Expanding the Reach of Fuzz Testing

Caroline Lemieux

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-43

http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-43.html

May 11, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Expanding the Reach of Fuzz Testing

by

Caroline Lemieux

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 Ion Stoica
Assistant Professor Peng Ding

Spring 2021

Expanding the Reach of Fuzz Testing

Copyright 2021
by
Caroline Lemieux

Abstract

Expanding the Reach of Fuzz Testing

by

Caroline Lemieux

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Koushik Sen, Chair

Software bugs remain pervasive in modern software systems. As software becomes increasingly intertwined in all aspects of our lives, the consequences of these bugs become increasingly severe. Since the mid-2010s, several important security vulnerabilities have emerged from basic correctness bugs in software. Modern fuzz testing or *fuzzing* tools have greatly helped reduce the consequences of these bugs. These fuzzing tools can automatically find inputs revealing bugs in large-scale software. Arming developers with these tools allows them to find bugs more rapidly, before they have significant security impacts.

However, even the most sophisticated modern fuzzing algorithms remain restricted in the software quality problems they solve. Most of the bugs they find automatically are memory mismanagement issues typical of C/C++ programs—like out-of-bounds array reads—and most of these bugs lie in the shallower parsing stages of programs. Further, in spite of the impressive search algorithms underlying modern fuzzing tools, they have only really been applied to this test-input generation problem. This thesis presents several algorithms which adapt fuzzing to new testing domains, and even looks at applying these algorithms to the problem of program synthesis. Taken as a whole, these algorithms provide a view of the promise of modern fuzzing algorithms, and how to alter these algorithms to solve diverse software quality problems.

This thesis finds that three key components of modern fuzzing algorithms must be extended in order to solve broader software quality problems. First, this thesis presents a generalized notion of feedback-directed fuzzing, which can be used to automatically find different types of bugs, including algorithmic complexity bugs, extreme memory allocations, and to target recently-modified code. Second, this thesis explores how well-structured mutations are key to enabling mutational fuzzers to explore deeper program states, and find bugs beyond those in parsers. Finally, this thesis shows that viewing random input generators as a specification of a search space, and adjusting the sampling distribution of these generators automatically, enables effective blackbox validity fuzzing and program synthesis for large APIs.

To my grandfather, who asked not if, but *where*, I would do my PhD.

Contents

List of Figures

List of Tables

Acknowledgments

While I was in the process of writing this dissertation, I was told that the ultimate product of my Ph.D. is not my dissertation, but the researcher and person I have become. Accordingly, I must thank all those who formed me both as a researcher and person over the course of my doctoral studies. Finishing a Ph.D. during the 2020-21 academic year was certainly strange, thanks in no small part to the COVID-19 pandemic which forced me to work from home from March 2020 onwards. Suffice to say, I could not have done it without all these folks.

First, I must thank my advisor Koushik Sen. When I visited Berkeley in 2016, I did not think it was where I would end up pursuing my Ph.D. studies—I didn't have any connections to the Computer Science department there before applying. But, sitting in Koushik's office as he explained concolic execution to me and told me about this new exciting AFL tool, I saw a research agenda pop up in front of my eyes. So, I came to Berkeley to accomplish it. Koushik helped me throughout my Ph.D. in honing my technical skills—the projects I start now I would not have dared starting in 2016—as well as my research communication skills. I have always appreciated the confidence he has put in me as a researcher, especially at times when I was unsure of myself; thank you, Koushik.

I must also extend thanks to the folks who integrated me into the Computer Science community at UBC, without whom I would never have gotten to Berkeley. First, Gregor Kiczales, who encouraged me to apply for a TAship after seeing me answer questions in a class he subbed. He and Meghan Allen helped further integrate me into the CPSC 110 TA community. Thank you to Ivan Beschastnikh, who sent me an email in early 2014 asking me if I was interested in a research opportunity; this led to the Texada project. I grew a lot as a researcher and communicator under his mentorship. Also thanks to Ron Garcia, who further integrated me into research and exposed me to the field of Programming Languages.[1] Finally, thank you to Gail Murphy and Elisa Baniassad, who kept in touch with me throughout the Ph.D. and encouraged me in my academic career.

Many other professors have helped form me as a researcher, integrated me into the computer science community, opened up career opportunities, and given me advice through various different roadblocks: Ion Stoica, Yuriy Brun, Andreas Zeller, Claire Le Goues, Yves Le Traon, David Patterson, Alvin Cheung, Sarah Chasins, Marcel Böhme and Abhik Roydchoudhury, amongst others. My 2018 internship at Google helped me better understand the impact of my work and broaden my research horizons. I am thankful for my mentors and team members there, who helped me both during my internship as well as with advice over the rest of my career: Stefan Bucur, László Szekeres, Franjo Ivančić, Domagoj Babic, Wei Wang, Tim King, Markus Kusano.

This dissertation contains several works conducted jointly with my collaborators at Berkeley, who deserve extra gratitude. Rohan Padhye and I first started working together in Fall 2017 on a project to build a tool that found performance bugs in Java programs; I led the implementation on the half that became PerfFuzz (Chapter 3), and he led the

[1]And also, gave me the piece of advice with which I opened these acknowledgments.

implementation on the half that became Zest (Chapter 6). We wrote both the papers together. He led the implementation of the FuzzFactory (Chapter 4) framework as well—I built the incremental fuzzing application and helped with writing the paper, proofs, etc. My collaborations with Rohan were not only productive, but also great fun, and revealed to me the importance of collaborators in my research style.

I worked with Sameer Reddy while he was an undergraduate at Berkeley; I posed him the validity guidance problem statement given in Chapter 7, and he ran with it from there. We decided to turn the project into a paper after his initial tech report showed good results. I then led the paper writing, with Rohan Padhye helping with the writing and positioning of the paper. I also want to thank my other undergraduate collaborators at Berkeley: Purva Gupta, Sicheng Liang, Jonathan Shi, and Neil Kulkarni. They not only conducted great work, but also taught me a lot about mentorship.

I am thankful to have had the opportunity to have worked with Rohan Bavishi on the AutoPandas project (Chapter 8). He led the implementation of the project; I helped with some abstraction components, but primarily as a brainstorming sounding board and with paper writing. Rohan and I have continued to collaborate, even through the COVID-19 pandemic; I am continually impressed my his work ethic and the inventiveness of his work. Having a colleague to brainstorm ideas with and to commiserate with through this very strange year has been immensely valuable, and I am thankful for his friendship.

It takes a village of graduate students to build a graduate student. I am certain I will miss some names, but I might as well try. I want to thank my office mates, who brought entertainment and company to my day-to-day work in the lab: Kevin Läufer, Wontae Choi, Liang Gong, Rafael Dutra, Alex Reinking, Ben Mehne, Ed Younis, Giulia Giudi, Alok Tripathy. Thanks also to my broader lab mates throughout the Programming Systems group at Berkeley, some of whom I had the joy to meet in person, others who I mostly met with through Zoom: Andrew Head, Rolando Garcia, Justin Lubin, Gabriel Matute, Justin Wong, Matthew Milano, Gilbert Bernstein, Lisa Rennels.

The first semester of my Ph.D. was one of the most fun social times in my life, thanks to several other students in my cohort, some of whom I have mentioned, but also including Michael Chang, Stephanie Wang, Tyler Westenbroek, Jeremy Warner, Laura Brinker, Stanley Smith, and others. Rent conditions in Berkeley necessitate having several roommates, and over the years I lived with Erin Grant, Esther Rolf, Victoria Cheng, Carolyn Chen, Amanda McLaughlin, Olivia Ashmoore, Rachel Chen and Phillip Sanvictores. Though there are ups and downs in any roommate relationship, I have learned so much from these folks and made some long-lasting friendships along the way; thanks to all of you.

I must give special thanks to Benjamin Brock for his companionship throughout our PhD journeys. I have learned and grown a lot over these five years, and Ben has been by my side throughout. I am thankful that he helped me push my comfort zone a little in traveling, and shared some lovely trips with him. Good timing, given the pandemic that would soon hit us and keep us in Berkeley indefinitely! I think the thing I am most thankful for—beyond the patient support he provided me when I consistently freaked out before *every single one* of my job interviews—is the joy and levity he brings to my life. Apparently many folks stop

laughing once they turn 23 (according to some studies in the book *Humour, Seriously*), but that is certainly not what I have experienced; I have Ben to thank for that.

Last but not least, I am thankful for my family. My paternal grandfather was an academic, and perhaps unsurprisingly as a result, much of my family is. I know this gave me a big leg up in academia, and am thankful for my privilege. Though I am also thankful for the family members out of academia, who help keep me grounded (and bring a bit more spice to family gatherings—can you imagine how boring it is to be around academics all the time?).

But the biggest thanks go out to my parents. I would not be here without them for very fundamental existence reasons, of course, but also more directly. My parents never really pushed me to get good grades in school or to enroll in a whole bunch of extra curricular activities, something which surprised folks seeing my GPA. But I think they modeled the important things for me in their work ethic, in their prioritization of family activities, in helping me ponder about what really made me happy.[2] I am so thankful to have had my mother as a role model as a woman in a highly technical field; knowing her made me know that I, too, belonged as a woman in a highly technical field. Though things are not perfect, they have gotten better since she was a graduate student, and I am very grateful for that.

My parents have always been just a phone call away—I cannot estimate how many hours I've spent talking to them throughout my graduate studies—and have given me such valuable advice at all the little frustration points in my graduate studies. I could write so much more about how thankful I am for them, how blessed and privileged I have been to have been born their child, but we should start talking about fuzz testing soon, so in close, let me just say: *merci infiniment.*

[2]When I was 10 or so, I was very frustrated playing some video game on the PlayStation2. My mom, seeing this, asked me why I was "playing" a video game if it just made me annoyed. That little throw-away comment deeply stuck with me, and I still find myself returning to this advice in my day-to-day life.

Chapter 1

Introduction

Software bugs remain pervasive in modern software systems. As software becomes increasingly intertwined in all aspects of our lives, the consequences of these bugs become increasingly severe. For instance, in the last few years, several important security vulnerabilities have emerged from basic correctness bugs in software [101, 155, 89].

Though not all bugs cause important security vulnerabilities, their overall cost remains high: one estimate puts the cost of dealing with software failures and defects *in the United States alone* in 2018 at nearly USD$1,540,000,000 [108].

This dissertation explores methods to help developers improve the quality of software before bugs ship into productions and incur these costs. The particular focus is on *fuzz testing* or *fuzzing* tools. These tools automatically find bug-inducing inputs in programs; inputs x that, when run on a program p, cause the program to crash.

Access to such tools greatly helps reduce the time to find and fix bugs, and thus, reduces the impact of these bugs. As a concrete example, consider a case study of the OpenSSL library [73]. In 2012, a bug was introduced into the library which allowed an attacker to read memory from a server that did not belong to them [144]. This bug, later named Heartbleed, was only discovered in 2014. By that time, it had spread widely, and was exploited to great cost. For instance, it enabled the leak of millions of patient records from the U.S.'s second largest hospital chain [76]. It also forced the Canada Revenue Agency website to shut down for several days [116], but not before over 900 Social Insurance Numbers were leaked from the agency's website [143].

By contrast, nearly 4 years later, another bug was introduced into OpenSSL. This bug could have allowed an attacker to execute arbitrary code, and was judged as much more severe than Heartbleed [145]. However, there are no widely publicized costs associated with this bug, CVE-2016-6309. This may be in large part because it was discovered one day after its introduction by a modern fuzz testing tool, honggfuzz [174].

The term *fuzz testing* dates back to Miller et al.'s seminal work from 1990 [135], but has re-emerged as an area of interest following the appearance of the pioneering AFL [185] around 2014. Fuzz testing tools assume the following program setting. There is a program p which runs on some input x. For the purposes of this dissertation, differences in the behavior of p

on different executions should be entirely explained by differences in the input x. That is, the program should be deterministic, single-threaded, and relatively fast-running. In the classical fuzzing domain, the input x is typically a file—modelled as a sequence of bytes—processed by the program p, but it can also be a more general data structure.

Fuzzing tools then repeatedly run the program p on inputs x that are generated via a variety of random search procedures, until an input x^* is found which induces a bug in p. Typically, these tools identify the presence of a bug via universal correctness oracles — if the program crashes or exits earlier due to an assertion error. The pure random fuzzing introduced by Miller et al. [135] simply created the inputs x by sampling random sequences of bytes, but modern fuzzing algorithms are much more sophisticated in their input creation methodology. Many modern fuzzing algorithms rely on detailed program feedback to guide the generation of inputs (Chapters 3, 4, 5, 6) or rely on user-provided information about input structure to make the search more targeted (Chapters 6, 7, 8).

Overall, this dissertation presents several methods to make fuzz testing tools more widely applicable and easy-to-use. This dissertation is divided into three main parts, each of which identifies a different key component of modern fuzzing algorithms, and how to generalize this component for different fuzzing goals: new bug domains, deeper program exploration, and applications beyond testing. Chapter 2 provides a discussion of related work, including detailed background of a particular coverage-guided fuzzing algorithm, AFL. The rest of this introduction describes the two modern fuzzing algorithms studied in this dissertation, and the key drawbacks addressed in each subsequent part of the dissertation.

