as: discovering performance bottlenecks or memory usage problems, covering recently modified code, exercising valid input behavior, etc.

FUZZFACTORY enables users to prototype fuzzers that target user-defined custom goals. To support custom or domain-specific goals, the user needs to specify: (1) the specific kind of feedback to collect from the execution of the program on any input, and (2) how this feedback should be used to determine if the input should be considered interesting and saved.

We next describe the mechanism with which the FUZZFACTORY user specifies the kind of domain-specific feedback they want from an execution. We then explain how the $is\_waypoint$ predicate uses such custom feedback to determine if an input needs to be saved. We also describe how to compose such domain-specific feedback. Finally, we show how to extend the CGF algorithm in Algorithm 1 to take domain-specific feedback into account.

6.2.1 Domain-Specific Feedback

In FUZZFACTORY, we provide a mechanism for users to specify a *domain* and to collect custom *domain-specific feedback* (DSF) from the execution of the program under test. A domain-specific feedback (DSF) is a map of the form $dsf_i : K \to V$, where i is a program input, K is a set of keys (e.g. program locations) and V is a set of values (usually a measurement of something we want to optimize). The map is populated by executing the program under test on input i. As an example, if we are interested in generating inputs on which the program execution increases memory allocation, then dsf_i is a map from $\mathbb{L}$ to $\mathbb{N}$, where $\mathbb{L}$ is the set of program locations where a memory allocation function (e.g. `malloc`) is called and $\mathbb{N}$ is the set of natural numbers. $dsf_i(k)$ represents the total amount of memory in bytes that is allocated at program location k during the execution of the program on the test input i.

In general, the user specifies a domain as a tuple of the form $d = (K, V, A, a_0, \rhd)$ where K is a set of keys, V is a set of values, A is a set of aggregation values, a_0 is an initial aggregation value, and $\rhd : A \times V \to A$ is a reducer function. The user specifies how to update the map dsf_i during an execution of the test program on input i, by inserting appropriate instrumentation in the test program. We explain the meaning of A, a_0, and $\rhd$ in a user-defined domain in the next subsection.

6.2.2 Waypoints

We use the dsf_i map from the execution of the test program on input i in order to determine if i needs to be saved. To do so, FUZZFACTORY aggregates the domain-specific feedback collected from the executions of multiple test inputs into a value that belongs to the user-defined set A. To compute this aggregate value, the user provides an initial aggregate value $a_0 \in A$ and a *reducer* function $\rhd : A \times V \to A$ as part of the domain. A reducer function must satisfy the following properties for any $a \in A$ and any $v, v' \in V$:

$$a \rhd v \rhd v = a \rhd v \tag{6.1}$$
$$a \rhd v \rhd v' = a \rhd v' \rhd v \tag{6.2}$$

These rules imply idempotence and application-order insensitivity, respectively, in the second operand. For the memory-allocation domain (say d^{mem}): both V and A are the set of natural numbers $\mathbb{N}$. The initial aggregate value $a_0 = 0$, and $\rhd$ is the *max* operation on natural numbers. We can therefore define $d^{mem} = (\mathbb{L}, \mathbb{N}, \mathbb{N}, 0, max)$. Property 6.1 is satisfied because $\max(\max(a, v), v) = \max(a, v)$ for any $a, v \in \mathbb{N}$. Property 6.2 is satisfied because $\max(\max(a, v), v') = \max(\max(a, v'), v)$ for any $a, v, v' \in \mathbb{N}$. The properties help ensure that the every saved waypoint contributes towards domain-specific progress; this point will be visited when encountering Theorem 2 below. Note that these properties are not statically verified by FUZZFACTORY; it is the responsibility of the user to ensure that their chosen reducer function satisfies Properties 6.1 and 6.2.

In general, let dsf_i be the DSF map populated during the execution of program p with i. For a given set of inputs $\mathcal{S} = \{i_1, i_2, \ldots, i_n\}$, we define the aggregated domain-specific feedback value $\mathcal{A}(\mathcal{S}, k, d)$ for the domain d and for key $k \in K$ as follows:

$$\mathcal{A}(\mathcal{S}, k, d) \stackrel{\text{def}}{=} a_0 \triangleright dsf_{i_1}(k) \triangleright dsf_{i_2}(k) \triangleright \ldots \triangleright dsf_{i_n}(k), \text{ where } d = (K, V, A, a_0, \triangleright) \quad (6.3)$$

Due to the Properties 6.1 and 6.2, the value of $\mathcal{A}(\mathcal{S}, k, d)$ is uniquely defined; the choice of ordering $i_1, \ldots, i_n$ does not matter.

For the memory-allocation domain, the aggregated feedback value $\mathcal{A}(\mathcal{S}, k, d^{mem})$ represents the *maximum* amount of memory allocated at program location $k \in \mathbb{L}$ across all inputs in $\mathcal{S}$. For this domain, we would like to save an input i to set $\mathcal{S}$ if the execution on i causes more memory allocation at some program location k than that of any of the allocations observed at k during the execution of the inputs in $\mathcal{S}$.

In FUZZFACTORY, we define the predicate $is\_waypoint(i, \mathcal{S}, d)$ as follows:

$$is\_waypoint(i, \mathcal{S}, d) \stackrel{\text{def}}{=} \exists k \in K : \mathcal{A}(\mathcal{S}, k, d) \neq \mathcal{A}(\mathcal{S} \cup \{i\}, k, d), \text{ where } d = (K, V, A, a_0, \triangleright)$$
$$(6.4)$$

The definition implies that we will save input i if the execution on the input results in a change in the aggregated domain-specific feedback value for some key.

6.2.2.1 Monotonicity of Aggregation

In order to decide if an input i should be considered a waypoint, we only check if the total aggregation *changes*; i.e., whether $\mathcal{A}(\mathcal{S}, k, d) \neq \mathcal{A}(\mathcal{S} \cup \{i\}, k, d)$. However an important consequence of Properties 6.1 and 6.2 is that this change is always in a direction that implies some sort of *domain-specific progress*, denoted by a partial order $\preceq$ on A. In other words, the function $\mathcal{A}$ is monotonic in its first argument with respect to partial order $\preceq$. For example, in the memory allocation domain d^{mem}: if $\mathcal{A}(\mathcal{S}, k, d^{mem}) \neq \mathcal{A}(\mathcal{S} \cup \{i\}, k, d^{mem})$ for some program location $k \in \mathbb{L}$, this means that the memory allocated at k during the execution of i is *more* than the memory allocated at k by any other input in $\mathcal{S}$. The partial order in this example is simply the total ordering on natural numbers: $\leq$. More generally, we can state the following theorem:

Theorem 2 (Monotonicity of Aggregation). A domain $d = (K, V, A, a_0, \triangleright)$ whose reducer function $\triangleright$ satisfies properties 6.1 and 6.2 imposes a partial order $\preceq$ on A such that the function $\mathcal{A}$ is monotonic in its first argument with respect to $\preceq$. That is, the following always holds for any such domain d, any key $k \in K$, and for some binary relation $\preceq$ on A:

$$\mathcal{S}_1 \subseteq \mathcal{S}_2 \Rightarrow \mathcal{A}(\mathcal{S}_1, k, d) \preceq \mathcal{A}(\mathcal{S}_2, k, d)$$

In order to prove Theorem 2, we first need to demonstrate a few lemmas.

Lemma 1 (No ping-pong). Given a reducer function $\triangleright : A \times V \to A$ satisfying Properties 6.1 and 6.2, then $\forall a \in A$ and any $n \geq 0$ terms $v_1, \ldots, v_n \in V$, if $a \triangleright v_1 \triangleright \ldots \triangleright v_n = a$, then:

$$\forall 0 \leq k \leq n : a \triangleright v_1 \triangleright \ldots \triangleright v_k = a$$

In other words, if we start with aggregate value a and then apply n reductions, and if the final result is also the value a, then the result of all the intermediate reductions must also be a. This lemma states that aggregate values cannot *ping-pong*; that is, they cannot oscillate between distinct values.

Proof. For $n = 0$, the lemma is trivially true. For $n > 0$, we prove the lemma by contradiction: given that $a \triangleright v_1 \triangleright \ldots \triangleright v_n = a$, assume that there exists some k, where $1 \leq k \leq n$, such that $a \neq a \triangleright v_1 \triangleright \ldots \triangleright v_k$. In this inequality, we can substitute the value of a on both sides with the equivalent $a \triangleright v_1 \triangleright \ldots \triangleright v_n$, to get:

$$a \triangleright v_1 \triangleright \ldots \triangleright v_n \neq a \triangleright v_1 \triangleright \ldots \triangleright v_n \triangleright v_1 \triangleright \ldots \triangleright v_k$$

Then, we can repeatedly apply Property 6.2 on the right-hand side to rearrange terms:

$$a \triangleright v_1 \triangleright \ldots \triangleright v_n \neq a \triangleright v_1 \triangleright v_1 \triangleright v_2 \triangleright v_2 \triangleright \ldots \triangleright v_k \triangleright v_k \triangleright v_{k+1} \triangleright v_{k+2} \triangleright \ldots \triangleright v_n$$

Then, we can repeatedly apply Property 6.1 on the right-hand side to remove redundant terms:

$$a \triangleright v_1 \triangleright \ldots \triangleright v_n \neq a \triangleright v_1 \triangleright \ldots \triangleright v_n$$

This is a contradiction; therefore, no such k can exist. $\qquad\square$

Definition 17 (Progress). If $\triangleright : A \times V \to A$ is a reducer function, then we can define a binary relation $\preceq$ on A called *progress* as follows:

$$a \preceq b \Leftrightarrow \exists \, v_1, \ldots, v_n \in V, \text{ where } n \geq 0, \text{ such that } a \triangleright v_1 \triangleright \ldots \triangleright v_n = b$$

Lemma 2 (Reflexivity of progress). If $\triangleright : A \times V \to A$ is a reducer function and $\preceq$ is its progress relation, then $\forall a \in A : a \preceq a$.

Proof. Straightforward from Definition 17 with $n = 0$. $\qquad\square$

Lemma 3 (Transitivity of progress). If $\triangleright : A \times V \to A$ is a reducer function and $\preceq$ is its progress relation, then $\forall a, b, c \in A : a \preceq b \land b \preceq c \Rightarrow a \preceq c$.

Proof. If $a \preceq b$ and if $b \preceq c$, then by Definition 17 there exist some terms $u_1, \ldots, u_m \in V$ and $v_1, \ldots, v_n \in V$ for $m, n \geq 0$ such that:

$$a \triangleright u_1 \triangleright \ldots \triangleright u_m = b \tag{6.5}$$

$$b \triangleright v_1 \triangleright \ldots \triangleright v_n = c \tag{6.6}$$

Substituting the b on the LHS of Equation 6.6 with the LHS of Equation 6.5, we can write:

$$a \triangleright u_1 \triangleright \ldots \triangleright u_m \triangleright v_1 \triangleright \ldots \triangleright v_n = c \tag{6.7}$$

Which, by Definition 17, means $a \preceq c$. $\qquad\square$

Lemma 4 (Anti-symmetry of progress). *If* $\rhd : A \times V \to A$ *is a reducer function and* $\preceq$ *is its progress relation, then* $a \preceq b \wedge b \preceq a \Rightarrow a = b$.

Proof. If $a \preceq b$ and if $b \preceq a$ then by Definition 17 there exist some terms $u_1, \ldots, u_m \in V$ and $v_1, \ldots, v_n \in V$ for $m, n \geq 0$ such that:

$$a \rhd u_1 \rhd \ldots \rhd u_m = b \tag{6.8}$$

$$b \rhd v_1 \rhd \ldots \rhd v_n = a. \tag{6.9}$$

Substituting the b on the LHS of Equation 6.9 with the LHS of Equation 6.8, we can write:

$$a \rhd u_1 \rhd \ldots \rhd u_m \rhd v_1 \rhd \ldots \rhd v_n = a.$$

By Lemma 1, all intermediate aggregates must be equal to a, in particular:

$$a \rhd u_1 \rhd \ldots \rhd u_m = a$$

Plugging this result into the LHS of Equation 6.8, we get $a = b$. $\qquad\square$

Proof of Theorem 2. Let $\preceq$ be the progress relation for the reducer $\rhd$. From Lemmas 2, 3, and 4, it follows that this relation is a *partial order*. Now, let $S_1 \subseteq S_2$. From the definition of $\mathcal{A}$ in Equation 6.3, we can write:

$$\mathcal{A}(\mathcal{S}_2, k, d) = \mathcal{A}(\mathcal{S}_1, k, d) \rhd v_1 \rhd \ldots v_n$$

where $\{v_1, \ldots, v_n\} = \mathcal{S}_2 \setminus \mathcal{S}_1$. From Definition 17, this implies that $\mathcal{A}(\mathcal{S}_1, k, d) \preceq \mathcal{A}(\mathcal{S}_2, k, d)$; that is, $\mathcal{A}$ is *monotonic* in its first argument with respect to $\preceq$. $\qquad\square$

Corollary 1. An input i is considered a waypoint iff the aggregated domain-specific feedback strictly makes progress for some key k, without sacrificing progress for any other key. In particular:

$$
\begin{aligned}
is\_waypoint(i, \mathcal{S}, d) \Leftrightarrow &(\forall k \in K : \mathcal{A}(\mathcal{S}, k, d) \preceq \mathcal{A}(\mathcal{S} \cup \{i\}, k, d)) \\
&\wedge (\exists k \in K : \mathcal{A}(\mathcal{S}, k, d) \prec \mathcal{A}(\mathcal{S} \cup \{i\}, k, d))
\end{aligned}
$$

where $a \prec b \Leftrightarrow a \preceq b \wedge a \neq b$

Proof. Follows from the definition of *is_waypoint* in Eq. 6.4 and Theorem 2. $\qquad\square$

Algorithm 4 The domain-specific fuzzing algorithm. The grey boxes indicate changes to Algorithm 1.

Input: an instrumented test program p, a set of initial seed inputs $\mathcal{I}$, a set of domain-specific feedback D

Output: a corpus of automatically generated inputs $\mathcal{S}$

1: $\mathcal{S} \leftarrow \mathcal{I}$
2: $totalCoverage \leftarrow \emptyset$

3: **repeat** ▷ Main fuzzing loop
4:   **for** i in $\mathcal{S}$ **do**
5:     **if** sample FUZZPROB(i) **then**
6:       $i' \leftarrow$ MUTATE(i) ▷ Generate new test input i'
7:       $coverage,\ \boxed{dsf_{i'}^{1}, \ldots, dsf_{i'}^{|D|}} \leftarrow$ EXECUTE(p, i') ▷ Run test with new input i'
8:       **if** $coverage \cap totalCoverage \neq \emptyset$ **then**
9:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{i'\}$ ▷ Save i' if new code coverage achieved
10:         $totalCoverage \leftarrow totalCoverage \cup coverage$
11:       **end if**
12:       **if** $is\_waypoint(i', \mathcal{S}, D)$ **then**
13:         $\mathcal{S} \leftarrow \mathcal{S} \cup \{i'\}$ ▷ Save i' to fuzzing corpus
14:       **end if**
15:     **end if**
16:   **end for**
17: **until** given time budget expires
18: **return** $\mathcal{S}$

6.2.3 Composing Domains

FUZZFACTORY allows the user to naturally compose multiple domains for a program under test. This enables fuzzing to target multiple goals simultaneously.