1.1 Coverage-Guided Fuzzing

In the mid-2010s, coverage-guided fuzzing (CGF) emerged as one of the most effective techniques for fuzzing real-world software. Pioneered by AFL [185], CGF has been implemented in several other popular tools including libFuzzer [167] and honggfuzz [174].

Like random fuzzing [135, 133]—which sends many random inputs to the program under test—CGF works by executing a test program with a large number of inputs generated via random search. However, instead of generating totally random inputs from scratch, CGF mutates inputs from a set of saved inputs to derive new inputs. Algorithm 1 walks through the high-level algorithm shared by most implementations of CGF.

The CGF algorithm takes an input an instrumented program (p) and a set of user-provided *seed inputs* (S_0). Again, typically this program accepts as input a byte-sequence (either via a file or standard input), and thus, each input in S_0 is modelled as a sequence of bytes.

CGF maintains two global states: (1) $\mathcal{S}$, a set of saved inputs to be mutated by the algorithm, and (2) *totalCoverage*,which tracks the cumulative coverage of the program on the inputs in $\mathcal{S}$. The instrumented test program p returns the coverage achieved by the input it is executed on. The forms of coverage most commonly used by CGF are branch coverage, basic block coverage, or basic block transition coverage. $\mathcal{S}$ is initialized to the set of

Algorithm 1 The generic coverage-guided fuzzing algorithm.

```
Input: an instrumented test program $p$, a set of initial seed inputs $S_0$
Output: a corpus of automatically generated inputs $\mathcal{S}$
 1: $\mathcal{S} \leftarrow S_0$
 2: $\textit{totalCoverage} \leftarrow \textsc{initCoverage}(S_0)$
 3: repeat                                                        ▷ Main fuzzing loop
 4:   for $\textit{input}$ in $\mathcal{S}$ do
 5:     with probability $\textsc{fuzzProb}(\textit{input})$ do
 6:       for $1 \le i \le \textsc{numChildren}(\textit{input})$ do
 7:         $\textit{input}' \leftarrow \textsc{mutate}(\textit{input})$          ▷ Generate new test input $\textit{input}'$
 8:         $\textit{coverage} \leftarrow \textsc{execute}(p, \textit{input}')$          ▷ Run test with $\textit{input}'$
 9:         if $\textit{coverage} \not\subseteq \textit{totalCoverage}$ then
10:           $\mathcal{S} \leftarrow \mathcal{S} \cup \{\textit{input}'\}$          ▷ Save $\textit{input}'$ if it covers new code
11:           $\textit{totalCoverage} \leftarrow \textit{totalCoverage} \cup \textit{coverage}$
12: until given time budget expires
13: return $\mathcal{S}$
```

user-provided seed inputs (Line 1) and *totalCoverage* is initialized to the coverage achieved by those user-provided seed inputs (Line 2).

The main coverage-guided fuzzing loop goes over each input *input* in the set of inputs $\mathcal{S}$. With some probability determined by an implementation-specific heuristic function $\textsc{fuzzProb}$, CGF decides whether to mutate the input *input* or not (Line 5). If *input* is selected for mutation, CGF decides on how many mutant inputs to generate from *input* via another heuristic function $\textsc{numChildren}$ (Line 6).

Then, $\textsc{numChildren}(\textit{input})$ times, CGF randomly mutates *input* to generate a mutant $\textit{input}'$. A random mutation typically involve choosing a random set of bytes in the inputs and performing a mutation operation there, like: bit flipping, byte flipping, incrementing and decrementing integer values, or replacing bytes with "interesting" integer values (0, `MAX_INT`). CGF executes the program with the newly generated input and collects the coverage of the input in the temporary variable *coverage* (Line 8). If the observed coverage contains some previously-unseen coverage points (Line 9), the new input $\textit{input}'$ is saved to the set of inputs $\mathcal{S}$ (Line 10). This $\textit{input}'$ can now be mutated during a future iteration of the fuzzing loop. The process repeats until the time budget expires.

1.1.1 Drawback: Fixed Testing Goal

Coverage-guided fuzzing tools were conceived as catch-all security auditing tools, to automatically find input-dependent crashes in the program under test. For this broad testing goal, where the location of a bug is unknown, increased coverage is a reasonable testing signal under which to save inputs. The logic is that this encourages broad exploration of paths through the program, and that exercising all paths in a program should reveal any bugs in the program.

However, not all bugs are uniformly distributed throughout the program under test, and furthermore, certain bugs are not revealed by branch coverage alone. For instance, a security auditor may know that bugs in a particular program component are likely to be more severe, and want to direct input generation to exercise that component. Or, if an input causes a negative data value to flow into the argument of a memory allocation, the program under test may use an unreasonable amount of memory, even if the coverage of the program under test is not abnormal for that input.

Unfortunately, the notion of interesting inputs being those that increase branch coverage is generally quite tightly integrated into coverage-guided fuzzing tools, and so, these testing goals cannot be achieved. Chapters 3 and 4 address this drawback directly. Chapter 3 introduces a multi-objective feedback-directed fuzzing algorithm and shows that this algorithm can be used to effectively find inputs with pathological performance behavior. Chapter 4 further generalizes this feedback-directed fuzzing algorithm, and introduces a framework that allows for easier customization of the testing goal.

1.1.2 Drawback: Malformed Inputs

Part of the success of random testing methods, including random fuzzing, comes from the fact that these methods produce inputs outside of the input space the software developer has reasoned about. As such, random testing quickly reveals the program's behavior in exceptional—and often buggy—circumstances.

Coverage-guided fuzzing builds on this intuition as well, with a twist. By generating inputs via small mutations to existing inputs, it can better ensure that the inputs still get to some interesting program state (by relying on reasonable inputs to start with), while still exploring corner cases (revealed via the small mutations of the input). This means that the inputs generated by CGF find deeper corner cases in input validation than an approach that relies on purely random sequences of characters as input.

However, the byte-level mutations applied by CGF are still likely to generate inputs that are, overall, malformed. This is great to stress test parsers, but means that it is very difficult for coverage-guided fuzzing to consistently generate inputs that pass the validation checks of a program and explore its core logic. Chapters 5 and 6 address this drawback. In particular, Chapter 5 describes a mutation masking approach that enables CGF to explore the program under test more deeply without any additional input from the user. Chapter 6 presents an approach that can explore the core logic of programs even better, but relies on a user-written generator to perform high-level mutations during coverage-guided fuzzing.

```
# Generate a (possibly negated) integer as a string
def generate_unexpr():
    val = str(random.integer())
    if random.boolean():
        val += "-"  + val
    return val

# Generate a binary expression
def generate_binexpr():
    lhs = generate_expr()
    op = random.choice(["-", "+", "/", "*"])
    rhs = generate_expr()
    return lhs + op + rhs

# Generate a (possibly parenthesized) expression
def generate_expr():
    if random.boolean():
         expr = generate_binexpr()
    else:
         expr = generate_unexpr()
    if random.boolean():
         expr = "(" + expr + ")"
    return expr
```

Figure 1.1: Generator of expressions in a simple calculator language.

1.2 Generator-Based Fuzzing

Coverage-guided fuzzing relies heavily on feedback from the programs's execution in its input generation strategy because it has so little information about the structure of inputs to the program under test. When information about input structure is available, the input generation strategy can be made much more effective.

This is the key of the effectiveness of generator-based fuzzing (also called *property-based testing* [58]). The core idea is to create new inputs by repeatedly calling a (typically, user-written) generator. This generator is a non-deterministic function, which, each time it is called, returns a random input in a given search space. An example generator is given in Figure 1.1: each time `generate_expr` is called, it returns an input in a simple calculator language, e.g. "`14`", "`(3) + 119`", or "`(87+1230-(467/234*2))`".

Given a generator like that in Figure 1.1, the fuzzing process is quite straightforward. Given a program p, repeatedly call the generator to generate an input x. As in coverage-guided fuzzing, run p on x to observe whether x induces a bug (i.e., a crash or assertion failure).

While the generator in Figure 1.1 still generates byte-sequence inputs, one can easily write similar generators for data structures. The term *property-based testing*, introduced by QuickCheck for Haskell [58], is used to refer to generator-based fuzzing in a more unit testing context. There, the program $p(x)$ takes in inputs of a type $\mathcal{X}$ and should encode a property $P(x) \Rightarrow Q(x)$ to be tested. For some types $\mathcal{X}$, a generator can be derived automatically from the type definition. By generating many inputs $x \in \mathcal{X}$ and running them through p, the developer can approximately check that the property $\forall x, P(x) \Rightarrow Q(x)$ holds.

1.2.1 Drawback: Coupling of Distribution and Search Space

The key drawback of generator-based fuzzing emerges when the domain of the generator does not closely match the space of "interesting" inputs for the program under test. In particular, if the program encodes a property $P(x) \Rightarrow Q(x)$, and very few inputs x generated by the generator satisfy $P(x)$, then most of the runs of the program under test are simply validating the case in which the property is vacuously true.

For example, for the generator of expression in Figure 1.1, suppose the developer wants to validate that in the presence of a division by zero, an exception is thrown *before* the evaluation of a division by zero. So $P(x)$ is true for inputs x which contain a division by zero. But many of the inputs sampled from the generator in Figure 1.1 will not include division by zero. The developer could spend some time tuning the generator in Figure 1.1 to increase the probability of generating inputs with a division by zero, for example, by forcing more generation of division expressions with the denominator as the integer literal `0`.

Requiring the developer to conduct this tuning comes with its own set of challenges. First, of course, tuning a generator so that most of the inputs x it generates satisfies $P(x)$ is a significantly more complex task than writing a simple generator of inputs. Second, as a human tries to tune the generator to a smaller input space, they may prune the space of inputs generated unnecessarily. There are many ways to write an expression that has a division by zero, e.g. "`4/(6-3*2)`". If the developer did the simple "add more divisions by `0`" tuning described above, they may prevent the generation of such expressions. The problem gets worse as the input space becomes more complex, e.g. inputs in a programming language.

The core problem here is that a generator is a specification of a search space (the domain of inputs x which can be generated by some path through the generator) paired with a probability distribution over that search space (the probability of any particular path being taken through the generator). If we could simply adjust the probability distribution from which inputs are drawn, we could automatically adapt a generic generator to a particular $P(x)$ without requiring the user to think about distributions. Chapter 6 discusses a method to increase the number of valid inputs generated by indirectly adjusting these probability distributions via coverage-guided fuzzing. Chapter 7 presents a reinforcement learning approach to tune these distributions more directly. Finally, Chapter 8 shows how this same abstraction—splitting the generator of elements from the distribution from which those elements are drawn—can be used for program synthesis.

Chapter 2

Background and Related Work

In this section, we first provide additional background on AFL [185], and then discuss the numerous fuzz testing and input-generation tools related to the work in this dissertation.

The discussion of related work focuses on those which are the most pertinent to this work in this dissertation. Some survey papers provide additional context. Valentin et al. present a unified model of fuzzing—blackbox, greybox and whitebox—, as well as a genealogy of significant fuzzers up to 2019 [127]. The model of coverage-guided fuzzing in this dissertation is less overarching, but instead focuses on the core components that must be altered for different testing goals. Godefroid's 2020 review overview the field at a higher-level, highlighting the *differences* between different branches of fuzzing, rather than unifying them in a single model [79]. Klees et al. discuss some of the common issues in evaluating different fuzzers, and how to build more consistent baselines [106].

2.1 American Fuzzy Lop (AFL)

American Fuzzy Lop, or AFL [185], was the first tool to introduce the coverage-guided fuzzing algorithm, which we discussed previously.

A number of the algorithms presented in this dissertation (Chapters 3, 4, 5) are implemented directly on top of AFL. For context of the implementation details of these algorithms, we detail the way in which AFL implements the abstract coverage-guided fuzzing algorithm introduced in Algorithm 1. The main implementation-specific points are: (1) how coverage is gathered, (2) how inputs are selected for mutation (FUZZPROB in Line 5), (3) how many mutants are produced (Line 6), and (4) how mutations are performed (Line 7).

AFL Coverage Calculation The notion of coverage collected by AFL (*coverage* in Line 8 of Algorithm 1) while the program under test executes, and its use in the fuzzing loop, is one of AFL's key innovations.

In order to collect coverage information efficiently, AFL inserts instrumentation into the program under test. To track coverage, it first associates each basic block with a random number via compile-time instrumentation. The random number is treated as the *unique ID* of the basic block. The basic block IDs are then used to generate unique IDs for the transitions between pairs of basic blocks. In particular, for a transition from basic block A to B, AFL uses the IDs of each basic block—$ID(A)$ and $ID(B)$, respectively—to define the ID of the transition, $ID(A \rightarrow B)$, as follows:

$$ID(A \rightarrow B) \overset{def}{=} (ID(A) \gg 1) \oplus ID(B).$$

Where $\oplus$ designates bitwise exclusive or (xor). Right-shifting ($\gg$) the basic block ID of the transition start block (A) ensures that the transition from A to B has a different ID from the transition from B to A. In this dissertation, in particular in Chapter 5, we associate the notion of basic block transition with that of a *branch* in the program, unless stated otherwise.

The coverage of the program under test on a given input is collected as a set of pairs of the form (*branch ID*, *branch hits*). If a (*branch ID*, *branch hits*) pair is present in the coverage set, it denotes that during the execution of the program on the input, the branch with ID *branch ID* was exercised *branch hits* number of times. The branch hits are simplified into one of 8 buckets: hit 1 time, 2 times, 3 times, 4–7 times, 8–15 times, 16–31 times, 32–127 times, or 128–255 times. AFL calls this set of pairs the "path" of an input. AFL says that an input achieves *new coverage* if it discovers a new (*branch ID*, *branch hits*) pair.

Choosing which input to mutate. In Line 5 of Algorithm 1, an input is selected from the set of saved inputs for mutation with a probability FUZZPROB. To compute this, AFL first determines a set of *favored* inputs. AFL assigns a FUZZPROB of 100% to favored inputs that have not yet been mutated, and 1% to non-favored or already-mutated inputs. If the set of favored inputs is empty, it relaxes this a little, assigning 25% chance of mutating not already-mutated inputs, and a 5% chance of mutating already-mutated inputs.

The notion of favored input is crucial in this computation. AFL creates this set in a greedy manner. For each *branch ID* that has been covered, it finds the input with the smallest product of execution time and input length. This input is the winner for that *branch ID*. AFL assigns a winner as favored if it also covers a *branch ID* that had not been seen in the previous fuzzing iteration.

Choosing the number of mutants and mutating inputs. In the abstract CGF algorithm we separated the process of choosing the number of mutants to produce (Line 6) from the mutation process (Line 7). In AFL, these components are tightly intertwined. AFL's mutation strategies assume the input to the program under test is a sequence of bytes, and can be treated as such during mutation. AFL mutates inputs in two main stages: the *deterministic* stages and the *havoc* stage.

All the deterministic mutation stages operate by traversing the input under mutation and applying a mutation at each position in this input. These mutations include bit flipping, byte flipping, arithmetic increment and decrement of integer values, replacing of bytes with "interesting" integer values (0, `MAX_INT`), etc. The number of mutated inputs produced in each

Algorithm 2 "Havoc" mutations in AFL.

```
1: procedure MUTATEHAVOC($Prog, input$)
2:     $numMutations \leftarrow$ RANDOMBETWEEN(1,256)
3:     $newinput \leftarrow input$
4:     for $0 \leq i < numMutations$ do
5:         $mutation \leftarrow$ RANDOMMUTATIONTYPE
6:         $position \leftarrow$ RANDOMBETWEEN($0, |newinput|$)
7:         $newinput \leftarrow$ MUTATE($newinput$, $mutation$, $position$)
```

of these stages is governed by the length of the input being mutated. So, if the deterministic mutation stages are not skipped, part of NUMCHILDREN is simply a linear function of the input length.

On the other hand, the havoc stage works by applying a sequence of random mutations to the input being mutated to produce a new input. Algorithm 2 shows this process. Several mutations are repeatedly applied to the original input (Line 7) before running it through the program. The type of mutation that can be applied includes all the mutations that can be applied in the deterministic stages.

The number of total havoc-mutated inputs to be produced is determined by a performance score, which is the non-input-length dependent part of NUMCHILDREN. This performance score is higher for inputs with (a) faster execution times, (b) higher average coverage, (c) which have been more recently added to the saved input set, and (d) which have been discovered "deeper" in the fuzzing process (a mutant of a seed input has depth 1, a mutant of that mutant has depth 2, etc.).

Finally, AFL also includes a crossover-like mutation phase (called *splicing*) which combines sequences of the parent input with sequences of other saved inputs. None of algorithms discussed in this dissertation alter this stage of mutation.

Trimming Inputs Not illustrated in Algorithm 1 is an additional trimming stage before inputs are mutated. For additional efficiency, inputs are trimmed with an approximate delta-debugging method [191], which tries to reduce the size of inputs as much as possible while ensuring they cover the same "path" (the set of *(branch ID, branch hits)* pairs).

2.2 Random and Mutational Fuzzers

The term *fuzz* was introduced by Miller et al.'s seminal work on randomized testing of UNIX utilities [135]. After observing that random characters caused by rain on dial-up phone lines could cause remote command-line utilities to crash, Miller et al. developed the `fuzz` tool. This tool stress-tests command line utilities by repeatedly sending random sequences of characters as input to these utilities. This original work also included the `ptyjig` tool, essentially a generator-based fuzzer for interactive utilities.

This study found that numerous popular utilities—including `bc`, `emacs`, `ftp` `telnet`, and `vi`—could be crashed with inputs generated by this pure random fuzzing. The study was revisited five years later [136], and though fewer UNIX utilities could be caused to crash with this random fuzzing, many of the bugs were still present, and more than half of X-Window utilities could be crashed with the process. Subsequent studies on Windows NT GUI applications [72] and Mac OS utilities and GUI applications [134] found similar results. Finally, a 2020 study found that this method was still effective at finding crashes, showing the persistent prevalence of pointer and array-related bugs in C/C++ code [133].

A step more sophisticated than these pure random fuzzers are *mutational* fuzzers, which generate inputs by mutating some seed inputs, but without the genetic-algorithm-like loop introduced by AFL. Traditional blackbox mutational fuzzers such as zzuf [95] mutate user-provided seed inputs according to a mutation ratio, which may need to be adjusted to the program under test. BFF [100] and SymFuzz [53] adapt this parameter automatically. BFF measures the crash density—the proportion of generated inputs that cause crashes—for different mutation ratios, and adjusts the mutation ratio accordingly. SymFuzz conducts the heavier duty strategy of input-bit dependence inference to adapt the mutation ratio to the program under test.

Radamsa is a modern mutational fuzzer which has found numerous CVEs in sophisticated software components such as Chrome, Mozilla Firefox, Adobe Reader and CISCO WebVPN [29]. One of its appeals is its very simple command line interface, which is in part due to its black-box nature. Unlike traditional mutational fuzzers, it has some sophistication in its mutation techniques, i.e. it can recognize parts of inputs that look like numbers and mutate them as numbers rather than treating them as random bytes.

2.3 Coverage-Guided Fuzzers

Coverage-guided fuzzing piqued the interest of the practitioner and academic community after the publication of some impressive results on the part of AFL, which we have already discussed extensively. Particularly impressive results include the automatic discovery of JPEG structure [187] and the shellshock bug [186].

Several other industrial coverage-guided fuzzers were built after this. First is libFuzzer [167], integrated into the LLVM-based `clang` compiler [113] toolchain. The underlying input-generation method is fairly similar to AFL; a notable difference is that only one mutant is generated per parent (i.e. numChildren in Line 6 of Algorithm 1 always returns 1) and that the search terminates after the discovery of any crash. Unlike AFL, which operates by repeatedly calling a compiled external program under test, libFuzzer is *in-process*, meaning it repeatedly runs a single test driver function. This test driver takes in a sequence of bytes and the length of that sequence as input. In order to run libFuzzer properly, the test driver should be side-effect free, not leak memory, etc., so that it can be invoked hundreds of thousands of time. Unlike AFL whose feedback instrumentation is fixed, libFuzzer provides hooks that allow users to inject extra instrumentation at basic blocks and comparison operations.

Another notable coverage-guided fuzzer is honggfuzz [174]. It supports the same test driver abstraction as libFuzzer. The distinguishing feature of honggfuzz is its high-performance aspect: it supports multi-process and multi-threaded fuzzing out-of-the-box, as opposed to requiring the launch of multiple fuzzing runs. It is the fuzzer that found the notable critical security vulnerability in OpenSSL mentioned in the introduction [145].

AFLFast [46] is the seminal work on improving coverage-guided fuzzing in academia. It presents a Markov Chain model of the fuzzing process, and the hypothesis that inputs which exercise low-frequency paths —coverage sets exercised by very few fuzzer-generated inputs—are more likely to produce inputs finding new coverage in the program under test. It built from this model several improvements to the NUMCHILDREN, that caused relatively larger emphasis on the havoc mutation stages of inputs exercising rare paths. This led to large increases in coverage and decreases in time to reveal bugs compared to AFL, and inspired improvements of some of AFL's default heuristics [188].

VUzzer [161] adds several smarter data-flow and control-flow analyses to the coverage-guided fuzzing process. In a pre-fuzzing static analysis stage, VUzzer identifies basic blocks containing error handling code, in order to de-prioritize them during fuzzing, as well as deeply nested basic blocks, in order to prioritize them during fuzzing. VUzzer also uses taint analysis to improve its mutation strategies, for example to determine which input bytes are used in direct comparisons, as well as the values they are directly compared against. Thus it can more easily get through "hard" comparisons, as discussed in Chapter 4. VUzzer re-implements the whole coverage-guided fuzzing loop rather than building on top of e.g. AFL or libFuzzer.

Steelix [117] is another coverage-guided fuzzer built on top of AFL, whose goal is to better get through hard comparisons. It relies on extra instrumentation to detect whether progress is made in a multi-byte equality comparison, i.e. one byte matches in a multi-byte comparison. When it notices this, it runs an adaptive mutation phase, exhaustively checking all byte values for the adjacent bytes to the matched byte, and continuing this process until no more adjacent bytes can be made to match. This is similar to the `cmp` fuzzer discussed in Section 4.4.2, but with more targeted mutations. pFuzzer [129] leverages unsatisfied byte-level comparisons in order to extract the byte that must be present at the last position of an input to get through the comparison, especially well-suited to fuzzing recursive descent parsers.

Angora [54] also improves on AFL using taint-tracking and a gradient descent methodology. It uses the LLVM instrumentation framework to track which input bytes flow into different path constraints, and in order to try and get through those constraints, restricts its mutations accordingly. Angora treats each predicate in a path constraint as a blackbox function $f(\mathbf{x})$ over the part of the input $\mathbf{x}$ that flows into the predicate. It slightly modifies the $\mathbf{x}$ and, with the results of this mutant, uses the finite difference method to approximate the gradient of $f(\mathbf{x})$ to guide its search. In addition to this, Angora also tries to infer the type of different byte sequences (i.e. are these four bytes used as an integer). And, instead of simply looking at increased branch coverage as AFL does, it adds a notion of *context*: the coverage point includes not only the branch, but also a hash of the call stack when that branch was taken.

Matryoshka [55] adopts some of Angora's taint-tracking and gradient-descent strategies, but targets the generation of inputs that satisfy highly-nested conditional statements. Given a target condition, it first identifies the set of conditions that dominate it, as well as the data flow of input bytes to these conditions. Then it uses a variety of mutation masking procedures, and combining inputs that satisfy individual conditions, to create inputs that satisfy the highly-nested condition.

MOPT [126] alters the order in which different AFL mutation stages are applied, and whether they are applied at all. In particular, it allows for the scheduling of each different deterministic mutation stage, as well as the havoc and splicing (crossover) stage. It models each of these mutation stages as a particle moving in a probability space. Then it uses Particle Swarm Optimization to try and maximize the efficiency of mutation, i.e. keep the mutation stages which resulted in most inputs with new coverage.

RedQueen [32] uses a more lightweight approach to get through hard branches. It identifies correspondences between the input and the program state by simple equality; are there data strings in the program that are exact subsequences of the input. It instruments comparison operators to get this data for direct equality conditions. It can then patch the corresponding input data to get through the equality conditions. It uses a similar approach to get through checksums: first by identifying computations that look like checksums, and patching inputs with the checksum-computed values.

ProFuzzer [183] aims to recover some structural information from seed inputs in order to improve the performance of fuzzing. For each byte in the seed inputs, it generate mutants with each different possible value of the byte. Based on the changes in coverage caused by the mutation, it determines a category for the byte, e.g.: is it a constant, is it used in a loop count, is it an offset, a size. Then it groups together sequential bytes of the same category, and performs more relevant mutations based on the category.

Entropic [44] leverages the model of software testing as species discovery [43] to improve the heuristics, in particular fuzzProb in Line 5 of Algorithm 1, of libFuzzer. It introduces the notion of local Shannon's Entropy for a seed t — essentially a score of how likely mutants of t are to discover new branches. Then, fuzzProb is proportional to a Bayesian estimator of this entropy metric, over low-frequency branches. Due to consistent improvements in coverage achieved and reductions in time to expose bugs, this implementation of fuzzProb was incorporated into mainstream libFuzzer.

AFL++ [71] has been developed to integrate many of the innovations discussed above into one single fuzzer. It includes AFLFast's energy schedules, RedQueen's input-to-state replacement, MOPT's mutation scheduling, amongst others. It also provides an API to more easily customize the underlying fuzzer's input generation strategies. As of September 2020, it was the fuzzer with the highest overall score as per Google's FuzzBench project [175].

An alternative to a single mega-fuzzer is to combine multiple fuzzers in parallel. EnFuzz [55] found that an ensemble fuzzer that shared saved inputs between multiple different fuzzers generally achieved higher coverage and bug-finding ability compared to the sum of its parts.

2.4 Specialized Feedback-Directed Fuzzers

As will be discussed in Chapter 4, there has been interest in using the high-level coverage-guided fuzzing algorithm for different specialized use cases beyond general increased coverage of command-line utilities.

AFLGo [45] is a directed fuzzing tool, which aims to generate inputs hitting a set of targets, each of which is a line in a file. It uses call graphs obtained from whole-program static analysis to compute the distances of between basic blocks, and gives positive feedback to inputs that reduce this distance. Wüstholz and Christakis propose a directed fuzzing tool for smart contracts, Bran [180]. Instead of conducting a whole-program static analysis, it conducts online static analysis to determine the smallest no-target-ahead prefix of an input's path—assuming that input does not exercise the target. It uses information about the rarity of these prefixes to influence numChildren, and emphasize the mutation of inputs which have a rare no-target-ahead path.