Assume that the user has specified a set of domains D, where $d = (K, V, A, a_0, \triangleright)$ for each $d \in D$. Then we extend the definition of the predicate $is\_waypoint$ to D as follows:

$$is\_waypoint(i, \mathcal{S}, D) \stackrel{\text{def}}{=} \bigvee_{d \in D} is\_waypoint(i, \mathcal{S}, d) \qquad (6.10)$$

which says that $is\_waypoint(i, \mathcal{S}, D)$ is true for a set of domains D iff $is\_waypoint(i, \mathcal{S}, d)$ is true for some domain $d \in D$. We save the input i in $\mathcal{S}$ if $is\_waypoint(i, \mathcal{S}, D)$ is true. Note that Corollary 1 naturally extends to a composition of multiple domains: $is\_waypoint(i, \mathcal{S}, D)$ implies strict progress in at least one key in at least one domain $d \in D$.

6.2.4 Algorithm for Domain-Specific Fuzzing

Algorithm 4 describes the domain-specific fuzzing algorithm implemented in FUZZFACTORY. The algorithm extends the conventional coverage-guided fuzzing algorithm described in Algorithm 1. The extensions are marked with grey background. The extension is quite straightforward: during the execution of the program p on an input i', the algorithm not only collects *coverage*, but also collects domain-specific feedback maps $dsf_{i'}^1, \ldots, dsf_{i'}^{|D|}$ for each domain in D. It then uses those maps in the call to $is\_waypoint(i', \mathcal{S}, D)$ to determine if the new input i' should be added to the set of saved inputs $\mathcal{S}$. Unlike Algorithm 1, the FUZZFACTORY algorithm does not necessarily track *failures*, although failure tracking can be easily added to Algorithm 4. This is because not all domain-specific testing objectives have readily identifiable failure cases; for example, when trying to discover performance bottlenecks (ref. Chapter 4). The high-level goal of Algorithm 4 is to generate a corpus of test inputs that may be interesting to investigate in a domain-specific post-processing step.

6.3 Domain-Specific Fuzzing Applications

We demonstrate the applicability of FUZZFACTORY by instantiating six independent domain-specific fuzzing applications. Some of these fuzzing algorithms were already proposed and implemented in prior work. Our motivation behind implementing these algorithms was to evaluate whether we could prototype these algorithms in our framework, without changing the underlying fuzzing algorithm or search heuristics. Sections 6.3.3 through 6.3.8 describe six domains, in increasing order of complexity:

1. `slow`: An application for maximizing execution path lengths, based on SlowFuzz [149]. This is the most trivial domain to implement in FUZZFACTORY.

2. `perf`: An application for discovering hot spots by maximizing basic block execution counts, based on PERFFUZZ (Chapter 4). In FUZZFACTORY, this naturally generalizes `slow`.

3. `mem`: A novel application for generating inputs that maximize dynamic memory allocations.

4. `valid`: An application of the validity fuzzing algorithm [142], which attempts to bias input generation towards inputs that satisfy program-specific validity checks.

5. `cmp`: A domain for smoothing hard comparisons. Although a lot of prior work address this application, our particular solution strategy is novel.

6. `diff`: A novel application for incremental fuzzing after code changes in a test program.

For each application, (1) we define the domain d in terms of the tuple $(K, V, A, a_0, \rhd)$ (2) we describe, with the help of some utilities defined Table 6.1, how we instrument test

Table 6.1: Definition of instrumentation functions used for injecting code which updates domain-specific feedback maps. They are used in Table 6.2 through 6.7. Hooks are activated when corresponding syntactic objects are encountered during a compile-time pass over the program under test. The handler logic for these hooks can inject code in the program under test. Actions are the functions that are used to actually inject code during instrumentation. Utility functions are available to the handler logic at compile-time.

Instrumentation Hooks	Description
$new\_basic\_block()$	Activated at the beginning of a basic block in the control-flow graph of the program under test.
$entry\_point()$	Activated at the entry point for test execution (e.g. start of the `main` function).
$fn\_call(name, args)$	Activated at an expression that invokes function named *name* with arguments *args*.
$binop(type, left, op, right)$	Activated at an expression with binary operator of the form '*left op right*' (e.g. `x == 42`), where the operands have type *type* (e.g. `long`).
$switch(type, val, cases)$	Activated when encountering a `switch` statement on value *val* of type *type*, where *cases* is a list of the `case` clauses.

Instrumentation Actions	Description
$ins\_after(inst)$	Inserts an instruction *inst* immediately after the instruction whose instrumentation hook is currently activated.
$ins\_before(inst)$	Inserts an instruction *inst* immediately before the instruction whose instrumentation hook is currently activated.

Utility functions	Description
$current\_program\_loc()$	Returns the program location (i.e., a value in set $\mathbb{L}$) corresponding to the current instrumentation location.
$target\_program\_loc(case)$	Returns the program location (i.e., a value in set $\mathbb{L}$) that is the target of a `case` within a `switch` statement.
$comm\_bits(a, b, n)$	Counts the number of common bits between two *n*-byte operands a and b. For example, $comm\_bits(1025, 1026, 4) = 30$, since only 2 bits in these 32-bit operands differ.

programs to populate the map dsf_i during test execution on input i^1, and (3) we report the results of applying the domain-specific fuzzing implementation to a set of real-world programs.

A key advantage of FUZZFACTORY is that it enables us to naturally compose multiple domain-specific fuzzing applications with no extra effort. In Section 6.4, we describe a composition of `cmp` and `mem` that smooths hard comparisons in order to exacerbate memory allocations. Remarkably, we find that such a composition can perform better than just the sum of its parts.

6.3.1 Program Instrumentation

Sections 6.3.3 through 6.3.8 describe how test programs are instrumented to implement each of the six domains that we present in this chapter. The instrumentation enables the collection of domain-specific feedback in the map dsf_i when executing the test program on an input i. Such instrumentation is performed at compile-time. Although our implementations performs instrumentation at the LLVM IR level, for ease of presentation we describe the instrumentation logic for each of the six domains at a higher level of abstraction. Table 6.1 lists some hooks, actions, and utility functions that we use in our abstract descriptions of domain-specific instrumentation. We next describe how to interpret the information in Table 6.1.

A *hook* is activated at compile-time by an instrumentation framework (e.g. LLVM) whenever a corresponding element in a program is encountered while making a pass over the test program. For example, the $fn\_call(name, args)$ hook is invoked at compile-time for every function call expression in the program. Here, *name* is a string and *args* is a list of references to the syntactic expressions that form the arguments to the function call. An instrumentation pass, such as the one we write for each fuzzing domain, specifies some logic to handle such hooks. The handler logic can optionally insert new code before or after the program element whose hook is currently activated. For example, a handler for fn_call can statically look at *name* (say f) to decide whether to insert code around a call to f. Code is inserted by invoking *actions* such as ins_after and ins_before. The inserted code can use compile-time constants or refer to static program elements such as: one or more arguments to f, global variables, or user-defined functions. For ease of presentation, we will show the inserted code as source-level pseudocode instead of an instruction in some IR. Commonly, we will insert code that updates the dsf_i map—in practice, we insert an instruction that invokes one of the APIs listed in Section 6.6.1. The handler logic is unrestricted; in our implementation, it is arbitrary C++ code that uses the LLVM API. The handler logic can make use of *utility functions* provided by FUZZFACTORY at compile-time. Table 6.1 only lists the hooks and utility functions required to describe the six domains presented in this chapter (Tables 6.2–6.7). To implement new domains, other language constructs such as branches, loads, stores, etc. can also be instrumented.

---

[1]We will drop the subscript i from dsf_i when it is clear from context.

6.3.2 Experimental Setup

For our experiments, we use six benchmark programs from the Google fuzzing test suite [77]. This suite contains specific historical versions of programs that have been thoroughly fuzzed using the OSS-fuzz infrastructure [75]. The six benchmarks we use include: (1) libpng-1.2.56, (2) libarchive-2017-01-04, (3) libjpeg-turbo-07-2017, (4) libxml2-v2.9.2, (5) vorbis-2017-12-11, and (6) boringssl-2016-02-12.[2] The benchmarks are written in C or C++. Benchmarks (1)–(4) were chosen because they are commonly used in the fuzzing literature [107, 108, 147, 34, 35, 150]. Benchmarks `vorbis` and `boringssl` were chosen because they expect markedly different input formats. We only used six benchmarks from Google's test suite because of resource constraints: for our evaluation, we spent two CPU-years fuzzing these six benchmarks alone.

All experiments were run on Amazon AWS 'c5.18xlarge' instances. Each experiment was repeated 12 times to account for variability in the randomized algorithms. Unless otherwise stated, our fuzzing experiments used the initial seed inputs provided in the benchmark suite, limited input sizes to at most 10KB during fuzzing, and were run for 24 hours at a time.

For each application, we evaluate the following research question: "*Does* FuzzFactory *help achieve domain-specific fuzzing goals, without modifying the underlying search algorithm?*". FuzzFactory is implemented as an extension to AFL, and inherits its mutation and search heuristics (ref. Section 2.5.1). For each application domain, we thus compare the results of domain-specific fuzzing with the baseline: conventional coverage-guided fuzzing using AFL. Naturally, the metrics on which we perform this comparison vary depending on the domain.

6.3.3 `slow`: Maximizing Execution Path Length

Fuzz testing can be used to generate inputs that exacerbate the algorithmic complexity of a program under test. SlowFuzz [149] introduced this idea using a resource-guided evolutionary search. The search uses a fitness function that counts the number of basic blocks executed during the execution of a single test input. We call this metric the *execution path length*.

Our first domain-specific fuzzing application is a port of SlowFuzz to our framework. The goal of this application is to generate inputs that maximize the execution path length in the program under test. We want to define the $is\_waypoint(i, \mathcal{S}, d)$ predicate as follows: an input i should be saved if its execution leads to a higher path length than any other input in $\mathcal{S}$.

The first row of Table 6.2 defines this domain (say d) as follows. The domain-specific feedback map dsf maps the single key 0 ($K = \{0\}$) to a natural number ($V = \mathbb{N}$). In the map, $dsf(0)$ represents the execution path length for a test input i. These values are aggregated into a number ($A = \mathbb{N}$) which represents the maximum execution path length observed across a set of inputs ($a_0 = 0$, $a \triangleright v = \max(a, v)$).

---

[2]For `boringssl`, we use the target `fuzz/server.cc`, which fuzzes the server side of the TLS handshake protocol, instead of the default `fuzz/privkey.cc`, which fuzzes the parsing of private key files.

Table 6.2: `slow`: Application for maximizing execution path length

Domain d: $K = \{0\}, V = \mathbb{N}, A = \mathbb{N}, a_0 = 0, \ a \triangleright v = \max(a, v)$	
Hook	**Instrumentation**
$entry\_point()$	**ins_after**('$dsf(0) \leftarrow 0$')
$new\_basic\_block()$	**ins_after**('$dsf(0) \leftarrow dsf(0) + 1$')
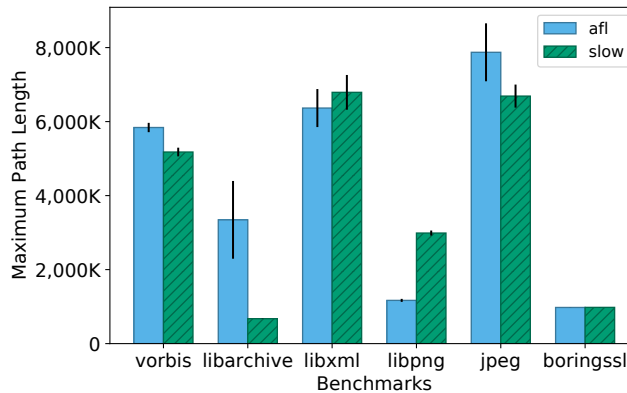
Figure 6.2: Maximum execution path lengths achieved by baseline (afl) and domain-specific fuzzing application (slow). Higher is better.

Table 6.2 also describes how we instrument test programs to correctly update entries in the map dsf at run-time. We make use of the instrumentation hooks $entry\_point$ and new_basic_block, and the action ins_after, all defined in Table 6.1. Using these functions, we can interpret the description in Table 6.2 as follows: At the entry point of the program under test, insert a statement that sets $dsf(0)$ to 0. Then, at each basic block in the program, insert a statement that increments the value stored at $dsf(0)$. Thus, during a test execution, the value of $dsf(0)$ is incremented by one each time a basic block is visited. At the end of the test input execution, the value of $dsf(0)$ will contain the execution path length. Since the reducer function for this domain is defined to be max with an initial value of 0 (see first row of Table 6.2), the aggregated value of the domain-specific feedback $\mathcal{A}(\mathcal{S}, 0, d)$ will be the maximum execution path length observed across all the inputs in $\mathcal{S}$.

6.3.3.1 Experimental Evaluation

Figure 6.2 shows the results of our experiments with this application on our benchmark programs. We evaluate the maximum execution path lengths (across the generated test corpus) for the baseline (`afl`) and our domain-specific fuzzing application (`slow`), after 24

hours of fuzzing. The figure plots the mean value and standard error of this metric across 12 repetitions. For `libpng`, the domain-specific feedback enables the generation of inputs whose path lengths are more than 2.5× that of the baseline. For `boringssl` and `libxml`, the increase is not as significant. Interestingly, the maximum execution path length for `slow` is actually *lower* than that found by `afl` on the remaining three benchmarks. One possible explanation for this result is that `slow` attempts to aggressively maximize execution path lengths starting from the very first input. On the other hand, `afl` spends its time maximizing code coverage and discovers longer execution paths in components of the test program that are not exercised by the seed inputs.

The difference is most noticible in `libarchive`. Among all of the benchmarks we considered, `libarchive` is the only benchmark for which the initial seed input provided in Google's test suite is invalid. That is, the initial seed input for `libarchive` leads the test program to exit early in an error state. Since AFL spends its 24 hours increasing only code coverage, it is able to eventually generate inputs that are valid archives (e.g. ZIP files), whose processing leads to longer execution paths. On benchmarks such as `libpng`, the provided seed input is valid and already covers interesting code paths within the test programs; therefore, `slow` is able to maximize path lengths effectively. This SlowFuzz-inspired approach appears to work best when initial seed inputs already provide good code coverage.

6.3.4 `perf`: Discovering Hot Spots

PERFFUZZ uses fuzz testing for generating inputs with pathological performance. As described in Chapter 4, PERFFUZZ independently maximizes execution counts for each basic block in the program under test. To do this, PERFFUZZ extends the coverage-guided fuzzing algorithm to save newly generated inputs if they increase the maximum observed execution count for *any* basic block. In this domain, the goal is to find inputs that execute the same basic block many times.

Table 6.3 describes how we implement PERFFUZZ in our framework. The first line defines the domain. The keys in the DSF map (i.e. K) range over the set of program locations $\mathbb{L}$. The values of the DSF map as well as the aggregated values represent execution counts (i.e. $V = \mathbb{N}$ and $A = \mathbb{N}$). The reducer function (i.e. $\triangleright$) is *max* with initial value $a_0 = 0$, just as in SlowFuzz.

Table 6.3 also describes how we instrument the program under test. At the start of every test execution (*entry_point*), we initialize the entire DSF map with values 0. Each time a new basic block k is visited, we increment the value stored at $dsf(k)$. This is done in the instrumentation hook function *new_basic_block*, using the *current_program_loc()* function to statically get the program location of the basic block being instrumented (ref. Table 6.1). At the end of test execution, $dsf(k)$ will contain the number of times that basic block k was executed. Since the reducer function is *max*, a newly generated input will be considered a waypoint if it increases the execution count for any basic block k in the test program.