Ijon [31] allows security analysts to insert feedback statements into the program under test, much like the approach described in Chapter 4. The feedback statements are at a different level, however. They allow the developer to increment/decrement/maximize feedback keys, as in Chapter 4, but also to enable and disable coverage feedback in different parts of the program, and also provide an explicit notion of state feedback.

SlowFuzz [157], built on top of libFuzzer, aims to find inputs showing algorithmic complexity vulnerabilities. The main idea is to create inputs with longer path lengths through the program by saving inputs with longer path lengths, and adapting the mutation strategy to locations that result in longer paths. Chapter 3 contrasts this approach with PerfFuzz.

Nezha is a differential fuzzing tool [156] which fuzzes multiple programs at the same time on one input and tries to maximize the difference in their behavior on that input. In particular, Nezha introduces the concept of δ-diversity and uses this to force the generation of inputs showing different behaviors in the program under test. It prioritizes the inputs which have radically different paths through the programs compared to their parents, as well as the outputs. This allows for faster discovery of inputs which illustrate semantic differences between the programs—most likely bugs. DifFuzz [140] also looks at running multiple versions of a program on a same input, but in order to find side-channel vulnerabilities. Essentially, the input being mutated consists of a public input as well as two distinct secrets. Then DifFuzz prioritizes inputs which increase the difference in execution cost between the runs of the program under the two secrets.

Memlock [178] aims to find memory consumption crashes, vulnerabilities, and memory leaks. It uses an approach similar to that described in Chapter 3 to accomplish this, but it instruments memory allocation and deallocation statements in order to build a *perfmap* mapping allocations locations to the amount of memory allocated there. MemFuzz [61] saves inputs that read/write new values to input-dependent memory addresses; this allows it to find new bugs compared to AFL.

Lauefer et al. [109] use validity feedback in order to fuzz circuits that have constrained interfaces. This validity feedback is similar to that described in Chapter 6. Harvey [179] is a greybox fuzzer optimized for smart contracts. It adds feedback that guides it through conditional statements, by translating conditional statements to linear expressions and trying to minimize those expressions. It also detects branches that require more than one transaction, and uses this bound the transaction sequences it explores. When exploring sequences of length more than 1, it only prioritizes increased coverage of the last transaction in the sequence.

Bugariu et al. build a fuzzer to test implementations of abstract domains [49] by building specialized oracles for static analysis bugs. Each of these oracles is encoded as a test-driver, which also encodes the generation of inputs in a given abstract interpretation domain. A coverage-guided backend controls the generation of these inputs. In some ways, this can be seen as a specialized guided generator-based fuzzer, although the inputs are captured as a sequence of operations in the abstract interpretation domain.

2.5 Structure-Aware and Generator-Based Fuzzers

There is a rich history of randomized testing methods that leverage specifications of input structure in order to generate inputs. At some level, most of these fall under the abstraction of generator-based fuzzing, but there are many different branches of generator-based fuzzing, and ways to utilize structural information in the production of inputs.

Generator-based fuzzing, also called property-based testing, which was discussed in the introduction, was first popularized by QuickCheck [58]. It allows users to quickly check a property of the form $P(x) \Rightarrow Q(x)$ over a domain of inputs $\mathcal{X}$. In particular, given a user-defined test driver implementing $P(x) \Rightarrow Q(x)$ with a generator of inputs $x \in \mathcal{X}$, QuickCheck will execute $P(x) \Rightarrow Q(x)$ on many inputs x. This gives an approximate check of $\forall x \in \mathcal{X}, P(x) \Rightarrow Q(x)$. The method has grown in popularity thanks to implementations in many different languages [14, 13, 18, 19, 151], including prominent languages such as Python [15], JavaScript [16], and Java [98].

UDITA [78] 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 [123, 124] guides input generators used in property testing towards a user-specified fitness value using techniques such as hill climbing and simulated annealing. GödelTest [67] 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, similar to the guides discussed in Part III.

One problem with traditional property-based testing is when few of the generator-generated inputs satisfy $P(x)$. One way to solve this problem is to do whitebox analysis [81, 47, 78] of the generator and/or the implementations of $P(x)$ and $Q(x)$. A constraint solver can be used to generate inputs $x \in \mathcal{X}$ that are guaranteed to satisfy $P(x)$, which also exercise different code paths within the implementation of $Q(x)$ [164]. Another approach is to collect code coverage during test execution [112]. This information can be used in an evolutionary algorithm to generate inputs that are likely satisfy $P(x)$, while optimizing to increase code coverage through

$Q(x)$. Chapter 6 presents a similar code-coverage-based approach; a blackbox approach is discussed in Chapter 7.

Grammar-based fuzzing [130, 170, 62, 40] techniques rely on context-free grammar specifications to generate structured inputs. Godefroid et al. [80] use grammars to build symbolic constraints at the level of grammar elements, resulting in much higher coverage of the programs under test than regular *symbolic execution* (discussed in the next subsection). LangFuzz [97] generates random programs using a grammar and by recombining code fragments from a codebase, a mixture of generational and mutational fuzzing. The PEACH fuzzer [17] allows for more effective fuzzing of programs expecting structured inputs by randomly sampling inputs according to complex models of common file formats.

CSmith [181] generates random C programs for differential testing of C compilers. The generator of these programs is highly optimized to prevent the generation of C programs with undefined behavior, though it requires heavy duty dynamic checks to prevent all unsafe programs from being generated. YARPGen [122], yet another random generator of C and C++ programs, does not rely on these dynamic checks. It carefully separates variables into different categories (read-only, write-only, read-write). Then it interleaves a type-checking static analysis with the generation process. This analysis is bottom-up, identifying sub-expressions with undefined behavior and mutating them to remove the undefined behavior.

Sulley [28] and BooFuzz [154] allow users to effectively fuzz protocols, based on specifications of those protocols. These specifications must be user-provided. LZFuzz [48] tries to automatically learn some of the structure of inputs for unknown protocols. It uses an adapted compression algorithm to identify blocks in the inputs, and leverages this learned structure for fuzzing.

Several approaches have also looked at leveraging input structures to improve the performance of coverage-guided fuzzing. For instance, libprotobuf-mutator [168] extracts high-level mutation operations from protocol buffer specifications of inputs. AFLSmart [158] uses PEACH specifications of inputs to get higher-level mutation operators, as well as validity feedback to guide CGF to generating inputs that fully parse.

Superion [176] uses grammars to enhance the mutation strategies of AFL—including a LangFuzz-like strategy—and add a structured trimming (i.e. minimization) phase that works on the AST level. Nautilus [30] also enhances coverage-guided fuzzing with grammars. It leverages the grammar to (1) generate inputs from scratch which cover diverse aspects of the grammar; (2) minimize inputs; (3) mutate inputs at the AST level. It introduces 4 different AST-based mutation strategies as well as retaining some AFL mutations on the leaves.

Another direction is to learn some version of the input structure, rather than assume the user provides a specification of input structure. LZFuzz [48], which we already discussed, does this for network protocols. GRIMOIRE [41] learns something close to a context-free grammar while fuzzing inputs. Essentially it learns a single-level hierarchy grammar, by figuring out which parts of the input can be modified while still maintaining the newly-discovered branch which made the input interesting to save. DIFUZE [63] infers device driver interfaces from a running kernel to bootstrap subsequent structured fuzzing. Learn&Fuzz [83] uses a sequence-to-sequence model to learn PDF objects, then leverages this model to generate new

PDF documents with different objects. Interestingly, this model-based approach resulted in lower overall coverage, likely due to AFL's coverage of error states.

There are also some recent works that focus more on the grammar learning rather than fuzzing. GLADE [37] uses an iterative approach and repeated calls to an oracle to learn a context-free grammar for a set of inputs. The first phase of this learning generalizes each input as a regular expression; a second phase merges learned substructures of the regular expression. The learned grammars are not meant to be human-readable, but can be used to sample inputs for fuzzing.

Others approaches use whitebox or greybox information about the program under test to learn grammars. Lin et al.'s work examines execution traces in order to reconstruct program input grammars [120, 119]. AUTOGRAM [99] tracks input flows into variables, and uses this dataflow information to learn a well-labeled grammar. Mimid [87] goes a step further, tracking the control-flow nodes in which input characters are accessed. It directly maps this control-flow structure to the grammar structure, and takes advantage of function names in order to sensibly label nonterminals.

2.6 Other Approaches to Automated Testing

The core idea behind symbolic execution [59, 104] is to generate inputs by reasoning about the *path constraints* of the program under test. A *path constraint* is the conjunction of logical constraints that must be satisfied for an input to follow a certain path through a program. Advances in SMT solvers have made these methods viable in practice [65, 36, 66].

Traditionally the term symbolic execution is used to refer to methods that collect the whole set of path constraints at the same time. KLEE is the classic example of such a symbolic execution engine [51]. KLEE works by analyzing the execution of inputs at the LLVM bytecode level. Each time it hits a branch statement, it adds the two alternative paths to the state space of the program. To explore the state space efficiently, it uses either random sampling of paths, or tries to explore paths that are more likely to result in increased coverage. One of the big problems with this full exploration is that the space of path constraints grows exponentially in the number of branches; this is called the path explosion problem.

Concolic execution or dynamic symbolic execution [165, 81] works slightly differently. It starts with a concrete input and runs it through the program, collecting the constraint of the path it follows. Then it flips the last branch in that path which has not been fully explored, and generates an input that solves that path constraint. If all branches are flipped eventually, this will ensure full coverage of the program under test. Nonetheless, it still suffers from the aforementioned path explosion problem. MultiSE [166] uses an alternative representation of the search space—value summaries—that enables it to incrementally merge the state space, alleviating some of these problems.

The term whitebox fuzzing [82] generally refers to input-generation techniques that utilize constraint solving to generate inputs, but explore the path constraint space in a more random manner. SAGE [82] is the most well-known whitebox fuzzer. As in concolic execution, it starts with a concrete path through the program. However, instead of just flipping the last constraint in the path, it tries to flip each constraint individually, generating a new child input for each constraint. Ideally, this process would be repeated for the child inputs, but this too explodes the search space. So SAGE uses the heuristic of prioritizing the expansion of child inputs which result in the largest increase in basic block coverage.

Mayhem's [52] goal is to find exploitable bugs in program binaries. It switches between a concrete executor client and a symbolic execution server to generate inputs exposing potentially exploitable bugs. The concrete execution side performs dynamic taint tracking, and when it finds a branch condition or jump target which is tainted, it switches control to the symbolic execution side. The idea is that if a jump target is tainted by the input, it could potentially be used maliciously by an attacker to cause a jump to unverified code. The symbolic execution engine then tries to determine whether the branches sent to it from the concrete executor are feasible. In addition to checking path constraints, Mayhem also keeps track of an exploitability formula when hitting tainted jump conditions. If this formula is satisfiable, then the input satisfying it will be an exploit, a distinctive feature of Mayhem.

There has been a recent resurgence of interest in building more scalable symbolic execution engines from the security community. SymCC [159] is a drop-in replacement to the `clang` or `clang++` compiler, which builds concolic execution into the binary. The concept is similar to that in CUTE [165]; during compilation, instructions are added to the binary which, when executed, build up the symbolic constraints. The important point is that this reasoning about building up symbolic path expressions is only conducted at compile time, as opposed to interpreter-based symbolic executors [56, 169]. SymQEMU [160] achieves similar benefits, but without requiring access to the source code of the program under test. They use the Tiny Code Generator (TCG) component of QEMU [39] to do this. In regular QEMU, this component translates the binary to TCG operations before compiling them to the target machine code. During this process, SymQEMU also adds TCG operations that will, when executed, build up the symbolic constraints.

The key advantage of the constraint solver based techniques is that they can easily get through "hard" constraints, a key problem for coverage-guided mutational fuzzers. Several works have looked at integrating symbolic execution and coverage-guided mutational fuzzing to get the best of both worlds. Driller [172] runs coverage-guided fuzzing until it gets "stuck" in certain component, i.e. has not discovered new branches in a certain function for a while. At this point, it invokes concolic execution on the saved inputs for that component. In only tries to flip constraints that have not been previously exercised by a saved input. Munch [142] also orchestrates symbolic execution (KLEE) and AFL. It can run in two hybrid modes. If seed inputs are available, it runs the FS hybrid mode: run AFL for some time, then use KLEE to generate inputs that reach function not reached by AFL. It saves some of the symbolic execution effort by excluding paths to functions explored by AFL already. Otherwise, it runs the SF hybrid mode: start by generating seed inputs via symbolic execution, then run fuzzing.

Before describing QSym [184], Yun et al. evaluate the actual performance costs of hybrid (symbolic + coverage-guided) fuzzing. They find that the path explosion problem is not the only bottleneck to performance. In particular, (1) they help reduce the cost of emulation by writing a concolic executor at the x86 instruction level rather than the LLVM IR level; (2) they get rid of some snapshotting since the goal is to use concolic execution to get through a given hard branch; and (3) they reduce the cost of constraint solving by only trying to flip the target branch, rather than trying to keep an exact path prefix. This resulted in the discovery of new bugs in software that had been heavily fuzzed by coverage-guided fuzzing.