Table 6.3: `perf`: Application for discovering hot spots

Domain d: $K = \mathbb{L}, V = \mathbb{N}, A = \mathbb{N}, a_0 = 0, \ a \triangleright v = \max(a, v)$	
Hook	**Instrumentation**
$entry\_point()$	**ins_after**('$\forall k \in K : dsf(k) \leftarrow 0$')
$new\_basic\_block()$	$k \leftarrow current\_program\_loc()$
	ins_after('$dsf(k) \leftarrow dsf(k) + 1$')

Figure 6.3: Maximum basic block execution counts achieved by baseline (afl) and domain-specific fuzzing application (perf). Higher is better.

6.3.4.1 Experimental Evaluation

Figure 6.3 contains the results of our experiments with this application on our benchmark programs. We evaluate the FUZZFACTORY domain-specific fuzzing application (`perf`) with the baseline (`afl`), on the metric *max hot spot*. As per Definition 16, the *max hot spot* is the maximum execution count for any basic block across all inputs in the generated test corpus. The figure plots the mean value and standard error of this metric across 12 repetitions.

Figure 6.3 shows that `perf` is able to generate inputs that significantly maximize hot spots for three of the six benchmarks: `vorbis`, `libpng`, and `libjpeg-turbo`. For `libpng` and `libjpeg-turbo`, the hot spots discovered by `perf` execute 2× and 1.7× more than those discovered by the baseline `afl`. For `libarchive`, the `perf` application performs much worse. Similar to the experiments reported in the previous section, the main problem here is that the initial seed inputs provided with `libarchive` lead to an early exit. Since baseline AFL spends more time increasing code coverage rather than basic block execution counts, it eventually generates valid archive files (e.g. ZIP). Given that `libarchive` is a program that performs decompression, the generation of a valid archive is sufficient to discover a huge hot

Table 6.4: `mem`: Application for exacerbating memory allocation

Domain d: $K = \mathbb{L}, V = \mathbb{N}, A = \mathbb{N}, a_0 = 0,\ a \rhd v = \max(a, v)$	
Hook	**Instrumentation**
$entry\_point()$	**ins_after**('$\forall k \in K : dsf(k) \leftarrow 0$')
$fn\_call(name, args)$	if $name \in \{\text{‘malloc’}, \text{‘calloc’}\}$:
	$\quad k \leftarrow current\_program\_loc()$
	$\quad bytes \leftarrow args[0]$
	$\quad$ **ins_after**('$dsf(k) \leftarrow dsf(k) + bytes$')
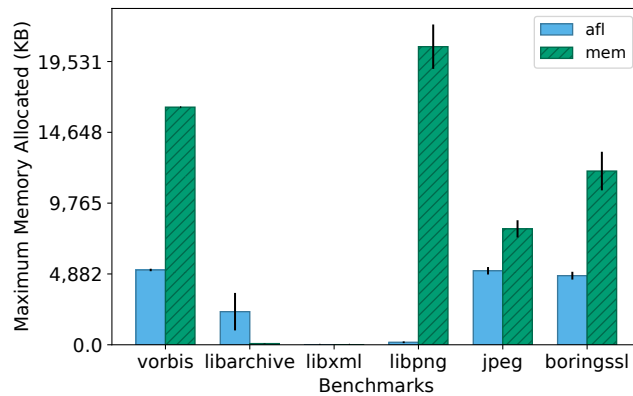
Figure 6.4: Maximum amount of dynamic memory allocated (in KB) due to inputs generated by baseline (afl) and domain-specific fuzzing application (mem). Higher is better.

spot in the code component that performs decompression. On the other hand, `perf` only discovers hot spots in `libarchive`'s parsing of file meta-data.

6.3.5 `mem`: Exacerbating Memory Allocations

We now describe a novel application of FUZZFACTORY: generating inputs that exacerbate memory allocation. There are several use cases for such a domain such as discovering the maximum amount of memory the program under test may dynamically allocate for a given size input, discovering inputs that could lead to bugs related to out-of-memory conditions, or generating a corpus of memory-stress tests for benchmarking purposes.

Table 6.4 describes our instrumentation for the memory-allocation domain. The definition of the domain on the first line of this table, as well as the initialization of *dsf* at the entry point, is exactly the same as that of the PERFFUZZ domain (Table 6.3). However, instead of incrementing the values in the DSF map at every basic block, we instrument expressions

```
1 void Test(uint8_t* data, int size) {        1 void Test(uint8_t* data, int size) {
2   /* set up png_ptr */                       2   /* set up png_ptr */
3   if (png_get_IHDR(png_ptr, ...) != 0)       3   assume(png_get_IHDR(png_ptr, ...)
4     return; // invalid header                4     == 0); // valid header
5   /* process PNG data */                     5   /* process PNG data */
6 }                                            6 }
```

(a) Original test driver   (b) Modified test driver

Figure 6.5: Sample change to `libpng` test driver to enable validity fuzzing.

in the test program that invoke the function `malloc` or `calloc`. Whenever the test program allocates new memory using `malloc` or `calloc` at program location k, we increment the value of $dsf(k)$ by the number of bytes allocated. At the end of test execution, the value of $dsf(k)$ contains the total number of bytes allocated at program location k for all such locations k.

6.3.5.1   Experimental Evaluation

Figure 6.4 shows the results of our experiments with this application on our benchmark programs. We evaluate the domain-specific fuzzing application (`mem`) as well as the baseline (`afl`) on the maximum amount of dynamic memory allocated by generated inputs after the 24-hour fuzzing runs. The plots show means and standard errors of this metric across 12 repetitions.

The benchmark `libxml` did not seem to perform any input-dependent dynamic memory allocations. On the benchmarks `vorbis`, `libpng`, `libjpeg-turbo` and `boringssl`, our domain-specific fuzzing application generated inputs that allocate 1.5×–120× more memory. For `libpng` our application generated input PNG images whose metadata specified the maximum allowable image dimensions–as per the validation rules hard-coded in the test driver—of 2 million pixels. Even though such PNG files themselves were only about 1KB in size, their processing required over 24MB of dynamically allocated memory. In Section 6.4, we discuss a composite domain-specific fuzzing application that generates PNG images of dimensions smaller than one thousand pixels, but whose processing required over 2GB of dynamic memory allocation from `libpng`.

Just like with `slow` and `perf` (ref. Sections 6.3.3 and 6.3.4 respectively), the `mem` application was not effective on `libarchive`. Recall that this is the only benchmark in our suite where the initial seed input leads to an early exit due to a validation error.

6.3.6   `valid`: Validity Fuzzing

A major problem associated with CGF is that most randomly generated inputs are invalid; that is, they cause the test program to exit early with an error state. For example,

Table 6.5: `valid`: Application for validity fuzzing

Domain d: $K = \mathbb{L}, V = \mathbb{N}, A = 2^{\mathbb{N}}, a_0 = \emptyset, a \rhd v = a \cup \log_2(v)$	
Hook	**Instrumentation**
$entry\_point()$	**ins_after**('$\forall k \in K : dsf(k) \leftarrow 0$')
$new\_basic\_block()$	$k \leftarrow current\_program\_loc()$
	ins_after('$dsf(k) \leftarrow dsf(k) + 1$')
$fn\_call(name, args)$	if $name = $ '`assume`' :
	$\quad cond \leftarrow args[0]$
	$\quad$**ins_before**('if $cond = $ false then $\forall k \in K : dsf(k) \leftarrow 0$')
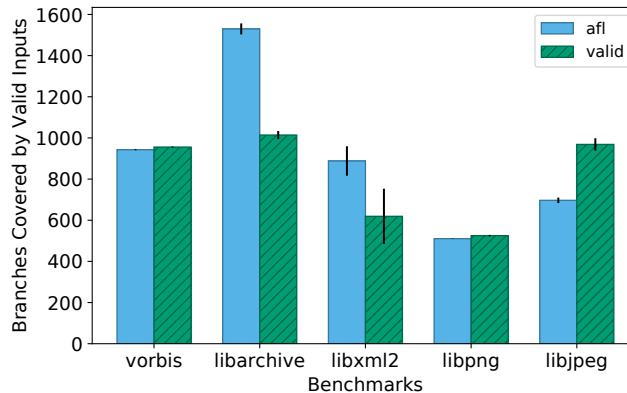
Figure 6.6: Branch coverage among valid inputs, as achieved by inputs generated by baseline (afl) and domain-specific fuzzing application (valid). Higher is better.

traditional CGF on `libpng` is unlikely to generate many valid PNG images, even if fuzzing is seeded with valid inputs to begin with.

Most of the code coverage achieved by the newly generated inputs lies in code paths that deal with input validation and error reporting. Therefore, CGF algorithms struggle to effectively test and find bugs in the main functionalities of such programs.

In many cases, it is desirable to generate *valid* inputs that maximize code coverage. For example, one may want to test programs such as image viewers and media players that download and process files that were uploaded on a social media website. Most likely, such websites do not allow users to upload invalid files. Bugs in the image viewers or media players would then manifest only during the processing of valid files.

Validity fuzzing [142] addresses the problem of generating valid inputs. In validity fuzzing, test programs are augmented to return feedback about whether or not an input is *valid*,

according to some program-specific notion of validity, e.g. whether an input to `libpng` is a valid PNG file. During the fuzzing loop, newly generated inputs are saved either (1) if they increase overall code coverage, or (2) if the newly generated input is valid *and* it covers code that has not been covered by any previously generated *valid* input. The first criterion allows saving intermediate inputs regardless of validity as long as they produce new cumulative code coverage. The hope is that mutating these inputs will lead to more interesting valid inputs being generated later on. The second criterion attempts to maximize code coverage among the valid inputs. Validity fuzzing is equivalent to ZEST's semantic fuzzing technique (Algorithm 3), but without the use of parametric generators. Other researchers have also used notions of program-specific validity to guide the fuzzing search towards generating more valid inputs [101, 150].

We now demonstrate how we implemented the validity fuzzing algorithm in our framework. First, we modified the test drivers that ship with the benchmark suite to add program-specific `assume(expr)` statements. The semantics of `assume` is similar to that of the more familiar `assert`: if the argument `expr` evaluates to `true` at run-time, then the statement is a no-op; otherwise, the test execution is stopped. Figure 6.5 demonstrates one of the three single-line changes we made to the `libpng` test driver. Instead of exiting early due to an invalid PNG header, we simply wrap the validity check with an `assume` statement. We were able to make such small changes in the test drivers of all benchmarks except `boringssl`. Across the five benchmarks whose drivers we modified, we added 1–3 `assume` statements that wrapped existing validity checks in the test drivers, changing 1–11 lines of code. Second, we instrumented the test program to populate the DSF map with information about code coverage during test execution, similar to traditional coverage-guided fuzzing. At runtime, if any of the arguments to `assume` evaluates to `false`, the entire DSF map is reset to the initial state before exiting. Therefore, the DSF map mirrors the traditional code coverage information if and only if the test input is valid. Invalid inputs produce no domain-specific feedback. This scheme leads to the following behavior for Algorithm 4: a newly generated input is saved if either it leads to new cumulative code coverage, or if the input is valid and achieves more code coverage (i.e., changes the aggregate domain-specific feedback) than any other valid input seen so far (i.e., among inputs that produce domain-specific feedback).

Table 6.5 describes the validity fuzzing application more formally. The first line of this table defines the domain. The DSF map for this domain maps program locations (i.e. $K = \mathbb{L}$) to execution counts (i.e. $V = \mathbb{N}$), similar to the `perf` application (ref. Section 6.3.4). However, when aggregating domain-specific feedback, the validity fuzzing application collects a *set of orders of magnitude* of the execution counts for each basic block (i.e. $A = 2^{\mathbb{N}}$). This mirrors the heuristics used by AFL in collecting code coverage [197]. The aggregation is defined by the reduce operator: $a \triangleright v = a \cup \log_2(v)$, where $\log_2(v)$ extracts the position of the highest set bit in the value v extracted from the DSF map. The initial value is the empty set: $a_0 = \emptyset$. Such information allows for differentiation between inputs that execute the same code fragment, say, 2 times versus 4 times (since these counts have different orders of magnitude), but not, say, 10 times versus 11 times (since these counts have the same order of magnitude). The actions described for hooks *entry_point* and *new_basic_block*

in Table 6.5 are exactly the same as those for the `perf` application (Table 6.3). The hook for *fn_call* handles calls to `assume()`. The instrumentation inserts code that performs the required logic: if the argument to `assume` evaluates to `false`, then clear all entries in the DSF map before calling `assume`, which stops the test.

6.3.6.1 Experimental Evaluation

Figure 6.6 contains the results of our experiments with this application on our benchmark programs. We evaluate the domain-specific fuzzing application (`valid`) as well as the baseline (`afl`) on the branch coverage achieved by valid inputs after the 24 hour fuzzing runs. Branch coverage is computed using `gcov` [177]. The plots show means and standard errors of branch coverage across 12 repetitions.

The experiments show that validity fuzzing enables improvement in branch coverage among valid inputs for `libpng` (3%) and `libjpeg-turbo` (39%). For `vorbis`, validity feedback did not appear to have any impact. For `libxml`, the validity fuzzing algorithm produced 30% less branch coverage among valid inputs. Unlike the other benchmarks, which process binary input data, `libxml` expects valid inputs to conform to a context-free grammar. For such a domain, validity fuzzing by itself does not appear to be sufficient. Intuitively, mutating valid XML files using byte-level mutations does not necessarily help produce more valid XML files with diverse code coverage. On `libarchive`, as usual, the domain-specific fuzzing application is not very effective. Since `libarchive` is seeded with an invalid input, most of the inputs generated during the first few hours of fuzzing lead to assumption failures. Naturally, the validity fuzzing algorithm relies on having some valid inputs to begin with in order for its domain-specific feedback to be useful.

6.3.7 `cmp`: Smoothing Hard Comparisons

We next describe a novel solution to a well-known problem, that of hard comparisons. Recall the motivating example in Figure 6.1, which required generating inputs `a` and `b` that were equal to each other. For CGF, similar obstacles arise when encountering operations such as `strncmp`, `memcmp`, and `switch-case` statements. The problem of hard comparisons has been addressed by several researchers in the past [102, 178, 155, 109, 147, 195]. Common solutions to this problem include, but are not limited to: (1) starting with seed inputs that already satisfy most of the complex invariants, (2) mining magic constants—such as `0x0123`—from the test program and then randomly inserting these values as part of the mutation process, (3) transforming the test program to "unroll" an n-byte comparison into a sequence of branches performing 1-byte comparisons, and (4) performing sophisticated static analysis, dynamic taint analysis, or symbolic execution to identify and overcome hard comparisons. Some solutions, such as statically mining magic constants or unrolling multi-byte comparisons, do not work with hard comparisons of variable-length arguments, e.g. `memcmp(a, b, n)`, where all operands are derived from the program input.