Another approach to smarter fuzzing is to find locations in seed inputs related to likely crash locations in the program and focus mutation there. BuzzFuzz [77] starts from a set of seed inputs and runs them through the program. It conducts dynamic taint tracking to figure out which bytes of the input flow into dangerous locations—e.g. the input of a `malloc` call. Then it creates mutant inputs by setting those bytes to extremal values. Similarly, TaintScope [177] tracks which bytes flow to security-sensitive operations and focuses mutations on these bytes. In addition, it automatically identifies checksums in the program under test, and bypasses these during fuzzing. If an interesting input is found which bypasses these checksums, it uses dynamic symbolic execution to repair the checksum.

Dowser [92] focuses on finding inputs which cause buffer overflows. It identifies locations in the program where buffer overflows may occur, in particular, loops containing pointer dereferences. It then conducts taint-tracking to determine which bytes flow to these target locations. Then it conducts partial dynamic symbolic execution, treating only those bytes that flow to the target locations as symbolic.

While coverage-guided fuzz testing emerged from the security community, the idea of using genetic algorithms for testing has been long explored in the software engineering community [137, 107]. The field of search-based software testing [131, 91, 90, 182] uses optimization techniques such as hill climbing and genetic algorithms to generate inputs that optimize some observable fitness function. These techniques work well when the fitness curve is smooth with respect to changes in the input.

For instance, Sapienz [128] automatically creates test suites for Android applications. It uses proper multi-objective search (i.e. Pareto optimal [93]) to maximize code coverage and number of crashes found while reducing the length of interaction sequences in the test suite. Since the whole test suite is being evolved at a time, the maximization of code coverage is a reasonable approach; coverage-guided fuzzing tools like AFL operate on the single test-case level, and thus must rely on the novelty search approach of saving inputs that find new coverage. Sapienz uses several mutation operators specialized to the Android domain in order to optimize the search.

Finally, another direction in automated testing is to automatically create test cases rather than inputs to a test driver. Randoop [147] and EvoSuite [74] automatically produce JUnit tests for a particular class by incrementally trying and combining sequences of method invocations on the component classes. During the generation of sequence of calls, both Randoop and EvoSuite take some form of feedback into account. For instance, Randoop avoids extending call sequences that led to exceptions. EvoSuite uses a genetic algorithm to evolve a test suite using code coverage as a fitness function.

Part I

Generalized Feedback-Directed Fuzzing

Chapter 3

PerfFuzz: Multi-Objective Performance Fuzzing

Coverage-guided fuzzing tools such as AFL and libFuzzer find a variety of bugs in software. They particularly excel at exposing memory management issues in C/C++, or bugs that can be exposed by the `clang` sanitizers [21, 22, 23, 24].

This chapter presents a fuzzing algorithm, PerfFuzz, that finds inputs exposing a different type of software issue: performance problems. On top of being difficult to detect and fix [103], performance problems can lead to security vulnerabilities. An algorithmic complexity vulnerability [64, 1, 2, 4, 5, 20] can be leveraged by an attacker to cause denial-of-service (DoS) attacks on a deployed service [3].

There exist a number of techniques to help developers diagnose and fix performance performance problems [88, 139, 34, 141, 171], but almost all of these techniques require the execution of the program on test inputs. Traditional sources for such inputs include (1) specially hand-crafted performance tests [141, 138], (2) standardized benchmark suites [34, 35, 60], (3) inputs that are commonly encountered in normal program usage (sometimes called *representative workloads*) [84, 190], or (4) inputs sent by users experiencing performance problems [171]. However, these test inputs may either not expose performance problems, or only be obtainable after some damage has already.

PerfFuzz's goal is to automatically find inputs exposing performance problems, enabling developers to diagnose and fix these problems before the deployment of software. In particular, PerfFuzz aims to automatically find what we call *pathological* inputs. *Pathological inputs* are those inputs which exhibit worst-case algorithmic complexity in different components of the program. For example, a program may use data structures such as hash tables and sorting algorithms such as quicksort. Pathological inputs would be those which, when executed, lead to many collisions in the hash table or many swaps in the sorting routine. Such pathological inputs can be identified as those which, given a fixed input length, maximize the execution count of a particular program component.

In this chapter, we first motivate the algorithm behind PerfFuzz: a fuzzing algorithm to perform multi-objective maximization. Then we present a formal description of the

```c
 1 // Hash-map entry; also a linked list
 2 // node, to resolve hash collisions
 3 typedef struct entry_t {
 4   char* key;
 5   int value;
 6   struct entry_t* next;
 7 } entry;
 8
 9 // Table of hash-map entries.
10 const int TABLE_SIZE = 1001;
11 entry* hashtable[TABLE_SIZE] = {0};
12
13 // Computes a hash value for a word.
14 unsigned int compute_hash(char* str) {
15   unsigned int hash = 0;
16   for (char* p = str; *p!='\0'; p++) {
17     hash = 31 * hash + (*p);
18   }
19   return hash % TABLE_SIZE;
20 }
21 // Increments word count in the hash-map
22 void add_word(char* word) {
23   // access the appropriate bucket
24   int bucket = compute_hash(word);
25   entry* e = hashtable[bucket];
26
27   // find matching entry
28   while (e != NULL) {
29     if (strcmp(e->key, word) == 0) {
30       // increment count
31       e->value++;
32       return;
33     } else {
34       // traverse linked list
35       e = e->next;
36     }
37   }
38   // If no entry found, create one
39   hashtable[bucket] = new_entry(word,
40           1, hashtable[bucket]);
41 }
```

Figure 3.1: Extract from a C program that counts the frequency of words in an input string.

algorithm, highlighting the differences between PerfFuzz and traditional coverage-guided fuzzing. While the formal algorithm works on an abstract notion of "performance feedback", we implement the algorithm with a focus on finding pathological inputs. We find that the implementation performs better than single-objective performance fuzzing and coverage-guided fuzzing in terms of finding pathological inputs and algorithmic complexity issues.

The next chapter revisits the more general algorithm, and provides a framework to enable users to utilize it to achieve their own domain-specific testing goals.

3.1 Motivation

The C program in Figure 3.1 is a simplified version of `wf` [6], a simple word frequency counting tool packaged in the Fedora 27 RPM repository. The main program driver (omitted for brevity) splits its input 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 22–41). This function first computes a hash value for that word—implemented in `compute_hash` (Lines 14–20)—and then attempts to find an 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 run on inputs consisting of 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. In this case, the program spends 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.

With access to inputs illustrating these bottlenecks, a developer could run the program through a standard profiling tool such as GProf [88] or Valgrind [139] and identify where the program spends most of its time. Most of the time, unfortunately, a developer does not have access to such inputs—but PerfFuzz can help.

The goal of PerfFuzz is to *automatically* generate inputs showing a variety of performance behaviors in the program under test. In particular, it aims to generate a set of inputs, each of which *maximize* the execution count of different edges in the control-flow graph (CFG) of a program. PerfFuzz accomplishes this via a feedback-directed fuzzing loop, similar to the coverage-guided fuzzing introduced in Section 1.1. It works, essentially as follows:

1. Initialize a set of inputs, called the *parent inputs*, with some given seed inputs.
2. Pick a parent input that maximizes 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*.
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.

To provide some intuition for *why* this algorithm works, consider running the PerfFuzz algorithm for the word frequency counting program `wf` shown in Figure 3.1.

Suppose we start with the seed input "`the quick brown fox jumps over the lazy dog`". 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. Consider, the result of a few different mutations:

M1. 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.

M2. 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.

M3. 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.

As in coverage-guided fuzzing, saved inputs are added to the parent set, and a new parent input is selected for mutation. Saved inputs that maximize the execution count of at least one CFG edge are *favored*; that is, they are picked as parents for fuzzing with higher probability. A favored input will cease to be favored when new 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 various edges in the program—the same favored input may maximize the execution counts of correlated 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 3.1 for an example). For the running example, PerfFuzz outputs strings including

```
tvÇ1PFEj??A4A+v!^?^AE!§^?MPttò8dg80ÿ(8mrÿÿÿÿ,
```

a single long word, which maximizes the execution count of Line 17 in `compute_hash`; and `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 3.4.1.2 includes a detailed report on 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. For instance, the mutation **M3** actually *reduces* the total number of words, resulting in a smaller path length, but is nonetheless saved because it increases the hit counts of the linked-list traversal edges.

This is in contrast to previous tools which use a greedy approach and consider only increases in total path length [157]. This feature helps PerfFuzz find inputs exercising worst-case behavior even when the performance response of the program is non-convex. The hash collision example illustrates such non-convexity: a first hash collision decreases the path length, but the path length will become much longer in the presence of multiple collisions. Our empirical evaluation supports the importance of this multi-objective approach.

3.2 The PerfFuzz Algorithm

We now describe the PerfFuzz technique formally. Algorithm 3 outlines the high-level input generation strategy. The high-level algorithm is based on the coverage-guided fuzzing algorithm; the grey boxes highlight the key additions to Algorithm 1.

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. The PerfFuzz algorithm can be easily adapted to maximize a variety of values for different program components: the number of bytes allocated at `malloc` statements, the number of cache misses or page faults at memory load/store instructions, the number of I/O operations across system components, etc. Chapter 4 shows this concretely.

PerfFuzz is given a program, p, and a set of initial seed inputs (S_0). These seed inputs are used to initialize a set of *parent inputs*, denoted $\mathcal{S}$ (Line 1). Inputs in set $\mathcal{S}$ form the base from which new inputs are generated via mutation.

PerfFuzz then considers each input from the set $\mathcal{S}$ (Line 4) and probabilistically decides whether or not to select that input for mutational fuzzing (Line 5). The selection probability fuzzProb is high for an input that is currently *favored* (maximizes a performance value, detailed in Definition 5) and low otherwise.

Algorithm 3 The PerfFuzz algorithm; differences from the coverage-guided fuzzing algorithm are highlighted in grey.

Inputs: program p, set of initial seed inputs S_0

```
1:  $\mathcal{S} \leftarrow S_0$
2:  $totalCoverage \leftarrow$ initCoverage($S_0$)
3:  repeat                                                    ▷ begin a cycle
4:      for $input$ in $\mathcal{S}$ do
5:          with probability fuzzProb($input$) do
6:              for $1 \leq i \leq$ numChildren($p$, $input$) do
7:                  $input' \leftarrow$ mutate($input$)
8:                  $coverage$, $perfmap_{input'} \leftarrow$ Execute($p$, $input'$)
9:                  if $coverage \not\subseteq totalCoverage$ then
10:                     $\mathcal{S} \leftarrow \mathcal{S} \cup \{i'\}$
11:                     $totalCoverage \leftarrow totalCoverage \cup coverage$
12:                 if hasNewMax($input'$, $perfmap_{input'}$) then
13:                     $\mathcal{S} \leftarrow \mathcal{S} \cup \{i'\}$
14: until given time budget expires
```

Each time a parent input is chosen for fuzzing, PerfFuzz determines a number of new child inputs to generate (Line 6). It generates these children by mutating the chosen parent input (Line 7). PerfFuzz then executes the program under test with every newly generated child input (Line 8). During the execution, PerfFuzz collects feedback which includes code coverage information (e.g., *which* CFG edges were executed) as well as values associated with the program components of interest (e.g., *how many times* each CFG edge was executed). If an execution results in new code coverage (newCov) or if it maximizes the value for some component (newMax), then the corresponding input is added to the set of parent inputs for future fuzzing (Line 10, Line 13). 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.

Once PerfFuzz completes a full cycle through the set $\mathcal{S}$, it simply repeats this process until a given time budget expires (Line 14).

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 hasNewMax) and for an input to be *favored* and thus selected for fuzzing (fuzzProb).

Definition 1. A *performance map* is a function $perfmap : \mathcal{K} \to \mathcal{V}$, 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. As outlined earlier, the sets $\mathcal{K}$ and $\mathcal{V}$ have deliberately been left abstract to make the algorithm flexible to different *(program component, performance value)* pairs.

Definition 2. The *cumulative maximum map* at time step t is a function $\mathit{cumulmax}_t : \mathcal{K} \to \mathcal{V}$. 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:

$$\forall k \in \mathcal{K} : \mathit{cumulmax}_t(k) = \max_{i \in \mathcal{I}_t} \mathit{perfmap}_i(k).$$

The first key to the PerfFuzz algorithm is saving inputs which achieve a new maximum compared to previously observed values (Line 13). In terms of *cumulmax*, an input has a new maximum if:

Definition 3. The function hasNewMax will return `true` for a newly generated input i at time step t if the following condition holds:

$$\exists k \in \mathcal{K} \text{ s.t. } \mathit{perfmap}_i(k) > \mathit{cumulmax}_t(k).$$

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

Definition 4. 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:

$$\mathit{maximizes}_t(i, k) \Leftrightarrow \mathit{perfmap}_i(k) = \mathit{cumulmax}_t(k).$$

Definition 5. An input i is *favored* for fuzzing at time step t if and only if it maximizes a performance value for some component:

$$\mathit{favored}_t(i) \Leftrightarrow \exists k \in \mathcal{K} \text{ s. t. } \mathit{maximizes}_t(i, k)$$

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.

In addition, we would like to make sure that inputs do not remain favored if they maximize some key k whose value is unlikely to increase; for example, a CFG edge at the start of `main` that can only be executed once by an input. We characterize this via *staleness*.

Definition 6. Let $\mathcal{P}_t$ be the list of parent inputs that have been selected for mutation as of time t. The *staleness* of a key k at time t is defined as

$$\mathit{staleness}_t(k) = \left|\{ß \in \mathcal{P}_t : \mathit{perfmap}_i(k) = \mathit{cumulmax}_t(k)\}\right|.$$

That is, the staleness of a key k increments any time an input maximizing its current *cumulmax* value has been selected as parent for mutation, *but none of its mutants increase the value at k.* After a mutant i' with $\mathit{perfmap}_{i'}(k) > \mathit{cumulmax}_t(k)$ is found, the staleness of k is 0. On the other hand, for a key that is executed once in every execution of an input, its staleness at time t will be the total number of inputs that have been selected as parents for mutation in Line 5. We use this to compute staleness score for inputs:

Definition 7. The *staleness score* of an input i at time t is defined as follows. Let $\mathcal{K}_i$ be the $k \in \mathcal{K}$ such that $\mathit{perfmap}_i(k) > 0$. Then:

$$\mathit{staleness\_score}_t(i) = \frac{\min_{k \in \mathcal{K}_i} \mathit{staleness}_t(k) - \min_{k \in \mathcal{K}} \mathit{staleness}_t(k)}{\max_{k \in \mathcal{K}} \mathit{staleness}_t(k)}$$

The intuition behind this score is that an input which only maximizes the value for the most stale key k will have a staleness score near 1; one maximizing a not-stale key k will have a staleness score near 0. However, if all keys k are somewhat stale, any input exercising a key k with minimum staleness will also have staleness score of zero.

Finally, we can define the probability that an input will be selected as a parent for fuzzing:

Definition 8. The *selection probability* of an input i at time t is:

$$\textsc{fuzzProb}_t(i) = \begin{cases} 1 - \sigma \cdot \mathit{staleness\_score}_t(i) & \text{if } \mathit{favored}_t(i) \\ \alpha \cdot (1 - \sigma \cdot \mathit{staleness\_score}_t(i)) & \text{otherwise} \end{cases}.$$

That is, favored inputs with staleness score of 0 are always selected, and inputs with a higher staleness score are less likely to be selected for mutation. α is the base probability of selecting a non-favored input, and stale inputs are similarly de-prioritized. In our experiments we use $\alpha = 0.01$ and $\sigma = 0.8$. In our evaluation of the impact of staleness, we will compare this to the selection $\alpha = 0.01$ and $\sigma = 0$; on our benchmarks, the impact of staleness on our results is mixed.

3.3 Implementation

PerfFuzz is built on top of American Fuzzy Lop (AFL) [185], and inherits many of its implementation details. Notably, the number of child inputs to produce (Line 6 in Algorithm 3, the mutations performed (Line 7 in Algorithm 3), and the notion of new coverage (Line 9 of Algorithm 3) are borrowed directly from AFL. In our evaluation, we enable only the "havoc" mutation stages of AFL. Refer to Section 2.1 for a detailed discussion.

Note that although the hit count of a CFG edge/branch is considered in AFL's notion of coverage (Section 2.1), an input that achieves new coverage may not have a new maximum and vice versa. For example, let e represent a CFG edge. An input hitting e 10 times when e has only been hit 20 times by previously generated inputs achieves new coverage but not a new maximum. On the other hand, an input hitting e 190 times when e has already been hit 130 times achieves a new maximum but not new coverage.

Performance map In the implementation evaluated in the chapter, 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.

3.4 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 [157]?

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

RQ3. How does staleness impact the performance of PerfFuzz?

RQ4. Does the multi-dimensional objective of PerfFuzz help find a range of inputs that exercise distinct hot spots?

We chose four real-world C programs 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 [167]. The availability of good `libFuzzer` drivers was key to being able to fairly compare PerfFuzz to SlowFuzz [157] in Section 3.4.1. 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 [26]) 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 SlowFuzz 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 PerfFuzz and SlowFuzz are non-deterministic algorithms, 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 3.4.1.2), 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 [50] using the PCRE library, and (3) `wf-0.41` [6], a simple word-frequency counting tool found in the Fedora 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 9. The *maximum path length* is the longest execution path across all inputs generated so far.

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

Definition 10. The *maximum hot spot* is the highest execution count observed for any CFG edge across all inputs generated so far.

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

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.

3.4.1 Comparison with SlowFuzz

SlowFuzz [157] 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, PerfFuzz produces many (typically at least thousands, often tens of thousands) of inputs from one chosen parent input (Line 6 of Algorithm 1). SlowFuzz instead produces one mutant for each parent. This reduces the importance of selecting inputs to fuzz. Thus, while PerfFuzz prioritizes inputs to fuzz using the concept of *favored* inputs (Line 5 of Algorithm 1), SlowFuzz randomly selects a parent input to fuzz. Second, PerfFuzz applies AFL's *havoc* mutations (as detailed in Section 3.3) 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` [167], an LLVM-based fuzzing tool. 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 [157], and we found that it performed best on a selection of micro-benchmarks in our initial experiments.

3.4.1.1 Maximizing Execution Counts

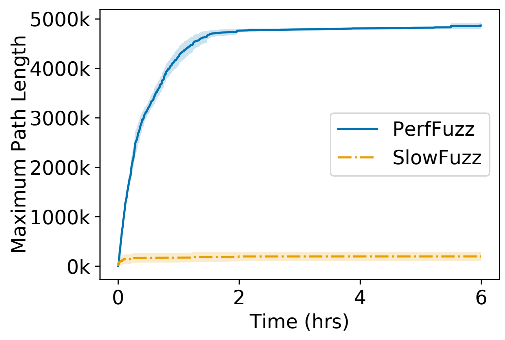
Figure 3.2 shows the progress made by PerfFuzz and SlowFuzz during 6-hour runs in maximizing total path length (on the left) and the maximum hot spot (on the right). The lines in the plot represent average values over 20 repeated 6-hour runs, while the shaded areas represent 95% confidence intervals, calculated with *Student's t-distribution*.

It is clear from Figure 3.2 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×–24.7× higher and the maximum hot spots are 5×–69× higher. This is in spite of the fact that SlowFuzz produces more inputs in each of this 6-hour runs (from 1.7× more for `libxml2` to 17.7× more for `libjpeg-turbo`).

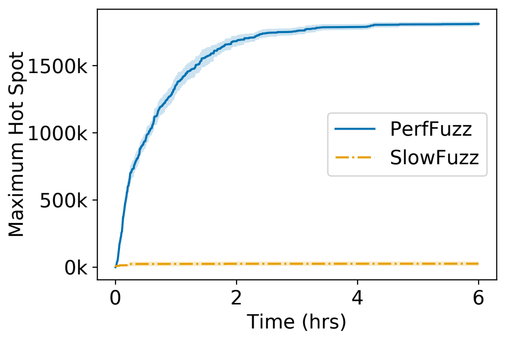
The results show that not only is PerfFuzz better than SlowFuzz at finding hot spots, for which the PerfFuzz algorithm is tailored, but that PerfFuzz is superior to SlowFuzz even for finding inputs that maximize total path length, for which SlowFuzz is tailored. Intuitively, we believe that this is because 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.

3.4.1.2 Algorithmic Complexity Vulnerabilities

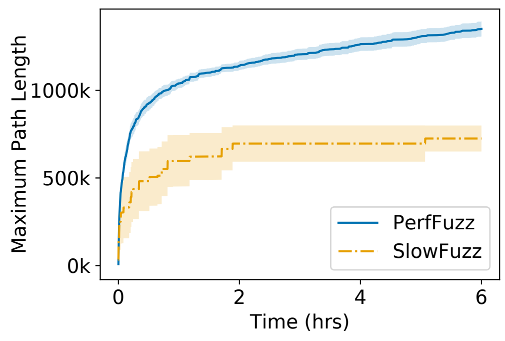
SlowFuzz was designed to find algorithmic complexity vulnerabilities, where programs exhibit worst-case behavior that is asymptotically worse than their average-case behavior. Such



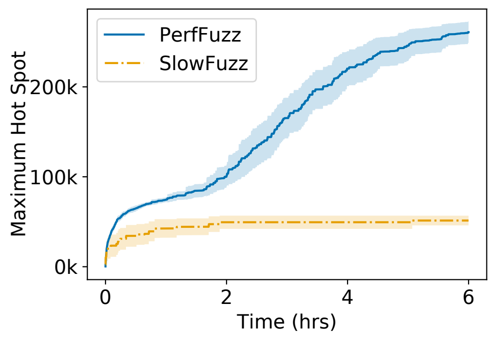
(a) libpng - max. path length



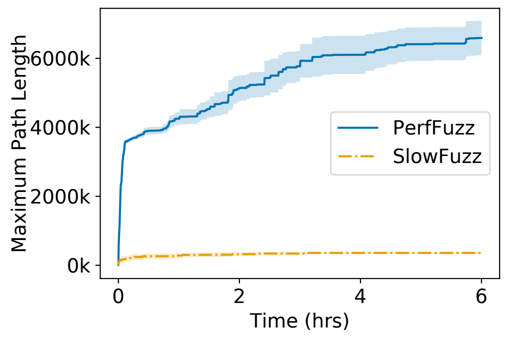
(b) libpng - max. hot spot



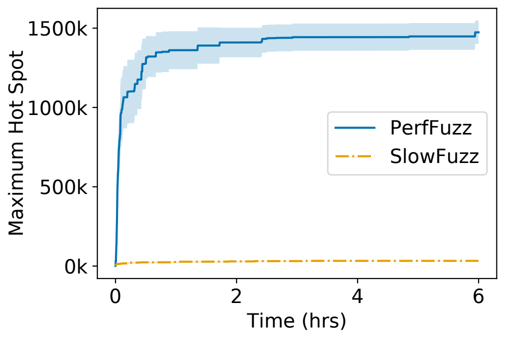
(c) libxml2 - max. path length



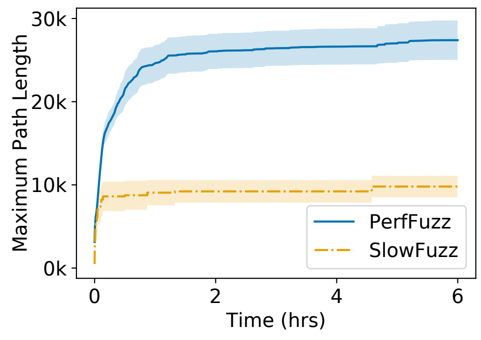
(d) libxml2 - max. hot spot



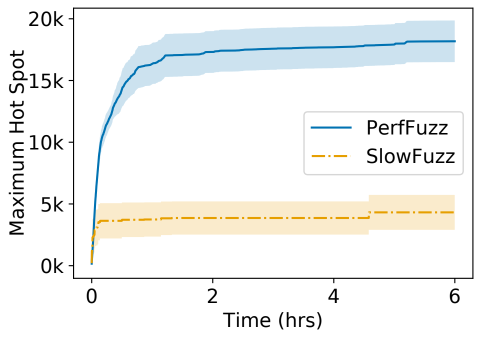
(e) libjpeg - max. path length



(f) libjpeg - max. hot spot

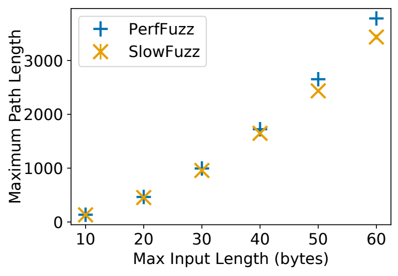


(g) zlib - max. path length

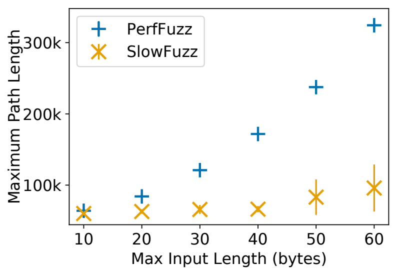


(h) zlib - max. hot spot

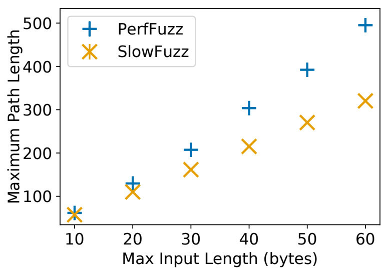
Figure 3.2: PerfFuzz vs. SlowFuzz on macro-benchmarks: maximum path length and maximum hot spot found throughout the duration of the 6-hour fuzzing runs. Lines and bands show averages and 95% confidence intervals across 20 repetitions; higher is better.



(a) Insertion Sort



(b) PCRE URL regex



(c) wf

Figure 3.3: PerfFuzz vs. SlowFuzz on micro-benchmarks: maximum path length found with given time budget, for varying input sizes; higher is better.

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 [64]. We now show that PerfFuzz also addresses this use case, and in fact out-performs 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 3.3 shows the results of these runs: points plot the average maximum path length, while lines show 95% confidence intervals.