Table 6.6: `cmp`: Application for smoothing hard comparisons

| **Domain** d: $K = \mathbb{L}, V = \mathbb{N}, A = \mathbb{N}, a_0 = 0,\ a \rhd v = \max(a, v)$	
Hook	**Instrumentation**
$entry\_point()$	**ins_after**('$\forall k \in K : dsf(k) \leftarrow 0$')
$binop(type, left, op, right)$	if $op \in \{`==', `!='\}$:
	$\quad k \leftarrow current\_program\_loc(),\ n \leftarrow \mathtt{sizeof}(type)$
	$\quad$**ins_after**('$dsf(k) \leftarrow \max(dsf(k), comm\_bits(left, right, n)$')
$fn\_call(name, args)$	if $name \in \{`\mathtt{memcmp}', `\mathtt{strncmp}', `\mathtt{strncasecmp}'\}$:
	$\quad k \leftarrow current\_program\_loc()$
	$\quad left \leftarrow args[0],\ right \leftarrow args[1],\ n \leftarrow args[2]$
	$\quad$**ins_after**('$dsf(k) \leftarrow \max(dsf(k), comm\_bits(left, right, n)$')
$switch(type, val, cases)$	for $case \in cases$:
	$\quad k \leftarrow target\_program\_loc(case),\ n \leftarrow \mathtt{sizeof}(type)$
	$\quad$**ins_after**('$dsf(k) \leftarrow \max(dsf(k), comm\_bits(val, case, n)$')
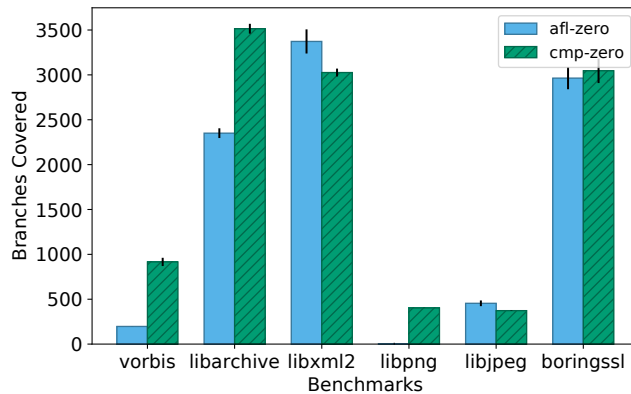
Figure 6.7: Branch coverage, as achieved by inputs generated by baseline (afl-zero) and domain-specific fuzzing application (cmp-zero). The suffix `zero` indicates that seed inputs were simply strings of zeros. Higher is better.

We show how we can prototype a solution for overcoming hard comparisons using FUZ-ZFACTORY. We do not rely on the domain knowledge in seed inputs or on expensive symbolic analysis. Table 6.6 describes our domain-specific fuzzing application. The core idea is to provide domain-specific feedback for each comparison operation in the test program ($K = \mathbb{L}$), where the feedback represents the number of bits $V = \mathbb{N}$ that are common between the two operands being compared. The feedback is aggregated using the *max* reduce operator; therefore, a newly generated input will be saved as a waypoint if it maximizes the number of bits that match at any hard-comparison operation in the program under test. Table 6.6 goes on to describe the program instrumentation strategy. Refer to Table 6.1 for definitions of *binop*, *switch*, *target_program_loc*, and *comm_bits*. The instrumentation strategy is as follows: First, the DSF map is initialized to 0 at the entry point. Then, operations such as integer equality, string comparisons, and `switch-case` statements are instrumented. The inserted code populates the DSF map entries corresponding to their program location with the maximum observed count of common bits between their operands.

6.3.7.1   Experimental Evaluation

Figure 6.7 contains the results of our experiments with this application on the benchmark programs. For this experiment alone, we do not use the initial seed inputs provided in the benchmark suite, but instead seed all fuzzers with an input containing a string of zeros. We do this so that we can study how hard comparisons can be overcome without relying on program-specific knowledge embedded in the seeds. This experiment also simulates a scenario where one wishes to fuzz a program that has an unknown input format, and therefore has no seed inputs available. We evaluate the domain-specific fuzzing application (`cmp-zero`) as well as the baseline (`afl-zero`) on the branch coverage (as computed by `gcov`) achieved by inputs after the 24 hour fuzzing runs. The suffixes `zero` indicate that these experiments did not use meaningful seed inputs. The plots show means and standard errors of branch coverage across 12 repetitions.

From the figure, we see that `cmp-zero` achieves higher code coverage than the baseline in four benchmarks: `vorbis`, `libarchive`, `libpng`, and `boringssl`. Manual investigation revealed that these programs expected their inputs to either contain magic values or to satisfy strict invariants that required hard comparisons. On `vorbis`, the `cmp` front-end achieved 5× more code coverage. On `libpng`, the baseline (`afl-zero`) performed particularly poorly, since the PNG image format requires an 8-byte magic value at the beginning of every input file; the test program exits early if this magic value is not found. The `cmp` front-end effortlessly surpassed this hard comparison and was able to cover over 100× more branches. On `libxml` and `libjpeg-turbo`, the `cmp` front-end does not appear to be useful. In these benchmarks, we did not find any input-dependent hard comparisons between operands larger than two bytes in size. Thus, the baseline approach was sufficient.

Table 6.7: `diff`: Application for incremental fuzzing

Domain d: $K = \mathbb{L} \times \mathbb{L}, V = \mathbb{N}, A = 2^{\mathbb{N}}, a_0 = \emptyset, \; a \triangleright v = a \cup \log_2(v)$	
Hook	**Instrumentation**
$entry\_point()$	$c \leftarrow current\_program\_loc()$
	ins_after('$\forall k \in K : dsf(k) \leftarrow 0$')
	ins_after('$hits\_diff \leftarrow$ false')
	ins_after('$p \leftarrow c$')
$new\_basic\_block()$	$c \leftarrow current\_program\_loc()$
	if $within\_diff(c)$:
	ins_after('$hits\_diff \leftarrow$ true')
	ins_after('if $hits\_diff$ then $dsf(\langle p, c \rangle) \leftarrow dsf(\langle p, c \rangle) + 1$')
	ins_after('$p \leftarrow c$')

```
1  int foo(int a, int b) {
2    int d = a;
3    if ((a + b) % 2) {
4  -    d = 2 * a;
4  +    d = 2 - a;
5    }
6    if (a % 3 && a > 0) {
7      return b/d;
8    } else {
9      return 0;
10   }
11 }
```

(a) Program with a diff: the `*` in Line 4 is modified to a `-`.

Input	**Execution Path**
i_1 : a=3,b=4	$\langle 2,4 \rangle$, ⚡, $\langle 4,6 \rangle$, $\langle 6,9 \rangle$
i_2 : a=4,b=4	$\langle 2,6 \rangle$, $\langle 6,7 \rangle$
i_3 : a=4,b=3	$\langle 2,4 \rangle$ ⚡, $\langle 4,6 \rangle$, $\langle 6,7 \rangle$

(b) Inputs and their execution paths through the program in Figure 6.8. $\langle x, y \rangle$ designates an executed edge between x and y, and ⚡ the hitting of a diff. $\langle x, y \rangle$ highlights the first time an input exercises $\langle x, y \rangle$ *after* hitting the diff during execution.

Figure 6.8: Example motivating new post-diff edge as DSF for incremental (diff) fuzzing.

6.3.8 `diff`: Incremental Fuzzing

We now describe another novel application of FUZZFACTORY: incremental fuzzing after code changes. It is common practice to let fuzzing tools run for many hours or days in order to find bugs in stable versions of complex software. However, if a developer makes a change to such software, there is currently no straightforward way for them to *quickly* fuzz test their changes. They could use the test corpus generated by the long-running fuzzing session on the previous version of the software as a regression test suite, but those inputs may not exercise code paths affected by the changes to the software. They could also start a new fuzzing session with the previously generated corpus of inputs as the initial seeds. However, they
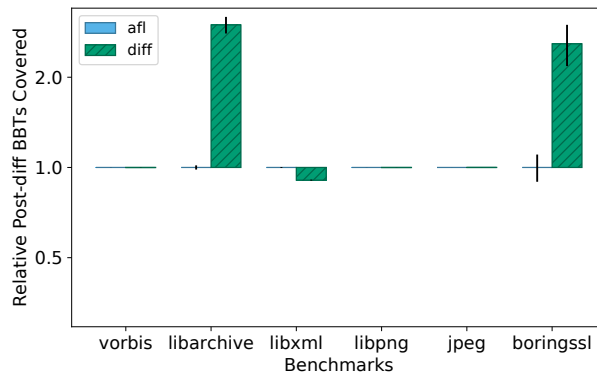
Figure 6.9: Relative coverage of edges after five minutes of incremental fuzzing with the domain-specific `diff` front-end. The baseline is the average coverage achieved by `afl`.

have no way to communicate to the fuzzing engine that it should focus on the code paths affected the changes to the software. Directed fuzzing tools such as AFLGo [23] address this application, but can require several hours of static analysis to pre-compute distances to target program locations[3]. Such approaches may not be practical for use in continuous integration environments where a developer wishes to perform quick regression tests after every code change.

To this end, we propose and implement a domain-specific fuzzing application for incremental fuzzing. The goal of this application is to guide fuzzing towards *quickly* discovering interesting code paths that visit the lines of code that have just been modified. We refer to the set of modified lines of code as the *diff*. To measure the variety of paths executed by the inputs, we will focus on *edges* in the control-flow graph rather than basic blocks alone.

Consider the example program given in Figure 6.8a. This program performs a division at Line 7. In the original program, the divisor `d` was always a multiple of the input `a`, so the division at Line 7 was always safe. Unfortunately, the new change to the program, which switches `2 * a` to `2 - a` in Line 4, makes a division by zero possible. Figure 6.8b shows some inputs and the execution paths they take through this program. The execution path is represented as the sequence of edges executed by the input. We use $\langle x, y \rangle$ to represent the transition from the basic block starting at line x to the basic block starting at line y. We represent the execution of a diff-affected basic block with the symbol ⚡.

Consider the three inputs in Figure 6.8b. Input i_1 (`a=3,b=4`) exercises the diff, but not the division at Line 7. Input i_2 (`a=4,b=4`) exercises the division at Line 7, but not the diff at Line 4. Notice that input i_3 (`a=4,b=3`) does not exercise new edges compared to inputs i_1 and i_2, so regular coverage-guided fuzzing would not save it. However, input i_3 is the first to exercise the true branch leading to Line 7 *after having hit the diff*. We call the edges executed

---

[3]`https://github.com/aflgo/aflgo/issues/21`

after hitting the diff as *post-diff edges*; the newly exercised post-diff edges are highlighted in blue in Figure 6.8b. Since input i_3 covers a new post-diff edge, it is interesting in an incremental fuzzing setting because it exercises a new code path affected by the change in the diff. In fact, it is only one mutation away from `a=2, b=3`, which would trigger a division by zero.

Our FuzzFactory application, `diff`, ensures that input such as i_3 are saved as way-points. It does so by populating the DSF map with the number of times each edge is executed *after* the diff code has been executed (i.e., it must keep track of the edges after the ⚡). For example, for input i_1, the DSF map is $\{\langle 4, 6 \rangle \mapsto 1, \langle 6, 9 \rangle \mapsto 1\}$. For input i_2, the DSF map is $\{\}$ because input i_2 does not hit the diff. Finally, for input i_3, the DSF map is $\{\langle 4, 6 \rangle \mapsto 1, \langle 6, 7 \rangle \mapsto 1\}$.

Table 6.7 formally defines the incremental fuzzing domain and describes the instrumentation. Since we keep track of edges rather than simply basic blocks, $K = \mathbb{L} \times \mathbb{L}$. To better approximate paths, the DSF map collects order-of-magnitude aggregation of edge execution counts, similar to that used for domain `valid` (ref. Section 6.3.6). Thus, $A = 2^{\mathbb{N}}$, $a_0 = \emptyset$, and the reducer function is $a \triangleright v = a \cup \log_2(v)$. To keep track of edges, the instrumentation adds a global variable p to track the location of the *previously visited* basic block. p is combined with the current block c to create the edge tuple $\langle p, c \rangle$. This is inspired by AFL's edge tracking logic [197].

To make sure that we only track *post-diff* edges, the instrumentation also defines a new global variable `hits_diff` in the test program. This variable is set to `false` at the test entry point. At each basic block, the instrumentation adds a check to see whether the basic block is *within_diff*—that is, the basic block was added or modified in the code change of interest—and sets `hits_diff` to `true` if that is the case. Then, the DSF for the edge $\langle p, c \rangle$ is only incremented if `hits_diff` is `true`, effectively counting only post-diff edges.

6.3.8.1 Experimental Evaluation

To simulate the incremental fuzzing environment on our benchmarks without cherry-picking diffs, we perform the following procedure. For each benchmark, we randomly choose one of the saved input directories from our 24-hour runs of AFL on the benchmark. This is our new starting set of test inputs, I. To find a relevant code change, we then advance the code repository by one git commit until we find a diff that (1) affects code in the main test driver, and (2) is exercised by at least one input in I. We keep advancing through the commit history, and accumulate the diffs, until such a diff is found, or until the most recent commit.

To evaluate utility in a continuous integration environment, we run the tools for *five minutes* each. Since we are interested in evaluating the power of the tools to generate inputs with high code coverage downstream from the diff, we logged any input AFL generated that hit the diff in the five minute run. In our coverage evaluation, we augment AFL's regular saved inputs with these.

Figure 6.9 contains the results of our 5-minute incremental fuzzing evaluation. The figure plots means and standard errors of the number of post-diff edges hit by all generated inputs,
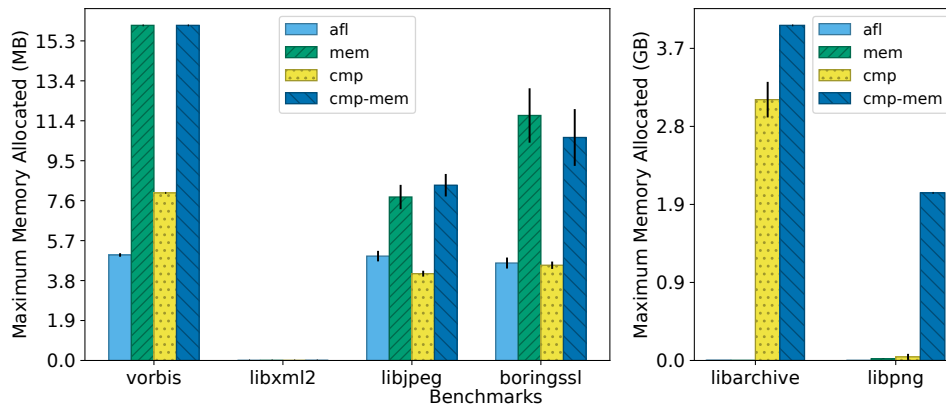
Figure 6.10: Evaluation of composing `cmp` and `mem` into the `cmp-mem` domain. Bars show the maximum dynamic memory allocated—in MB on the left and in GB on the right—at a single program location. Higher is better.

relative to the baseline `afl`. We plot the coverage achieved by our domain-specific fuzzing application, called `diff`, relative to `afl`. For `libpng` and `libjpeg-turbo`, the diffs yielded by our procedure were hit by all inputs in the starting corpus, and for `vorbis`, no inputs in the seed corpus initially hit the diff. This resulted in very large diffs. As expected for such large diffs, `diff` and `afl` were equally successful at finding a variety of post-diff behaviors on these benchmarks. For `libarchive` and `boringssl`, only a few inputs hit the initial diff, and the diff was not very large. These more closely mirrored the incremental changes motivated by our techniques. For these benchmarks, the FUZZFACTORY domain-specific fuzzing application `diff` achieves 2.5-3× more coverage downstream from the diff than `afl`.

6.4 Composing Multiple Domains

Due to the clean separation between domain-specific feedback maps and the underlying fuzzing algorithm, we can easily compose multiple domain-specific fuzzing applications in the same test program binary. Composing two domain-specific fuzzing applications requires no more than incorporating the instrumentation associated with each domain. In our implementation, this is as simple as setting compile-time flags for each domain. Each domain's associated instrumentation only updates its own DSF map. Similarly, our domain-specific fuzzing algorithm aggregates feedback from each registered domain independently (ref. Algorithm 4).