For insertion sort, for all input lengths, PerfFuzz found a significantly (at 95% confidence) longer maximum path length, but as Figure 3.3a 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 3.3a 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 3.3b 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 an input found by PerfFuzz that had maximum path length in one of the 50-byte runs was:

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

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 3.3b 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 3.1). Figure 3.3c 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:

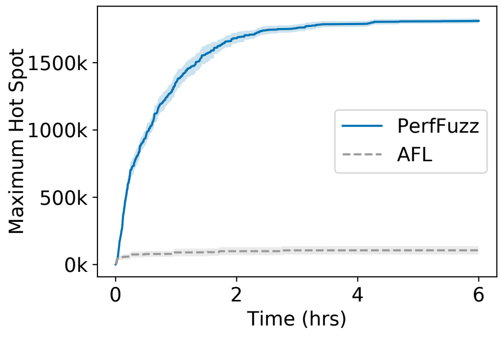
```
t <81>v ^?@t <80>!^?@t <80>!t t^Rn t t t t t t t t t
```

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.

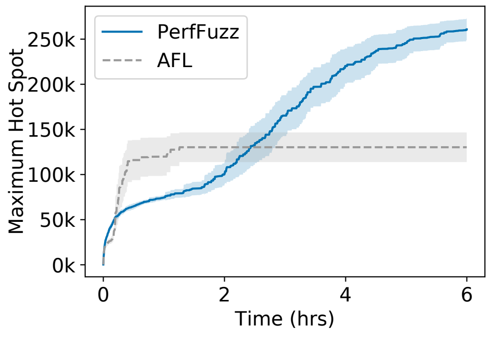
Overall, we see that in the same time constraints, PerfFuzz is able to find inputs with significantly longer paths than SlowFuzz, and can out-perform SlowFuzz in discovering inputs exercising near worst-case algorithmic complexity.

3.4.2 Comparison with Coverage-Guided Fuzzing

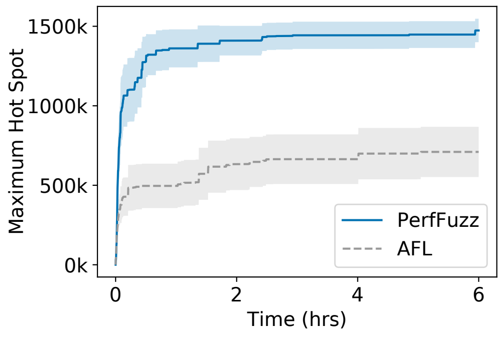
With the insight that PerfFuzz's efficacy is in part due to its multi-objective. coverage-guided progress, we ask whether PerfFuzz performs better than just AFL off-the-shelf. To evaluate this aspect, we ran AFL on our four C macro-benchmarks. Like PerfFuzz, AFL was configured to use only havoc mutations (`-d` option), because this configuration has been



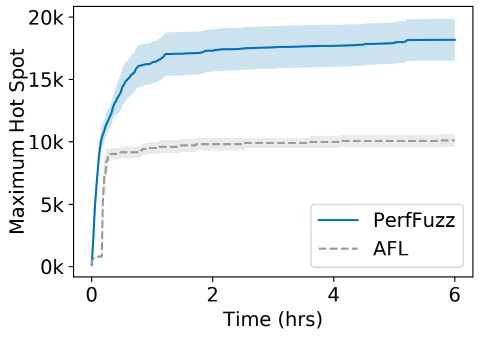
(a) libpng



(b) libxml2



(c) libjpeg



(d) zlib

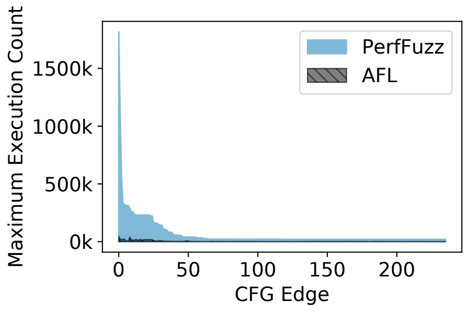
Figure 3.4: PerfFuzz vs. AFL: Time evolution of the maximum hot spot through the 6-hour runs. Lines and bands show averages and 95% confidence intervals across 20 repetitions. Higher is better.

shown to result in faster program coverage [188]. This experiment tests the value-add of PerfFuzz's performance maps and maximizing-input favoring heuristics.

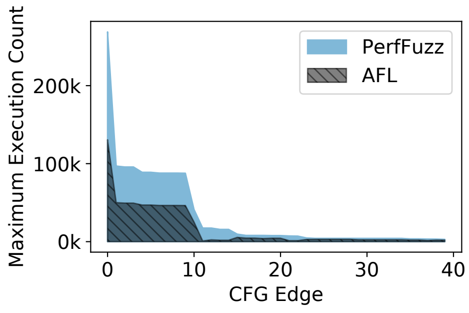
We begin by looking at the evolution of the maximum hot spot found by each technique through time, shown in Figure 3.4. For the `libpng`, `libjpeg-turbo`, and `zlib` benchmarks (Figures 3.4a, 3.4c, 3.4d), we see that PerfFuzz rapidly finds a hot spot with a significantly higher execution count. For the `libxml2` benchmark (Figure 3.4b), 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× higher execution count after 6 hours. Overall, Figure 3.4 demonstrates that PerfFuzz's performance-map feedback has a significant effect on its ability to generate pathological inputs, exercising hot spots with 2×–18× higher execution counts.

Figure 3.4 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 counts found by PerfFuzz are significantly higher than those found by AFL over all hot spots in the program. Figure 3.5 provides this information.

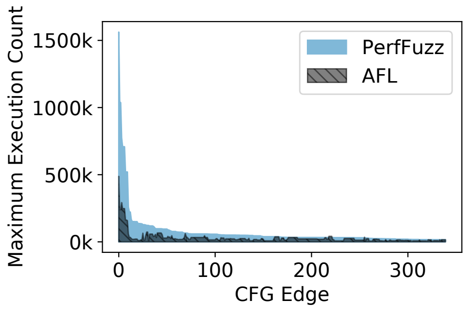
In particular, Figure 3.5 shows the maximum execution count per CFG edge found by each technique at the end of the 6 hour runs. We plot the median of this measure across the 20



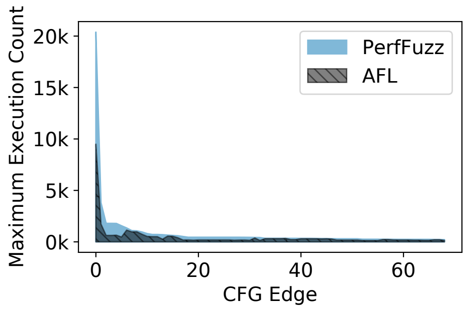
(a) libpng



(b) libxml2



(c) libjpeg-turbo



(d) zlib

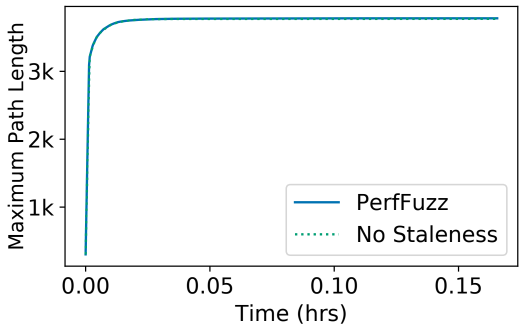
Figure 3.5: Distribution of maximum execution counts across CFG edges found by PerfFuzz and AFL after 6-hour runs. For each edge, the median over 20 runs is plotted.

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 3.5 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 2$\times$ more times than AFL-generated inputs, and 238 edges which PerfFuzz-generated inputs exercise over 10$\times$ more times.

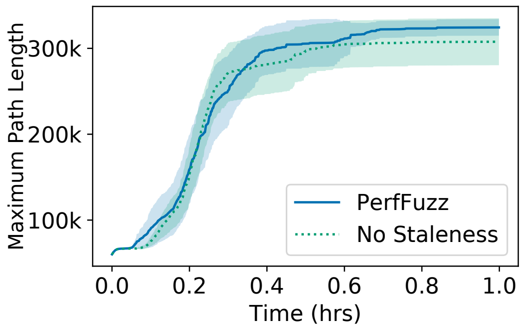
3.4.3 Impact of Staleness

In Section 3.2, we described the notion of *staleness*, which allowed us to de-prioritize inputs for mutation if the performance values they maximize have not been increased in several fuzzing iterations. Note that although the ISSTA'18 paper on PerfFuzz [114] does not describe staleness, all the experiments in that paper were run with this staleness criterion enabled. In this section, we evaluate the impact of this staleness criterion for input selection on a subset of our micro (Insertion Sort, PCRE URL) and macro (libpng, zlib) benchmarks.

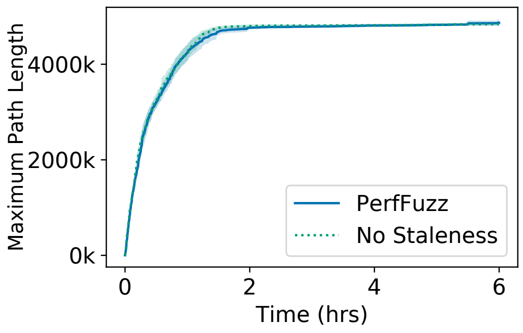
On these benchmarks, we ran PerfFuzz with the default staleness discount factor



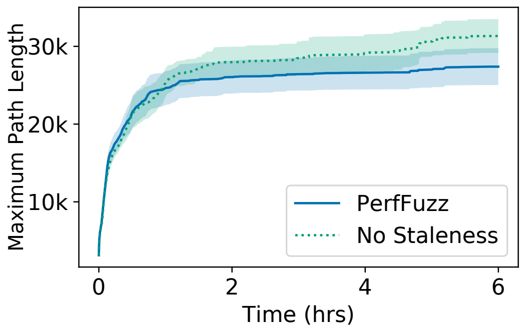
(a) Insertion Sort



(b) PCRE URL



(c) libpng



(d) zlib

Figure 3.6: Impact of staleness on maximum path length found through time. Lines and bands show averages and 95% confidence intervals across 20 repetitions; higher is better.

$\sigma = 0.8$ (a favored input with maximum staleness is selected for mutation 20% of the time) as well as $\sigma = 0$ (a favored input is always selected for mutation, regardless of its staleness). In particular, Figure 3.6 shows the maximum path length of PerfFuzz-generated inputs through time for $\sigma = 0.8$ (labeled "PerfFuzz") and $\sigma = 0$ (labeled "No Staleness"). Again, we ran each configuration 20 times, plotting the means and 95% confidence intervals of maximum path length through time in Figure 3.6.

The results in Figure 3.6 are mixed, and suggest in general that this staleness criterion has minimum impact. On the insertion sort and libpng benchmarks, the performance of the two configurations is nearly identical. On PCRE URL, $\sigma = 0.8$ has a slight edge; on zlib, $\sigma = 0$ has a slightly larger edge; in both cases, the 95% confidence intervals overlap.

Due to the minimal impact and the increased complexity the notion of staleness brings, we suggest that the simplified version of the algorithm, without staleness, be adopted in most cases; see the ISSTA'18 paper for a simplified description [114]. However, if in some contexts, the algorithm consistently prioritizes inputs for mutation with uninteresting performance characteristics, this concept of staleness may have a greater impact.

Table 3.1: 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	
Exec. count	CFG edge	Exec. count	CFG edge	Exec. 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`

3.4.4 Case Studies

PerfFuzz is designed to generate inputs that demonstrate pathological behavior in programs across different program components (in this evaluation, CFG edges). 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 3.1 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.

`libpng` From Table 3.1, 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 3.7, 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 3.1, input #9189 maximizes the number of executions of the inner loop when pixel depth is 1 (Line 3715 of Figure 3.7), 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 3.7), 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 3.1 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.

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

`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.

`zlib` Compared to image formats, the functionality of the `zlib` decompressor is relatively straightforward. This was reflected by the fact that there were very few edges exercised a huge number of times (fewer "hot spots"). Nonetheless, PerfFuzz discovered an input with a compression factor of nearly 126×, whose processing lead to a long execution path.

`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 PerfFuzz uncovers non-trivial hot spots. 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.

3.5 Discussion

Like many other input generation techniques founded in a genetic algorithm-style model, 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 performance bottlenecks in a program or the absolute worst-case behavior for each performance bottleneck it discovers. This is common over most algorithms in this dissertation.

This chapter focused on discovering bottlenecks due to increase in computational complexity; therefore, PerfFuzz measures execution counts of CFG edges instead of total running time. This helps ensure that the 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. The next chapter presents a framework that more easily allows for the creation of a different cost model.

Finally, we believe that the reason that PerfFuzz outperforms greedy techniques such as SlowFuzz is that its multi-objective approach can 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 3.1, we have not mapped the performance spaces of our benchmarks to measure their convexity. Nonetheless, our results suggest that changing the feedback used to save inputs is key to finding different types of bugs. We explore this further in the next chapter.

Chapter 4

FuzzFactory: A Framework for Specialized Fuzzers

The last chapter introduced PerfFuzz, which, by adding a notion of performance feedback to the coverage-guided fuzzing algorithm, was able to consistently find performance issues in programs. While the algorithm presented was generic over (*program component*, *performance value*) feedback maps, the PerfFuzz implementation only produced (*control flow graph edge*, *hit count*) maps. A natural question here is whether there are other performance domains to which the core PerfFuzz algorithm—fuzzing aiming to maximize multiple performance objectives at a time—is applicable.