Figure 6.10 shows the results of our experiments with a composition of `cmp` (ref. Section 6.3.7) and `mem` (ref. Section 6.3.5). The goal of this experiment is to maximize memory allocation in the test programs, while also smoothing hard comparisons which may be required to exercise hard-to-reach program branches. This experiment used the initial seed

inputs that ship with the benchmark suite. We compare the composite domain (`cmp-mem`) with the baseline (`afl`) as well each independent application (`cmp` and `mem`). For most benchmarks, the composite application `cmp-mem` generates inputs that allocate more (or equal amounts of) memory than those generated by `cmp` or `mem`. In particular, the combined `cmp-mem` application was able to generate inputs that allocate the maximum memory possible with `libarchive` and `libpng`—4GB and 2GB respectively. For `libarchive`, this result is remarkable because the `mem` domain itself performed much worse than the `afl` baseline, due to the fact that the initial seed inputs were invalid (ref. Section 6.3.5). However, when combined with the application that smooths hard comparisons, it was able to quickly generate valid archive files and eventually generated a LZ4 bomb: a small input that when decoded leads to excessive memory allocation. Similarly, in `libpng`, the `cmp-mem` application was able to generate a PNG bomb. Unlike the most memory-allocating input discovered by `mem` alone, which was an image that declared very large geometric dimensions in its metadata (ref. Section 6.3.5), the PNG bomb generated by `cmp-mem` exploits the decoding of `pCAL`/`sCAL` chunks. Such an input demonstrates a known bug: simply capping an image's geometric dimensions does not limit memory usage when decoding PNG files. We can conclude that a composition of the `cmp` and `mem` domains can perform better than the sum of its parts.

6.4.1   New bugs discovered

Since the benchmark suite used in our experiments contains old, historical versions of heavily fuzzed software, we expected to only find previously known bugs, if any, while fuzzing. To our surprise, we found that the inputs saved by `cmp-mem` when fuzzing the January 2017 snapshot of `libarchive` revealed two *previously unknown* bugs in the latest (March 2019) version: a memory leak[4] and an inadvertent integer sign cast that leads to huge memory allocation[5].

6.5   Discussion

Our framework allows developers and researchers to control the process of fuzz testing by defining a strategy to selectively save intermediate inputs. Our framework does not currently provide any explicit hooks into various other search heuristics used in the CGF algorithm, such as the mutation operators or seed selection strategies. In principle, it should be possible to port general-purpose heuristics such as those used in AFLFast [24] or FairFuzz [108] to work with any of the various domain-specific fuzzing applications described in this chapter. The work on improving general-purpose fuzzing heuristics is orthogonal to this chapter's contributions. Our main contribution is the proposed separation of concerns between the fuzzing algorithm and the choice of feedback from the instrumented program under test.

---

[4]`https://github.com/libarchive/libarchive/issues/1165` and CVE-2019-11463
[5]`https://github.com/libarchive/libarchive/issues/1237`

```
type dsf_t; /* Domain-specific feedback map */

/* Register a new domain. To be invoked once during initialization. */
dsf_t new_domain(int key_size, function reduce, int a_0);

/* Updates to the DSF map. To be invoked during test execution. */
int  dsf_get(dsf_t dsf, int k);              // return dsf[k]
void dsf_set(dsf_t dsf, int k, int v);       // dsf[k] = v
void dsf_increment(dsf_t dsf, int k, int v); // dsf[k] = dsf[k] + v
void dsf_union(dsf_t dsf, int k, int v);     // dsf[k] = dsf[k] | v
void dsf_maximize(dsf_t dsf, int k, int v);  // dsf[k] = max(dsf[k], v)
```

Figure 6.11: API for domain-specific fuzzing in pseudocode.

In theory, a basic increase in code coverage can itself be considered a domain-specific feedback. That is, we could define a domain d where $is\_waypoint(i, \mathcal{S}, d)$ is satisfied when input i leads to the execution of code that is not covered by any input in $\mathcal{S}$. However, in Algorithm 4, we always save an input if it increases code coverage, instead of modeling this criteria through yet another domain. In practice, we found that an increase in code coverage is useful for all domains, since it leads to discovering new program behavior. To put it another way, we always compose every custom domain with a default domain that tries to maximize code coverage. Our implementation allows disabling the default domain via an environment variable if desired.

Recently, even more specialized fuzzers that fit our abstraction of *waypoints* have appeared: e.g. (1) Coppik et al. [42] save inputs that read/write new values to input-dependent memory addresses, and (2) Nilizadeh et al. [130] discover side-channel vulnerabilities by saving inputs whose execution paths maximally differ from a reference path. Such work strengthens the case for FUZZFACTORY.

6.6   Implementation

We have implemented FUZZFACTORY as an extension to AFL and made it publicly available at: `https://github.com/rohanpadhye/FuzzFactory`. In FUZZFACTORY, domain-specific fuzzing applications are implemented by instrumenting test programs. In our applications, we performed instrumentation using LLVM. However, test programs can also be instrumented using any other tool, such as Intel's Pin [115]. In fact, domain-specific fuzzing applications can also be implemented by manually editing test programs to add code that calls the FUZZFACTORY API. We next describe this API.

6.6.1   API for Domain-Specific Fuzzing

Figure 6.11 outlines the API provided by FUZZFACTORY. The type `dsf_t` defines the type of a domain-specific map. In our implementation, the keys and values are always 32-bit

unsigned integers. However, users can specify the size of the DSF map; that is, the number of keys that it will contain.

The API function `new_domain` registers a new domain whose key set K contains `key_size` keys. The arguments `reduce` and `a_0` provide the reducer functions (of type `int x int -> int`) and the initial aggregate value respectively. For the `slow` domain, `key_size` is 1. For applications where K is a set of program locations $\mathbb{L}$, we use `key_size` of 2^{16} and assign 16-bit pseudorandom numbers to basic block locations, similar to AFL. For the incremental fuzzing applications, where $K = \mathbb{L} \times \mathbb{L}$, we use a hash function to combine two basic block locations into a single integer-valued key. The sets V and A are defined implicitly by the usage of DSF maps and the implementation of the `reduce` function. For applications such as validity fuzzing, where A is a set of orders of magnitude, we use bit-vectors to represent sets.

The function `new_domain` returns a handle to the DSF map, which is then used in subsequent APIs listed in Fig. 6.11, such as `dsf_increment`. Calls to the `new_domain` are inserted at test program startup, before any tests are executed. It is up to the user to ensure that the provided reducer function satisfies properties 6.1 and 6.2, which in turn guarantee monotonic aggregation (Theorem 2). API functions that start with '`dsf_`' manipulate the DSF map. The argument `key` must be in the range `[0, key_size)`.

6.7 Summary

In this chapter, we presented FUZZFACTORY, a framework for specifying domain-specific applications using custom feedback collected dynamically during test executions. We described FUZZFACTORY's domain-specific fuzzing algorithm, which incorporates custom feedback as well as user-provided reducer functions to selectively save intermediate inputs, called *waypoints*. We identified key properties that reducer functions must satisfy in order to guarantee that every saved waypoint contributes towards domain-specific progress. We then described the implementation of six domain-specific fuzzing applications implemented using our framework, along with results of our experimental evaluation of these applications on six real-world test programs. We also demonstrated how FUZZFACTORY can be used to compose multiple domain-specific fuzzing applications and empirically show how such compositions can perform better than their constituents. Finally, we described the API provided by our domain-specific fuzzing framework and provided a URL to its publicly available source code.

FUZZFACTORY highlights yet again how a powerful combination of domain expertise and sophisticated feedback-directed fuzzing algorithms can unlock the capability of targeting new automated testing applications.

Chapter 7

Related Work

This chapter discusses related work in automatically finding algorithmic performance issues, coverage-guided fuzzing, automatically generating complex structured test inputs, and customizing fuzzing algorithms.

7.1 Algorithmic Performance Bugs

Crosby and Wallach [45] were the first to demonstrate how algorithmic complexity bugs can be the target of denial-of-service (DoS) attacks. Subsequent work on detecting and preventing DoS attacks [6, 201, 54] has typically focused on measuring aggregate resource exhaustion and does not specifically identify input characteristics that exploit worst-case algorithmic complexity.

Input-sensitive profiling techniques [73, 41, 200] help estimate the algorithmic complexity of a program function empirically by profiling its execution under varying input sizes. However, such techniques require worst-case inputs to already be available.

7.1.1 Redundant Computation Analysis

Redundant data-structure traversals, as targeted by TRAVIOLI (Chapter 3), have been the subject of past study on automatically finding algorithmic performance bugs. Clarity [135] uses static analysis to detect program functions in which an $\mathcal{O}(n)$ traversal occurs $\mathcal{O}(m)$ times redundantly. As TRAVIOLI's' analysis is dynamic, we can determine if repeated traversals are redundant at a finer granularity. For example, if a binary-search-tree is repeatedly queried for different values, we do not report a redundancy if at least two traversals follow different paths in the tree. Clarity conservatively assumes all conditional branches to be equally likely, and thus cannot make such fine-grained distinctions automatically. Clarity therefore uses source-level annotations to recognize operations on standard Java collections that have sub-linear average-time complexity. However, Clarity's static analysis is a sound over-approximation, while our dynamic analysis is subject to false negatives.

Toddler [131] uses dynamic analysis to detect similar memory access patterns at the same execution context. It detects redundancies by analyzing the execution of long-running performance tests and extracting similarities in memory accesses across loop iterations. TRAVIOLI's use of acyclic execution contexts (AECs) and object connectivity allow us to detect traversals from as little as two iterations, and therefore we can detect redundant traversals using unit tests alone. Moreover, AECs enable the detection of *recursive* data-structure traversals, which is not supported by either of these tools.

MemoizeIt [51] uses dynamic analysis to detect functions whose computation can be memoized—this includes a special case of redundant traversals where the repeating subsequences are exactly equal. MemoizeIt can therefore detect the type of bug TRAVIOLI found in `express`, but not the bug we found in `d3-hierarchy`.

7.1.2 Data-Structure Analysis

A number of techniques have been developed to analyze data structures using dynamic analysis. HeapViz [1] summarizes relationships between Java collections to provide a concise visualization of the heap. MG++ [172] generates representations of dynamically evolving data structures. Pheng and Verbrugge [151] measure the number of data structures created and modified over time in Java programs. Laika [44] detects data structures in executing binaries using Bayesian unsupervised learning. DSI [187] identifies pointer-based data structures in C programs. Raman and August [154] detect recursive data structures and profile structural modifications in order to measure their stability.

Similarly, several static analysis techniques aim to discover abstract representations of data structures used in a program, and this body of work usually falls into the category of *shape analysis* [67]. Sophisticated frameworks can be used to prove complex data-structure invariants [161].

In all these techniques, the central theme has been identifying the type of data structures or their representation in program memory, and not on identifying functions that *traverse* these data structures to perform work.

7.1.3 Execution Contexts and AECs

In dynamic analysis, execution indexing [191] allows uniquely identifying a point in a program execution. Such execution indices are too fine-grained for TRAVIOLI. The problem of reasoning about an unbounded number of calling contexts in recursive programs is well-known in the field of static analysis [171, 169]. TRAVIOLI's approach of constructing AECs by removing cycles in execution contexts is similar to the approach employed by Whaley and Lam [186] for context-sensitive pointer analysis, where connected components in the call graph are collapsed to a single node. A subtle difference is that we retain the sequence of functions on paths from the entry of a connected component to its exit; therefore, the resulting AEC is a valid sequence of call sites that can be used for stack-trace debugging.

7.1.4 Worst-Case Execution Time

Researchers in the real-time and embedded systems community have developed methods to estimate Worst-Case Execution Time (WCET) or to prove that a program's WCET does not exceed specified bounds [10, 62, 20, 188, 81]. However, many of these methods require knowledge of loop bounds in the form of manually provided annotations or programming language restrictions. These methods do not easily apply to arbitrary programs written in languages such as C or JavaScript. Further, these methods do not generate the concrete inputs that demonstrate worst-case behavior.

7.1.5 Generating Pathological Inputs Automatically

SlowFuzz [149] uses feedback-directed fuzz testing to automatically generate inputs that can exacerbate algorithmic complexity vulnerabilities by saving intermediate inputs that increase total execution path length. Such a greedy optimization algorithm makes SlowFuzz susceptible to getting stuck in local maxima. PERFFUZZ (Chapter 4) easily outperforms SlowFuzz in finding performance bottlenecks, likely due to its multi-dimensional optimization.

Other tools similar to PERFFUZZ include FOREPOST [80, 116] and GA-Prof [170]. These automatically discover inputs that reveal performance bottlenecks in software, using repeated executions of the test program with candidate inputs. FOREPOST learns rules to select a subset of inputs from a known input space (e.g. a database of records) using unsupervised learning. GA-Prof employs a genetic algorithm, where highly-structured inputs (such as a set of URLs in a transaction) are encoded as genes. In contrast, PERFFUZZ requires no domain knowledge since its inputs are represented as byte sequences. PERFFUZZ's coverage-guided feedback allows it to automatically discover variety in the input space in order to explore deep program functionality.

Search-based software testing (SBST) [122, 83, 82, 194] leverages optimization techniques such as hill climbing to optimize an objective function. These techniques work well when the objective function is a smooth function of the input characteristics.

WISE [27] uses dynamic symbolic execution (ref. Section 2.3) to generate inputs that exercise worst-case behavior. This requires an exhaustive search of all program paths to find the longest path up to a bounded input length. Thus, WISE does not scale to large complex programs. PerfPlotter [33] addresses this concern by probabilistically selecting paths to explore, using heuristics to find best-case and worst-case execution paths. Zhang et al. [202] automatically generate load tests using mixed symbolic execution and iterative-deepening beam search. These tools are designed to maximize a single-dimensional objective function (e.g., total path length, total memory consumption).

SpeedGun [153] automatically generates multi-threaded performance regression tests that find bottlenecks due to synchronization. SpeedGun's input space is quite different, as it generates sequences of method calls in a Java class. On the other hand, PERFFUZZ does not specifically handle concurrent programs. PerfSyn [182] mutates Java programs to expose bottlenecks in a particular method.

7.2   Coverage-Guided Fuzzing

A large body of research exists on improving the heuristics of coverage-guided fuzzing (CGF) tools such as AFL [196]. AFLFast [24] is an extension of AFL that models the fuzzing process as a Markov chain in order to optimize seed selection (equivalent to tuning FUZZPROB in Algorithm 1). FairFuzz [108] modifies AFL to bias input generation towards branches that are rarely executed. AFLGo [23] directs fuzzing towards program points of interest using static analysis. Fuzzers such as VUzzer [155], Steelix [109], Angora [34], REDQUEEN [13] perform lightweight analysis of program executions to direct mutations such that generated inputs are likely to satisfy non-trivial constraints (e.g. magic bytes and checksums). These techniques mostly focus on fuzzing programs that process process untrusted binary data, such as network protocol implementations and media players. A survey on fuzzing by Manès et al. [118] provides a deeper analysis on these fuzzing tools as well as several others.

In work that is outside the scope of this dissertation, we have applied CGF to test ARM TrustZone-based operating systems. In this work, AFL is used as a module within PARTEMU [84], a platform for performing dynamic analysis of trusted operated systems using full-system emulation. The PARTEMU project helped identify over 40 security vulnerabilities in trusted applications across four major trusted operating systems that are deployed on billions of Android devices.

7.3   Generating Complex Inputs for Testing

ZEST (Chapter 5) addresses the problem of generating *inputs* when given a test driver and an input generator. Randoop [139] and EvoSuite [66] generate JUnit tests for a particular class by incrementally trying and combining sequences of calls. During the generation of sequence of calls, both Randoop and EvoSuite take some form of feedback into account. These tools produce unit tests by directly invoking methods on the component classes.

UDITA [68] allows developers to write random-input generators in a QuickCheck-like language. UDITA then performs *bounded-exhaustive* enumeration of the paths through the generators, along with several optimizations.

Targeted property-testing [113, 112] guides input generators used in property testing towards a user-specified fitness value using techniques such as hill climbing and simulated annealing. GödelTest [61] attempts to satisfy user-specified properties on inputs. It performs a meta-heuristic search for stochastic models that are used to sample random inputs from a generator.

Grammar-based fuzzing [121, 43, 173, 71, 22] techniques rely on context-free grammar specifications to generate structured inputs. CSmith [192] generates random C programs for differential testing of C compilers. LangFuzz [87] generates random programs using a grammar and by recombining code fragments from a codebase. These approaches fall under the category of generator-based testing, but primarily focus on tuning the underlying

generators rather than using code coverage feedback. ZEST is not restricted to context-free grammars, and does not require any domain-specific tuning.

Several CGF tools developed concurrently with ZEST also leverage input format specifications (either grammmars [12, 184], file formats [150], or protocol buffers [168]) to improve the performance of CGF. These tools develop mutations that are specific to the input format specifications, e.g. syntax-tree mutations for grammars. ZEST's generators are arbitrary programs; therefore, we perform mutations directly on the parameters that determine the execution path through the generators, rather than on a parsed representation of inputs.

As a follow-up to the ZEST project, we have explored the use of reinforcement learning to guide structured-input generators using JQF: RLCheck [158] can quickly generate a large number of diverse semantically valid inputs, which may be useful for applications such as regression testing.

7.4   Customizing Fuzzing Algorithms

The JQF (Section 5.3) framework allows users to implement custom fuzzing algorithms for guiding QuickCheck-like generators. In contrast, FUZZFACTORY (Chapter 6) is a framework for implementing domain-specific fuzzing applications by providing custom feedback from program execution. The main difference between these frameworks is the point of customization. In JQF, the instrumentation is fixed, while the search algorithm can be customized. In FUZZFACTORY however, the search algorithm is fixed, while the instrumentation and program feedback can be optimized. In principle, these techniques could be combined together in a framework that allows customizing input generation, program instrumentation, feedback, and search. However, we believe that the separation of concerns is important for preventing a combinatorial blowup of abstractions exposed to an end user. JQF is a great platform for researchers who want to evaluate various fuzzing search heuristics, whereas FUZZFACTORY is appropriate for researchers who wish to retarget fuzzing towards achieving new testing objectives.

The LLVM-based Clang compiler [106] provides a customizable tracing framework for C/C++ programs. With the use of command-line flags such as `-fsanitize-coverage`, one can ask Clang to instrument basic blocks and comparison operations to call specially named functions; users can link-in custom implementations of these functions to trace program execution. LibFuzzer (ref. Section 2.5.1) uses these hooks to provide feedback from a program under test in order to perform coverage-guided fuzzing. However, `libFuzzer` does not provide a mechanism to provide arbitrary domain-specific feedback with custom aggregation functions. That is, while LLVM provides hooks into a program's execution, there is currently no way to communicate information to the fuzzing algorithm. However, it is relatively easy to use LLVM's tracing hooks to call into FUZZFACTORY's API for domain-specific fuzzing.

Chapter 8

Conclusion

This chapter first highlights the key takeaway from this dissertation, and then presents opportunities for future work.

8.1 Key Takeaway

In Chapters 3 and 4, we described algorithms to find *algorithmic performance issues* using developer intent captured implicitly via a simple abstraction: *functional test cases*. Developers need only provide functional tests that demonstrate common uses of their software. Using clever algorithms, TRAVIOLI and PERFFUZZ use these artifacts to automatically find algorithmic complexity issues, via dynamic analysis and coverage-guided fuzzing respectively.

In Chapter 5, we showed how developer intent can be captured via the well-known abstractions of *generator functions* and *validity predicates*. The generators and validity predicates can be written in the same language as the program under test, and therefore do not require developers to learn any new specification or sampling language. For developers, the abstraction appears to only provide means to sample random instances of a type, say X, and then discard any $x \in X$ that do not satisfy the provided validity predicates. Algorithms such as ZEST use feedback from test execution to bias generation towards inputs that are likely to be valid and increase code coverage; this mechanism is transparent to the user. The JQF framework provides the reverse abstraction to fuzzing researchers: given some program feedback in the form of coverage and validity, the *guidance* algorithm must provide the next "input" represented as an infinite sequence of bytes. The fact that test programs use generators and complex predicates is largely hidden from the guidance mechanism. In this way, researchers developing guidances for JQF can utilize a developer's domain knowledge of their program's input structure for free.

In Chapter 6, we presented the abstraction of *waypoints*. We found that many domain-specific fuzzing applications can be effectively represented as a problem of determining which inputs should be saved as waypoints during coverage-guided fuzzing. To specify waypoints, FUZZFACTORY requires users to specify a reducer function and have the test program pop-

ulate domain-specific key-value maps during test execution. Key-value maps are a well known data structure; we believe the API to populate them should be entirely unsurprising to most developers. Reducer functions are commonly used in functional programming languages—they are sometimes called *folds*. In this way, FuzzFactory avoids imposing a burden on developers to learn new specification languages. The separation of concerns between input generation and feedback functions also allows for clean composition of multiple domain-specific fuzzing applications.

The key takeaway from this dissertation can be summarized as follows:

> *Automated testing tools can be made smarter by utilizing artifacts that incorporate the domain expertise of software developers. A good solution combines simple, clean abstractions to capture users' intent with sophisticated algorithms that can use this information to dramatically transform the search space. Such combinations can make challenging testing problems tractable and unlock new bug-finding capabilities.*

8.2 Future Work

The next generation of program analysis tools may have to deal with an increasing reliance on legacy software, rapid micro-deployments of code changes, client-side code validation, and a heterogeneous mix of application domains. There is a continuing need for automated testing tools that can effectively utilize a vast ecosystem of domain-specific data sources as needed.

First, most contemporary bug-finding tools assume that the program under test has never been seen before. In practice, however, critical software such as the GCC compiler have been fuzzed for decades. What can we learn from past fuzzing campaigns? Can we develop adaptive fuzzing approaches that can automatically adjust their heuristics based on previous executions? An application of machine learning techniques on generators may prove useful in isolating interesting input features. For example, we could perhaps learn that most bugs found in GCC relate to use of bit-wise arithmetic in C programs. We could then bias C-program generators towards producing a variety of bit-wise operators.

Second, a related problem is that of regression testing; that is, quickly finding bugs introduced by code changes. Ideally, we would like to test only the parts of the program that are affected by the change. With FuzzFactory, we presented a preliminary solution for this problem (Section 6.3.8), but it relies on starting with previously saved inputs which exercise the modified code. Other proposed approaches require too much time to statically analyze the code changes [23]. It would be great to have human-in-the-loop tools where developers can communicate what input features are desirable to test a particular code component. For example, if a GCC developer updates its vectorizing optimizations, how can they tell their fuzzing tool to generate C programs with lots of nested loops and array accesses?

Third, the use of program instrumentation for receiving feedback during fuzz testing slows down test execution. In turn, this reduces the number of inputs that can be evaluated per

unit time. Some recent work such as UnTracer [127] addresses this problem for conventional coverage-guided fuzzing by dynamically un-instrumenting program points that have already been covered by previous inputs. However, this does not generalize to domain-specific fuzzing applications that capture more information than just code coverage. In this regard, static program analysis may be helpful to identify in advance a small subset of program locations whose instrumentation is most likely to help in achieving domain-specific testing objectives. For example, to reveal performance bugs in GCC, static analysis might identify that it is important to instrument the optimizer but not necessarily the parser.

Finally, the test oracle problem [18] continues to challenge bug-finding tools. For example, TRAVIOLI identifies program points that likely perform redundant computations and PERFFUZZ automatically generates pathological inputs; yet, neither of these techniques can precisely say whether their findings constitute a performance bug that requires a fix. Perhaps a combination of data mined from past bug reports as well as machine learning techniques may be useful here as well. For example, when we find a hot spot in GCC's optimizer that is only triggered in corner cases, can we use the information about the performance characteristics of old versions of GCC or contemporary versions of other C compilers to guess whether or not we have found a bug?

It is clear that there is tremendous opportunity for making significant advances in automated testing by utilizing external data sources and using humans in the loop. We hope this dissertation helps seed further research in the area of effective and usable automated testing tools that can be specialized to a variety of programs domains and test objectives. Our hope is that such research will enable automated testing tools to permeate the mainstream software development lifecycle and strengthen our society's confidence in the quality and security of critical software systems.

Bibliography

[1]   Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. "Heapviz: Interactive Heap Visualization for Program Understanding and Debugging". In: *Proceedings of the 5th International Symposium on Software Visualization*. SOFTVIS '10. 2010, pp. 53–62. ISBN: 978-1-4503-0028-5. DOI: `10.1145/1879211.1879222`. URL: `http://doi.acm.org/10.1145/1879211.1879222`.

[2]   Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. *OSS-Fuzz—Continuous Fuzzing for Open Source Software*. `https://github.com/google/oss-fuzz`. Accessed April 17, 2019. 2016.

[3]   Tim Allison. *#threeCheersForFuzzing*. `https://twitter.com/_tallison/status/1050455776848949249`. Accessed April 17, 2019. 2018.

[4]   Cláudio Amaral, Mário Florido, and Vıtor Santos Costa. "PrologCheck–property-based testing in Prolog". In: *International Symposium on Functional and Logic Programming*. Springer. 2014, pp. 1–17.

[5]   Saswat Anand, Corina S. Păsăreanu, and Willem Visser. "JPF-SE: a symbolic execution extension to Java PathFinder". In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2007.

[6]   João Antunes, Nuno Ferreira Neves, and Paulo Jorge Verıssimo. "Detection and prediction of resource-exhaustion vulnerabilities". In: *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2008, pp. 87–96.

[7]   *Apache Ant*. `https://ant.apache.org`. Accessed August 24, 2018. 2018.

[8]   *Apache Byte Code Engineering Library*. `https://commons.apache.org/proper/commons-bcel`. Accessed August 24, 2018. 2018.

[9]   *Apache Maven*. `https://maven.apache.org`. Accessed August 24, 2018. 2018.

[10] Robert Arnold, Frank Mueller, David Whalley, and Marion Harmon. "Bounding worst-case instruction cache performance". In: *Real-Time Systems Symposium, 1994., Proceedings*. IEEE. 1994, pp. 172–181.

[11] Thomas Arts, John Hughes, Joakim Johansson, and Ulf T. Wiger. "Testing telecoms software with quviq QuickCheck". In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*. 2006, pp. 2–10. DOI: `10.1145/1159789.1159792`. URL: `http://doi.acm.org/10.1145/1159789.1159792`.

[12] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. "Nautilus: Fishing for Deep Bugs with Grammars". In: *26th Annual Network and Distributed System Security Symposium*. NDSS '19. 2019.

[13] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In: *NDSS*. Vol. 19. 2019, pp. 1–15.

[14] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. "Enhancing Symbolic Execution with Veritesting". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. 2014, pp. 1083–1094. ISBN: 978-1-4503-2756-5.

[15] Thomas Ball and James R. Larus. "Efficient Path Profiling". In: *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 29. 1996, pp. 46–57. ISBN: 0-8186-7641-8. URL: `http://dl.acm.org/citation.cfm?id=243846.243857`.

[16] Thomas Ball, Peter Mataga, and Mooly Sagiv. "Edge Profiling Versus Path Profiling: The Showdown". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '98. 1998, pp. 134–148. ISBN: 0-89791-979-3. DOI: `10.1145/268946.268958`. URL: `http://doi.acm.org/10.1145/268946.268958`.

[17] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. "SNOOZE: Toward a Stateful Network Protocol fuzZEr". In: *Proceedings of the 9th International Conference on Information Security*. ISC'06. 2006, pp. 343–358. ISBN: 978-3-540-38341-3. DOI: `10.1007/11836810_25`. URL: `http://dx.doi.org/10.1007/11836810_25`.

[18] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The oracle problem in software testing: A survey". In: *IEEE transactions on software engineering* 41.5 (2014), pp. 507–525.

[19] Clark Barrett and Cesare Tinelli. "Satisfiability modulo theories". In: *Handbook of Model Checking*. Springer, 2018, pp. 305–343.

[20] Guillem Bernat, Antoine Colin, and Stefan M Petters. "WCET analysis of probabilistic hard real-time systems". In: *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*. IEEE. 2002, pp. 279–288.

[21] *beStorm®Software Security*. `https://www.beyondsecurity.com/bestorm.html`. Accessed January 28, 2019. 2019.

[22] Michael Beyene and James H. Andrews. "Generating String Test Data for Code Coverage". In: *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*. 2012, pp. 270–279. DOI: `10.1109/ICST.2012.107`. URL: `https://doi.org/10.1109/ICST.2012.107`.

[23] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed Greybox Fuzzing". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. 2017.

[24] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-based Greybox Fuzzing As Markov Chain". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. 2016.

[25] Ella Bounimova, Patrice Godefroid, and David Molnar. "Billions and billions of constraints: Whitebox fuzz testing in production". In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 122–131.

[26] Melvin A Breuer. "A random and an algorithmic technique for fault detection test generation for sequential circuits". In: *IEEE Transactions on Computers* 100.11 (1971), pp. 1364–1370.

[27] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. "WISE: Automated Test Generation for Worst-Case Complexity". In: *Proceedings of the 31st International Conference on Software Engineering*. 2009.

[28] Mathias Bynens. *In search of the perfect URL validation regex.* `https://mathiasbynens.be/demo/url-regex`. 2014.

[29] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. 2008.

[30] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. "EXE: Automatically Generating Inputs of Death". In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. CCS '06. Alexandria, Virginia, USA: Association for Computing Machinery, 2006, pp. 322–335. ISBN: 1595935185. DOI: `10.1145/1180405.1180445`. URL: `https://doi.org/10.1145/1180405.1180445`.

[31] Cristian Cadar and Koushik Sen. "Symbolic execution for software testing: three decades later". In: *Communications of the ACM* 56.2 (2013), pp. 82–90.

[32] *cargo fuzz.* `https://github.com/rust-fuzz/cargo-fuzz`. Accessed June 30, 2020. 2019.

[33] Bihuan Chen, Yang Liu, and Wei Le. "Generating Performance Distributions via Probabilistic Symbolic Execution". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. 2016, pp. 49–60. ISBN: 978-1-4503-3900-1. DOI: `10.1145/2884781.2884794`. URL: `http://doi.acm.org/10.1145/2884781.2884794`.

[34] Peng Chen and Hao Chen. "Angora: Efficient Fuzzing by Principled Search". In: *Proceedings of the 39th IEEE Symposium on Security and Privacy*. 2018.

[35] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. "EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers". In: *28th USENIX Security Symposium (USENIX Security 19)*. Aug. 2019. URL: `https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang`.

[36] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "The S2E Platform: Design, Implementation, and Applications". In: *ACM Transactions on Computer Systems*. 30.1 (2012), p. 2.

[37] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*. ICFP. 2000.

[38] Lori A. Clarke. "A program testing system". In: *Proc. of the 1976 annual conference*. 1976, pp. 488–491.

[39] *Codenomicon Vulnerability Management*. `http://www.codenomicon.com/index.html`. Accessed January 28, 2019. 2019.

[40] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. "Improving your software using static analysis to find bugs". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. 2006, pp. 673–674.

[41] Emilio Coppa, Camil Demetrescu, and Irene Finocchi. "Input-Sensitive Profiling". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. 2012, pp. 89–98. ISBN: 978-1-4503-1205-9. DOI: `10.1145/2254064.2254076`. URL: `http://doi.acm.org/10.1145/2254064.2254076`.

[42] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. "MemFuzz: Using Memory Accesses to Guide Fuzzing". In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, pp. 48–58.

[43] David Coppit and Jiexin Lian. "Yagg: An Easy-to-use Generator for Structured Test Inputs". In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. 2005, pp. 356–359. ISBN: 1-58113-993-4. DOI: `10.1145/1101908.1101969`. URL: `http://doi.acm.org/10.1145/1101908.1101969`.

[44] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. "Digging for Data Structures". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. 2008, pp. 255–266. URL: `http://dl.acm.org/citation.cfm?id=1855741.1855759`.

[45] Scott A. Crosby and Dan S. Wallach. "Denial of Service via Algorithmic Complexity Attacks". In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM'03. 2003, pp. 3–3. URL: `http://dl.acm.org/citation.cfm?id=1251353.1251356`.

[46] *CyberFlood - Spirent*. `https://www.spirent.com/products/cyberflood`. Accessed January 28, 2019. 2019.

[47] *d3-collection: Handy data structures for elements keyed by string*. `https://github.com/d3/d3-collection`. Retrieved: August 2016.

[48] *d3-hierarchy: 2D layout algorithms for visualizing hierarchical data*. `https://github.com/d3/d3-hierarchy`. Retrieved: August 2016.

[49] *D3: a JavaScript library for visualizing data with HTML, SVG, and CSS*. `https://d3js.org`. Retrieved: August 2016.

[50] René David and Pascale Thévenod-Fosse. "Random testing of integrated circuits". In: *IEEE Transactions on Instrumentation and Measurement* 1 (1981), pp. 20–25.

[51] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. "Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. 2015, pp. 607–622. ISBN: 978-1-4503-3689-5. DOI: `10.1145/2814270.2814290`. URL: `http://doi.acm.org/10.1145/2814270.2814290`.

[52] Isil Dillig, Thomas Dillig, and Alex Aiken. "Static error detection using semantic inconsistency inference". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007, pp. 435–445.

[53] Joe W Duran and Simeon C Ntafos. "An evaluation of random testing". In: *IEEE transactions on Software Engineering* 4 (1984), pp. 438–444.

[54] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. "Radmin: early detection of application-level resource exhaustion and starvation attacks". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2015, pp. 515–537.

[55] Roy Emek, Itai Jaeger, Yehuda Naveh, Gadi Bergman, Guy Aloni, Yoav Katz, Monica Farkash, Igor Dozoretz, and Alex Goldin. "X-Gen: A random test-case generator for systems and SoCs". In: *High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International*. IEEE. 2002, pp. 145–150.

[56] Dawson Engler and Ken Ashcraft. "RacerX: effective, static detection of race conditions and deadlocks". In: *ACM SIGOPS operating systems review* 37.5 (2003), pp. 237–252.

[57] Dawson Engler and Madanlal Musuvathi. "Static analysis versus software model checking for bug finding". In: *International Workshop on Verification, Model Checking, and Abstract Interpretation.* Springer. 2004, pp. 191–210.

[58] *Eris: Porting of QuickCheck to PHP.* `https://github.com/giorgiosironi/eris`. Accessed January 28, 2019. 2019.

[59] *express.js: Fast, unopinionated, minimalist web framework for node.* `https://github.com/expressjs/express`. Retrieved: August 2016.

[60] Facebook. *Infer Static Analyzer.* `https://fbinfer.com/`. Retrieved: June 2020.

[61] Robert Feldt and Simon Poulding. "Finding test data with specific properties via metaheuristic search". In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE).* IEEE. 2013, pp. 350–359.

[62] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. "Reliable and precise WCET determination for a real-life processor". In: *International Workshop on Embedded Software.* Springer. 2001, pp. 469–485.

[63] Justin E Forrester and Barton P Miller. "An empirical study of the robustness of Windows NT applications using random testing". In:

[64] Pascale Fosse and René David. "Random testing of memories". In: *GI—7. Jahrestagung.* Springer, 1977, pp. 139–153.

[65] Gordon Fraser and Andrea Arcuri. "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite". In: *ACM Trans. Softw. Eng. Methodol.* 24.2 (Dec. 2014), 8:1–8:42. ISSN: 1049-331X. DOI: `10.1145/2685612`. URL: `http://doi.acm.org/10.1145/2685612`.

[66] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-oriented Software". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.* ESEC/FSE '11. 2011.

[67] Rakesh Ghiya and Laurie J. Hendren. "Is It a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-directed Pointers in C". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* POPL '96. 1996, pp. 1–15. ISBN: 0-89791-769-3. DOI: `10.1145/237721.237724`. URL: `http://doi.acm.org/10.1145/237721.237724`.

[68] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. "Test generation through programming in UDITA". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010.* 2010, pp. 225–234. DOI: `10.1145/1806799.1806835`. URL: `http://doi.acm.org/10.1145/1806799.1806835`.

[69]   *Global Software Testing Market 2017–2021.* `https://www.technavio.com/report/software-testing-services-market-size-industry-analysis`. Retrieved: June 2020.

[70]   Patrice Godefroid. "Fuzzing: Hack, art, and science". In: *Communications of the ACM* 63.2 (2020), pp. 70–76.

[71]   Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. "Grammar-based Whitebox Fuzzing". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '08. 2008.

[72]   Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '05. 2005.

[73]   Simon F. Goldsmith, Alex S. Aiken, and Daniel S. Wilkerson. "Measuring Empirical Computational Complexity". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering.* ESEC-FSE '07. 2007, pp. 395–404. ISBN: 978-1-59593-811-4. DOI: `10.1145/1287624.1287681`. URL: `http://doi.acm.org/10.1145/1287624.1287681`.

[74]   Google. *ClusterFuzz - README.* `https://github.com/google/clusterfuzz/blob/2ae06a430c6f9bfcf418490f3416f28a94d51515/README.md`. Retrieved: June 2020.

[75]   Google. *Continuous fuzzing of open source software.* `https://opensource.google.com/projects/oss-fuzz`. Accessed March 26, 2019. 2019.

[76]   Google. *Error Prone.* `https://errorprone.info/`. Retrieved: June 2020.

[77]   Google. *Set of tests for fuzzing engines.* `https://github.com/google/fuzzer-test-suite`. Accessed March 20, 2019. 2019.

[78]   *Google Closure.* `https://developers.google.com/closure/compiler`. Accessed August 24, 2018. 2018.

[79]   Susan L Graham, Peter B Kessler, and Marshall K Mckusick. "Gprof: A call graph execution profiler". In: *ACM Sigplan Notices.* Vol. 17. 6. ACM. 1982, pp. 120–126.

[80]   Mark Grechanik, Chen Fu, and Qing Xie. "Automatically Finding Performance Problems with Feedback-directed Learning Software Testing". In: *Proceedings of the 34th International Conference on Software Engineering.* ICSE '12. 2012, pp. 156–166. ISBN: 978-1-4673-1067-3. URL: `http://dl.acm.org/citation.cfm?id=2337223.2337242`.

[81]   Bogdan Groza and Marius Minea. "Formal modelling and automatic detection of resource exhaustion attacks". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* ACM. 2011, pp. 326–333.

[82]   Mark Harman. "The current state and future of search based software engineering". In: *2007 Future of Software Engineering.* IEEE Computer Society. 2007, pp. 342–357.

[83] Mark Harman and Bryan F Jones. "Search-based software engineering". In: *Information and software Technology* 43.14 (2001), pp. 833–839.

[84] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, Michael Grace, Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, Hayawardh Vijayakumar, et al. "Partemu: Enabling dynamic analysis of real-world trustzone software using emulation". In: *Proceedings of the 29th USENIX Security Symposium*. USENIX Security 2020. 2020.

[85] Sam Hocevar. *zzuf.* `http://caca.zoy.org/wiki/zzuf`. Accessed Jan 2018. 2007.

[86] Marc R Hoffmann, B Janiczak, and E Mandrikov. *Eclemma-jacoco java code coverage library.* 2011.

[87] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments". In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. 2012.

[88] Paul Holser. *junit-quickcheck: Property-based testing, JUnit-style.* `https://pholser.github.io/junit-quickcheck`. Accessed January 11, 2019. 2014.

[89] *Hypothesis for Python.* `https://hypothesis.works/`. Accessed January 28, 2019. 2019.

[90] *immutable.js: Immutable persistent data collections for Javascript.* `https://github.com/facebook/immutable-js`. Retrieved: August 2016.

[91] Darrel C. Ince. "The automatic generation of test data". In: *The Computer Journal* 30.1 (1987), pp. 63–69.

[92] Ranjit Jhala and Rupak Majumdar. "Software model checking". In: *ACM Computing Surveys (CSUR)* 41.4 (2009), pp. 1–54.

[93] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. "Understanding and Detecting Real-world Performance Bugs". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. 2012, pp. 77–88. ISBN: 978-1-4503-1205-9. DOI: `10.1145/2254064.2254075`. URL: `http://doi.acm.org/10.1145/2254064.2254075`.

[94] William Johansson, Martin Svensson, Ulf E Larson, Magnus Almgren, and Vincenzo Gulisano. "T-Fuzz: Model-based fuzzing for robustness testing of telecommunication protocols". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE. 2014, pp. 323–332.

[95] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "Why don't software developers use static analysis tools to find bugs?" In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 672–681.

[96] *JSVerify: Property-based testing for JavaScript.* `https://github.com/jsverify/jsverify`. Accessed January 28, 2019. 2019.

[97] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. "Fast and accurate static data-race detection for concurrent programs". In: *International Conference on Computer Aided Verification*. Springer. 2007, pp. 226–239.

[98] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. "Heap Reference Analysis Using Access Graphs". In: *ACM Trans. Program. Lang. Syst.* 30.1 (Nov. 2007). ISSN: 0164-0925. DOI: `10.1145/1290520.1290521`. URL: `http://doi.acm.org/10.1145/1290520.1290521`.

[99] James C. King. "Symbolic execution and program testing". In: *Commun. ACM* 19 (7 July 1976), pp. 385–394. ISSN: 0001-0782.

[100] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. "Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. 2018, pp. 2123–2138. ISBN: 978-1-4503-5693-0. DOI: `10.1145/3243734.3243804`. URL: `http://doi.acm.org/10.1145/3243734.3243804`.

[101] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. "RFUZZ: Coverage-directed Fuzz Testing of RTL on FPGAs". In: *Proceedings of the International Conference on Computer-Aided Design*. ICCAD '18. 2018, 28:1–28:8. ISBN: 978-1-4503-5950-4. DOI: `10.1145/3240765.3240842`. URL: `http://doi.acm.org/10.1145/3240765.3240842`.

[102] LafIntel. *Circumventing Fuzzing Roadblocks with Compiler Transformations*. `https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/`. Accessed March 20, 2019. 2016.

[103] Leonidas Lampropoulos and Konstantinos Sagonas. "Automatic WSDL-guided Test Case Generation for PropEr Testing of Web Services". In: *Proceedings 8th International Workshop on Automated Specification and Verification of Web Systems, WWV 2012, Stockholm, Sweden, 16th July 2012*. 2012, pp. 3–16. DOI: `10.4204/EPTCS.98.3`. URL: `https://doi.org/10.4204/EPTCS.98.3`.

[104] Axel van Lamsweerde. "Formal specification: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. 2000, pp. 147–159.

[105] David Larochelle and David Evans. "Statically detecting likely buffer overflow vulnerabilities". In: *10th USENIX Security Symposium*. 2001.

[106] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. 2004, pp. 75–. ISBN: 0-7695-2102-9. URL: `http://dl.acm.org/citation.cfm?id=977395.977673`.

[107] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. "PerfFuzz: Automatically Generating Pathological Inputs". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. 2018, pp. 254–265. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213874. URL: http://doi.acm.org/10.1145/3213846.3213874.

[108] Caroline Lemieux and Koushik Sen. "FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE '18. 2018.

[109] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. "Steelix: Program-state Based Binary Fuzzing". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. 2017.

[110] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. 1st. 2014. ISBN: 9780133905908.

[111] LLVM Developer Group. *libFuzzer*. http://llvm.org/docs/LibFuzzer.html. Accessed March 20, 2019. 2016.

[112] A. Loscher and K. Sagonas. "Automating Targeted Property-Based Testing". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. Vol. 00. Apr. 2018, pp. 70–80. DOI: 10.1109/ICST.2018.00017. URL: doi.ieeecomputersociety.org/10.1109/ICST.2018.00017.

[113] Andreas Löscher and Konstantinos Sagonas. "Targeted Property-based Testing". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. 2017, pp. 46–56. ISBN: 978-1-4503-5076-1. DOI: 10.1145/3092703.3092711. URL: http://doi.acm.org/10.1145/3092703.3092711.

[114] Ada Augusta Lovelace. "Sketch of The Analytical Engine invented by Charles Babbage, by L. F. Menabrea of Turin, Officer of the Military Engineers, with notes upon the Memoir by the Translator". In: *Taylor's Scientific Memoirs* 3 (1842), pp. 666–731.

[115] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034. URL: http://doi.acm.org/10.1145/1065010.1065034.

[116] Qi Luo, Denys Poshyvanyk, Aswathy Nair, and Mark Grechanik. "FOREPOST: A Tool for Detecting Performance Problems with Feedback-driven Learning Software Testing". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE '16. 2016, pp. 593–596. ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889164. URL: http://doi.acm.org/10.1145/2889160.2889164.

[117] Ravichandhran Madhavan and Raghavan Komondoor. "Null Dereference Verification via Over-Approximated Weakest Pre-Conditions Analysis". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA: Association for Computing Machinery, 2011, pp. 1033–1052. ISBN: 9781450309400. DOI: `10.1145/2048066.2048144`. URL: `https://doi.org/10.1145/2048066.2048144`.

[118] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. "The art, science, and engineering of fuzzing: A survey". In: *IEEE Transactions on Software Engineering* (2019).

[119] *Math.js: An extensive math library for JavaScript and Node.js.* `https://github.com/josdejong/mathjs`. Retrieved: August 2016.

[120] Peter M Maurer. "Design verification of the WE 32106 math accelerator unit". In: *IEEE Design & Test of Computers* 5.3 (1988), pp. 11–21.

[121] Peter M. Maurer. "Generating test data with enhanced context-free grammars". In: *Ieee Software* 7.4 (1990), pp. 50–55.

[122] Phil McMinn. "Search-Based Software Testing: Past, Present and Future". In: *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. ICSTW '11. 2011, pp. 153–163. ISBN: 978-0-7695-4345-1. DOI: `10.1109/ICSTW.2011.100`. URL: `http://dx.doi.org/10.1109/ICSTW.2011.100`.

[123] Barton P. Miller, Louis Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: `10.1145/96267.96279`.

[124] MITRE. *2019 CWE Top 25 Most Dangerous Software Errors.* `https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html`. Retrieved: June 2020.

[125] *Mozilla Rhino.* `https://github.com/mozilla/rhino`. Accessed August 24, 2018. 2018.

[126] Glenford J. Myers. *Art of Software Testing.* Wiley, 1979. ISBN: 0471043281.

[127] Stefan Nagy and Matthew Hicks. "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 787–802.

[128] M. G. Nanda and S. Sinha. "Accurate Interprocedural Null-Dereference Analysis for Java". In: *2009 IEEE 31st International Conference on Software Engineering*. 2009, pp. 133–143.

[129] Nicholas Nethercote and Julian Seward. "Valgrind: A program supervision framework". In: *Electronic notes in theoretical computer science* 89.2 (2003), pp. 44–66.

[130] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. "DifFuzz: Differential Fuzzing for Side-channel Analysis". In: *Proceedings of the 41st International Conference on Software Engineering.* ICSE '19. 2019, pp. 176–187. DOI: `10.1109/ICSE.2019.00034`. URL: `https://doi.org/10.1109/ICSE.2019.00034`.

[131] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. "Toddler: Detecting Performance Problems via Similar Memory-access Patterns". In: *Proceedings of the 2013 International Conference on Software Engineering.* ICSE '13. 2013, pp. 562–571. ISBN: 978-1-4673-3076-3. URL: `http://dl.acm.org/citation.cfm?id=2486788.2486862`.

[132] *Node.js.* `https://nodejs.org`. Retrieved: August 2016.

[133] Bashar Nuseibeh and Steve Easterbrook. "Requirements engineering: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering.* 2000, pp. 35–46.

[134] Robert O'callahan and Jong-Deok Choi. "Hybrid dynamic data race detection". In: *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming.* 2003, pp. 167–178.

[135] Oswaldo Olivo, Isil Dillig, and Calvin Lin. "Static Detection of Asymptotic Performance Bugs in Collection Traversals". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2015. 2015, pp. 369–378. ISBN: 978-1-4503-3468-6. DOI: `10.1145/2737924.2737966`. URL: `http://doi.acm.org/10.1145/2737924.2737966`.

[136] Tobias Ospelt. *AFL-based Java fuzzers and the Java Security Manager.* `https://www.modzero.ch/modlog/archives/2018/09/20/java_bugs_with_and_without_fuzzing/index.html`. Accessed April 17, 2019. 2018.

[137] OUSPG. *radamsa: a general-purpose fuzzer.* `https://gitlab.com/akihe/radamsa`. Accessed June 30, 2020. 2007.

[138] OW2 Consortium. *ObjectWeb ASM.* `https://asm.ow2.io`. Accessed August 21, 2018. 2018.

[139] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. "Feedback-Directed Random Test Generation". In: *Proceedings of the 29th International Conference on Software Engineering.* ICSE '07. 2007, pp. 75–84. ISBN: 0-7695-2828-7. DOI: `10.1109/ICSE.2007.37`. URL: `https://doi.org/10.1109/ICSE.2007.37`.

[140] Rohan Padhye, Caroline Lemieux, and Koushik Sen. "JQF: Coverage-guided Property-based Testing in Java". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ISSTA 2019. 2019, pp. 398–401. ISBN: 978-1-4503-6224-5. DOI: `10.1145/3293882.3339002`. URL: `http://doi.acm.org/10.1145/3293882.3339002`.

[141] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. "Semantic Fuzzing with Zest". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. 2019, pp. 329–340. ISBN: 978-1-4503-6224-5. DOI: 10.1145/3293882.3330576. URL: http://doi.acm.org/10.1145/3293882.3330576.

[142] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. "Validity Fuzzing and Parametric Generators for Effective Random Testing". In: *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*. ICSE '19. 2019, pp. 266–267. URL: https://dl.acm.org/citation.cfm?id=3339777.

[143] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. "FuzzFactory: domain-specific fuzzing with waypoints". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019), pp. 1–29. DOI: 10.1145/3360600. URL: https://doi.org/10.1145/3360600.

[144] Rohan Padhye and Koushik Sen. "Travioli: A dynamic analysis for detecting data-structure traversals". In: *2017 IEEE/ACM 39th International Conference on Software Engineering*. ICSE'17. IEEE. 2017, pp. 473–483. DOI: 10.1109/ICSE.2017.50. URL: https://ieeexplore.ieee.org/document/7985686.

[145] Manolis Papadakis and Konstantinos Sagonas. "A PropEr Integration of Types and Function Specifications with Property-based Testing". In: *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*. Erlang '11. 2011, pp. 39–50. ISBN: 978-1-4503-0859-5. DOI: 10.1145/2034654.2034663. URL: http://doi.acm.org/10.1145/2034654.2034663.

[146] *PeachFuzzer*. https://www.peach.tech/. Accessed January 28, 2019. 2019.

[147] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. "T-Fuzz: fuzzing by program transformation". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 697–710.

[148] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. "Nezha: Efficient domain-independent differential testing". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 615–632.

[149] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. 2017, pp. 2155–2168. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134073. URL: http://doi.acm.org/10.1145/3133956.3134073.

[150] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. "Smart greybox fuzzing". In: *IEEE Transactions on Software Engineering* (2019).

[151] Sokhom Pheng and Clark Verbrugge. "Dynamic Data Structure Analysis for Java Programs". In: *Proceedings of the 14th IEEE International Conference on Program Comprehension*. ICPC '06. 2006, pp. 191–201. ISBN: 0-7695-2601-2. DOI: `10.1109/ICPC.2006.20`. URL: `http://dx.doi.org/10.1109/ICPC.2006.20`.

[152] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[153] Michael Pradel, Markus Huggler, and Thomas R. Gross. "Performance Regression Testing of Concurrent Classes". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. 2014, pp. 13–25. ISBN: 978-1-4503-2645-2. DOI: `10.1145/2610384.2610393`. URL: `http://doi.acm.org/10.1145/2610384.2610393`.

[154] Easwaran Raman and David I. August. "Recursive Data Structure Profiling". In: *Proceedings of the 2005 Workshop on Memory System Performance*. MSP '05. 2005, pp. 5–14. ISBN: 1-59593-147-3. DOI: `10.1145/1111583.1111585`. URL: `http://doi.acm.org/10.1145/1111583.1111585`.

[155] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "VUzzer: Application-aware Evolutionary Fuzzing". In: *Proceedings of the 2017 Network and Distributed System Security Symposium*. NDSS '17. 2017.

[156] *React.JS*. `https://reactjs.org`. Accessed August 24, 2018. 2018.

[157] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. "Optimizing Seed Selection for Fuzzing". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. 2014, pp. 861–875. ISBN: 978-1-931971-15-7. URL: `http://dl.acm.org/citation.cfm?id=2671225.2671280`.

[158] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. "Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning". In: *42nd International Conference on Software Engineering (ICSE)*. ICSE '20. 2020, pp. 23–29.

[159] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366.

[160] Olatunji Ruwase and Monica S Lam. "A Practical Dynamic Buffer Overflow Detector." In: *Proceedings of the 2004 Network and Distributed System Security Symposium*. Vol. 2004. NDSS '04. 2004, pp. 159–169.

[161] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. "Parametric Shape Analysis via 3-valued Logic". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. 1999, pp. 105–118. ISBN: 1-58113-095-3. DOI: `10.1145/292540.292552`. URL: `http://doi.acm.org/10.1145/292540.292552`.

[162] *ScalaCheck: Property-based testing for Scala.* https://www.scalacheck.org/. Accessed January 28, 2019. 2019.

[163] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. "Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. 2013, pp. 488–498. ISBN: 978-1-4503-2237-9. DOI: 10.1145/2491411.2491447. URL: http://doi.acm.org/10.1145/2491411.2491447.

[164] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. 2005.

[165] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. "MultiSE: Multi-path symbolic execution using value summaries". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 842–853.

[166] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A fast address sanity checker". In: *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 2012, pp. 309–318.

[167] Konstantin Serebryany and Timur Iskhodzhanov. "ThreadSanitizer: data race detection in practice". In: *Proceedings of the workshop on binary instrumentation and applications*. 2009, pp. 62–71.

[168] Kostya Serebryany, Vitaly Buka, and Matt Morehouse. *Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator*. 2017.

[169] Micha Sharir and Amir Pnueli. "Two approaches to interprocedural data flow analysis". In: *Program Flow Analysis: Theory and Applications*. Ed. by Muchnick and Jones. 1981.

[170] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. "Automating Performance Bottleneck Detection Using Search-based Application Profiling". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. 2015, pp. 270–281. ISBN: 978-1-4503-3620-8. DOI: 10.1145/2771783.2771816. URL: http://doi.acm.org/10.1145/2771783.2771816.

[171] Olin Shivers. "Control-Flow Analysis of Higher-Order Languages". PhD thesis. Carnegie Mellon University, May 1991.

[172] V. Singh, R. Gupta, and I. Neamtiu. "MG++: Memory graphs for analyzing dynamic data structures". In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Mar. 2015, pp. 291–300. DOI: 10.1109/SANER.2015.7081839.

[173] Emin Gün Sirer and Brian N. Bershad. "Using Production Grammars in Software Testing". In: *Proceedings of the 2Nd Conference on Domain-specific Languages*. DSL '99. 1999, pp. 1–13. ISBN: 1-58113-255-7. DOI: `10.1145/331960.331965`. URL: `http://doi.acm.org/10.1145/331960.331965`.

[174] Steven S. Skiena. *The Algorithm Design Manual*. second. 2009. ISBN: 9781848000704. URL: `https://books.google.com/books?id=7XUSn0IKQEgC`.

[175] *Software Fail Watch: 5th Edition*. `https://www.tricentis.com/resources/software-fail-watch-5th-edition/`. Retrieved: June 2020.

[176] Linhai Song and Shan Lu. "Statistical Debugging for Real-world Performance Problems". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. 2014, pp. 561–578. ISBN: 978-1-4503-2585-1. DOI: `10.1145/2660193.2660234`. URL: `http://doi.acm.org/10.1145/2660193.2660234`.

[177] Richard M. Stallman et al. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. 2009. ISBN: 9781441412768.

[178] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In: *Proceedings of the 2016 Network and Distributed System Security Symposium*. NDSS '16. 2016.

[179] Robert Swiecki. *hongfuzz*. `https://honggfuzz.dev/`. Accessed June 30, 2020. 2010.

[180] *test.check: QuickCheck for Clojure*. `https://github.com/clojure/test.check`. Accessed January 28, 2019. 2019.

[181] Nikolai Tillmann and Jonathan de Halleux. "Pex - White Box Test Generation for .NET". In: *Proceedings of Tests and Proofs*. Apr. 2008.

[182] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. "Synthesizing Programs That Expose Performance Bottlenecks". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. 2018, pp. 314–326. ISBN: 978-1-4503-5617-6. DOI: `10.1145/3168830`. URL: `http://doi.acm.org/10.1145/3168830`.

[183] David A Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. "A first step towards automated detection of buffer overrun vulnerabilities." In: *Proceedings of the 2000 Network and Distributed System Security Symposium*. NDSS '00. 2000.

[184] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. "Superion: Grammar-Aware Greybox Fuzzing". In: *41st International Conference on Software Engineering*. ICSE '19. 2019.

[185] *wf - Simple word frequency counter*. Accessed Jan 2018. 2017. URL: `https://fedora.pkgs.org/27/fedora-x86_64/wf-0.41-16.fc27.x86_64.rpm.html`.

[186] John Whaley and Monica S. Lam. "Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams". In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. PLDI '04. 2004, pp. 131–144. ISBN: 1-58113-807-5. DOI: 10.1145/996841.996859. URL: http://doi.acm.org/10.1145/996841.996859.

[187] David H. White, Thomas Rupprecht, and Gerald Lüttgen. "DSI: An Evidence-based Approach to Identify Dynamic Data Structures in C Programs". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. 2016, pp. 259–269. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931071. URL: http://doi.acm.org/10.1145/2931037.2931071.

[188] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. "The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools". In: *ACM Trans. Embed. Comput. Syst.* 7.3 (May 2008), 36:1–36:53. ISSN: 1539-9087. DOI: 10.1145/1347375.1347389. URL: http://doi.acm.org/10.1145/1347375.1347389.

[189] David A Wood, Garth A Gibson, and Randy H Katz. "Verifying a multiprocessor cache controller using random test generation". In: *IEEE Design & Test of Computers* 7.4 (1990), pp. 13–25.

[190] Yichen Xie, Andy Chou, and Dawson Engler. "Archer: using symbolic, path-sensitive analysis to detect memory access errors". In: *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. 2003, pp. 327–336.

[191] Bin Xin, William N. Sumner, and Xiangyu Zhang. "Efficient Program Execution Indexing". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. 2008, pp. 238–248. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375611. URL: http://doi.acm.org/10.1145/1375581.1375611.

[192] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. 2011.

[193] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. "Droidfuzzer: Fuzzing the android apps with intent-filter tag". In: *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*. 2013, pp. 68–74.

[194] Shin Yoo and Mark Harman. "Pareto efficient multi-objective test case selection". In: *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM. 2007, pp. 140–150.

[195] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC'18. 2018, pp. 745–761. ISBN: 978-1-931971-46-1. URL: `http://dl.acm.org/citation.cfm?id=3277203.3277260`.

[196] Michał Zalewski. *American Fuzzy Lop*. `http://lcamtuf.coredump.cx/afl`. Accessed March 20, 2019. 2014.

[197] Michał Zalewski. *American Fuzzy Lop Technical Details*. `http://lcamtuf.coredump.cx/afl/technical_details.txt`. Accessed March 20, 2019. 2017.

[198] Michał Zalewski. *FidgetyAFL*. `https://groups.google.com/d/msg/afl-users/fOPeb62FZUg/CES5lhznDgAJ`. Accessed Jan 28th, 2019. 2016.

[199] Michał Zalewski. *Fuzzing random programs without execve()*. `https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html`. Accessed March 20, 2019. 2014.

[200] Dmitrijs Zaparanuks and Matthias Hauswirth. "Algorithmic Profiling". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. 2012, pp. 67–76. ISBN: 978-1-4503-1205-9. DOI: `10.1145/2254064.2254074`. URL: `http://doi.acm.org/10.1145/2254064.2254074`.

[201] Saman Taghavi Zargar, James Joshi, and David Tipper. "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks". In: *IEEE Communications Surveys & Tutorials* 15.4 (2013), pp. 2046–2069.

[202] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. "Automatic Generation of Load Tests". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. 2011, pp. 43–52. ISBN: 978-1-4577-1638-6. DOI: `10.1109/ASE.2011.6100093`. URL: `http://dx.doi.org/10.1109/ASE.2011.6100093`.

Author's Biography

Rohan Padhye is completing a Ph.D. in Computer Science at UC Berkeley, advised by Koushik Sen. He previously worked at IBM Research India and holds a master's degree from the Indian Institute of Technology Bombay. His research focuses on dynamic program analysis and automatic test-input generation. Complementing his doctoral work, he interned at Microsoft Research and Samsung Research America, developing techniques to automatically find software bugs in large-scale production systems. He is also the lead designer of the ChocoPy programming language, which currently underpins the undergraduate compilers course at Berkeley. He is the recipient of an ACM SIGSOFT Distinguished Paper Award, an ACM SIGSOFT Distinguished Artifact Award, a Tool Demonstration Award, an SOSP Best Paper Award, the C.V. Ramamoorthy Distinguished Research Award, and an Outstanding Graduate Student Instructor Award.

In Fall 2020, Rohan Padhye will join Carnegie Mellon University's School of Computer Science as an Assistant Professor in the Institute for Software Research.