At the same time, we have seen researchers add other notions of “feedback” to the coverage-guided fuzzing algorithm in order to target various other testing goals. For instance, specialized fuzzers have been built for the purposes of directed testing [45], differential testing [156], finding algorithmic complexity vulnerabilities [157], discovering side-channel attacks [140], finding memory usage bugs [178], generating valid inputs [149, 158, 109], and getting through magic byte comparisons [110, 161, 117].

This observation brings about a further question: is there actually an *even more general* version of PerfFuzz’s multi-objective fuzzing that could cover all these different fuzzing domains? This is exactly what is addressed in this chapter on FuzzFactory.

FuzzFactory is a framework for implementing domain-specific fuzzing applications. At a high level, the term *domain-specific fuzzer* here means a specialized fuzzer whose goal is more precise than the general code-coverage-increase goal of coverage-guided fuzzing. The fuzzers mentioned above with the goals of discovering side-channel attacks, complexity vulnerabilities, memory usage bugs, etc., are instances of domain specific fuzzers.

There are two key parts to the FuzzFactory framework. First is a generalized domain-specific fuzzing algorithm, which effectively extends the PerfFuzz algorithm. The key insight from the PerfFuzz algorithm is that, in order to find inputs showing the worst-case algorithmic complexity in a program, PerfFuzz had to save several intermediate inputs. In particular, it saved inputs which maximized the performance value for some key (Algorithm 3, Line 13)–in particular, increased the hit counts of some CFG edge. In

FuzzFactory we generalize this notion of intermediate input to the notion of *waypoints*, inspired by the corresponding term in the field of navigation. These waypoints give the fuzzing algorithm steps towards a domain-specific goal. We also prove some notion of fuzzing "progress" (Theorem 1), which also applies to the PerfFuzz algorithm.

Second, FuzzFactory provides an interface that allows users—either the developers of a domain-specific fuzzing application or a tester with detailed knowledge of their fuzzing application—to more easily craft these *domain-specific feedback maps*. This interface consists of an API, which, when called, *implicitly* defines the notion of a waypoint input. These API calls form the domain d, which in turn defines the predicate $is\_waypoint(i, \mathcal{S}, d)$. This predicate $is\_waypoint(i, \mathcal{S}, d)$ answers: given the domain d, should a newly generated input i be saved to the set of saved inputs $\mathcal{S}$?

FuzzFactory enables development of domain-specific fuzzing applications without requiring changes to the underlying search algorithm. This enables the rapid prototyping of domain-specific fuzzers. This chapter discusses a domain-specific fuzzing application for exacerbating memory allocations, smoothing hard comparisons, and targeting recently changed code. The OOPSLA'19 paper on FuzzFactory [150] discusses three additional domain-specific fuzzing applications beyond these.

In addition, a key advantage of FuzzFactory is that domain-specific feedback is naturally composable. Combining domain-specific fuzzing applications for exacerbating memory allocations and for smoothing hard comparisons produced a composite application that performs better than each of its constituents. The composite application automatically generates ZIP bombs and PNG bombs: tiny inputs that lead to dynamic allocations of 4GB in `libarchive` and 2GB in `libpng` respectively.

4.1 Motivation

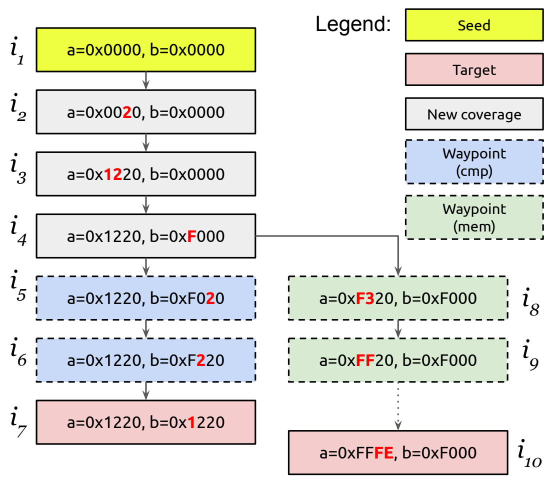
Consider the sample test program in Figure 4.1a. The function `Test` takes as input two 16-bit integers, `a` and `b`. Suppose we perform CGF (Algorithm 1) on this test program. Let us assume that we start with the *seed input*: a=0x0000, b=0x0000. The seed input does not satisfy the condition at Line 2. The CGF algorithm randomly mutates this seed input and executes the test program on the mutated inputs, looking for new code coverage.

Figure 4.1b 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 executes new code, it gets saved to $\mathcal{S}$. In Figure 4.1b, 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 Figure 4.1a. Figure 4.1b shows the corresponding inputs in our example: i_3 and i_4. This is because there are many possible solutions that satisfy the comparisons `a > 0x1000` and `b >= 0x0123`; we call these *soft* comparisons.

```
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 4.1: A motivating example for the FuzzFactory framework.

However, it is much more difficult for CGF to generate inputs to satisfy comparisons like `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 will likely 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.

4.1.1 Waypoints

We can overcome both these challenges—satisfying the comparison at Line 5 and increasing the amount of memory allocated at the statement in Line 8— 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*. The notion of saving inputs that maximize performance values, from the previous chapter, is one such notion of waypoints.

To overcome hard comparisons like `a == b`, we 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 Figure 4.1b 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 `a == b`. Thus, we easily discover an input that triggers `abort` at Line 6 of Figure 4.1a.

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 $\mathcal{S}$. Figure 4.1b shows sample waypoints i_8 and i_9 that may be saved with this `mem` strategy. 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 4.1a. 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 4.4.4, we demonstrate how a combination of these strategies allows us to automatically generate PNG bombs and ZIP bombs, i.e. tiny inputs that allocate 2–4 GB of memory, when fuzzing `libpng` and `libarchive` respectively.

4.2 The FuzzFactory Framework

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.

As we saw in the previous section, and in Chapter 3, coverage guidance alone is not sufficient to achieving domain-specific fuzzing goals. FuzzFactory enables users to prototype fuzzers that target domain-specific goals, by specifying: (1) the feedback to collect from the execution of the program on any input, and (2) how to use this feedback to determine if the input should be considered a waypoint.

We first describe the domain-specific feedback mechanism which allows users to specify the domain-specific feedback they want from an execution. We then explain how this feedback is used to define *is_waypoint*. We also describe how to compose such domain-specific feedback. Finally, we describe the multi-objective fuzzing algorithm, a generalization of Algorithm 3, which takes this feedback into account

4.2.1 Domain-Specific Feedback

FuzzFactory provides 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. We explain the meaning of A, a_0, and $\rhd$ in a user-defined domain in the next subsection. 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—the API calls that can be inserted are given in Section 4.3.

4.2.2 Waypoints

FuzzFactory uses the dsf_i map from the execution of the test program on input i in order to determine if i needs to be saved. In particular, it 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 $v, v' \in V$:

$$a \rhd v \rhd v \quad = \quad a \rhd v \tag{4.1}$$

$$a \rhd v \rhd v' \quad = \quad a \rhd v' \rhd v \tag{4.2}$$

These rules imply idempotence and application-order insensitivity, respectively, in the second operand. For the memory-allocation domain 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 4.1 is satisfied because $\max(\max(a, v), v) = \max(a, v)$ for any $a, v \in \mathbb{N}$. Property 4.2 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; see Theorem 1. 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 4.1 and 4.2.

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:

$$\mathcal{A}(\mathcal{S}, k, d) \overset{\text{def}}{=} a_0 \rhd dsf_{i_1}(k) \rhd dsf_{i_2}(k) \rhd \ldots \rhd dsf_{i_n}(k), \text{ where } d = (K, V, A, a_0, \rhd) \tag{4.3}$$

Due to the Properties 4.1 and 4.2, 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:

$$is\_waypoint(i, \mathcal{S}, d) \overset{\text{def}}{=} \exists k \in K : \mathcal{A}(\mathcal{S}, k, d) \neq \mathcal{A}(\mathcal{S} \cup \{i\}, k, d), \text{ where } d = (K, V, A, a_0, \rhd) \tag{4.4}$$

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.

Note that, 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 4.1 and 4.2 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 1 (Monotonicity of Aggregation). *A domain $d = (K, V, A, a_0, \rhd)$ whose reducer function $\rhd$ satisfies properties 4.1 and 4.2 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:*

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

The interested reader can find the proof of this theorem in Section 4.2.5.

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. Thus:*

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

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

Proof. Follows from the definition of *is_waypoint* in Eq. 4.4 and Theorem 1. □

Algorithm 4 The domain-specific fuzzing algorithm. The grey boxes indicate additions to the standard coverage-guided fuzzing algorithm in Algorithm 1.

```
Input: an instrumented test program $p$, a set of initial seed inputs $S_0$, a set of domain-specific
    feedback $D$
Output: a corpus of automatically generated inputs $\mathcal{S}$
 1: $\mathcal{S} \leftarrow S_0$
 2: $totalCoverage \leftarrow \textsc{initCoverage}(S_0)$
 3: repeat                                                        ▷ Main fuzzing loop
 4:   for $input$ in $\mathcal{S}$ do
 5:     with probability $\boxed{\textsc{fuzzProb}(input)}$ do
 6:       for $1 \leq i \leq \textsc{numChildren}(p, input)$ do
 7:         $input' \leftarrow \textsc{mutate}(input)$
 8:         $coverage,\ \boxed{dsf^1_{input'}, \ldots, dsf^{|D|}_{input'}} \leftarrow \textsc{execute}(p, input')$
 9:         if $coverage \not\subseteq totalCoverage$ then
10:           $\mathcal{S} \leftarrow \mathcal{S} \cup \{input'\}$
11:           $totalCoverage \leftarrow totalCoverage \cup coverage$
12:         if $is\_waypoint(input', \mathcal{S}, D)$ then
13:           $\mathcal{S} \leftarrow \mathcal{S} \cup \{input'\}$
14: until given time budget expires
15: return $\mathcal{S}$
```

4.2.3 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 $d = (K, V, A, a_0, \rhd)$ for each $d \in D$. Then we extend the definition of the predicate *is_waypoint* to D as follows:

$$is\_waypoint(i, \mathcal{S}, D) \overset{\text{def}}{=} \bigvee_{d \in D} is\_waypoint(i, \mathcal{S}, d) \tag{4.5}$$

meaning $is\_waypoint(i, \mathcal{S}, D)$ is true for a set of domains D if and only if $is\_waypoint(i, \mathcal{S}, d)$ 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 $d \in D$.

4.2.4 Algorithm for Domain-Specific Fuzzing

Algorithm 4 describes the domain-specific fuzzing algorithm implemented in FuzzFactory. Overall, observe that Algorithm 4 is a generalization of Algorithm 3 from the previous chapter. Again, the extensions to the conventional coverage-guided fuzzing algorithm (Algorithm 1) 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 $dsf^1_{i'}, \ldots, dsf^{|D|}_{i'}$ 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 $\mathcal{S}$.

In addition, fuzzProb is modified to select inputs for mutation if they have *the most recent aggregate value* for some key in one of the domains in D. If the aggregation function is maximization, this corresponds to the notion of *favoring* from Chapter 3. This is because the newest maximized value *is* the maximum value seen so far. Since $\preceq$ is a partial order, for some other reducers $\rhd$ (e.g. set union), this may not be the optimal notion of favoring.

4.2.5 Proof of Monotonicity of Aggregation

In order to prove Theorem 1, we first need to demonstrate a few lemmas.

Lemma 1 (No ping-pong). *Given a reducer function $\rhd : A \times V \to A$ satisfying Properties 4.1 and 4.2, then $\forall a \in A$ and any $n \geq 0$ terms $v_1, \ldots, v_n \in V$, if $a \rhd v_1 \rhd \ldots \rhd v_n = a$, then:*

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

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 oscillate between distinct values (i.e. *ping-pong*).

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

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

By applying Property 4.2 on the right-hand, we can rearrange terms:

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

Then ,by Property 4.1 on the right-hand side we remove the redundant terms:

$$a \rhd v_1 \rhd \ldots \rhd v_n \neq a \rhd v_1 \rhd \ldots \rhd v_n$$

This is a contradiction; therefore, no such k can exist. □

Definition 11 (Progress). If $\rhd : A \times V \to A$ is a reducer function, then we can define a binary relation $\preceq$ on A called *progress* as follows:

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

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

Proof. Straightforward from Definition 11 with $n = 0$. $\square$

Lemma 3 (Transitivity of progress)**.** *If* $\rhd : A \times V \to A$ *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 11 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:

$$a \rhd u_1 \rhd \ldots \rhd u_m = b \tag{4.6}$$

$$b \rhd v_1 \rhd \ldots \rhd v_n = c \tag{4.7}$$

Substituting the b on the LHS of Equation 4.7 with the LHS of Equation 4.6, we can write:

$$a \rhd u_1 \rhd \ldots \rhd u_m \rhd v_1 \rhd \ldots \rhd v_n = c \tag{4.8}$$

Which, by Definition 11, means $a \preceq c$. $\square$

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

Proof. If $a \preceq b$ and if $b \preceq a$ then by Definition 11 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:

$$a \rhd u_1 \rhd \ldots \rhd u_m = b \tag{4.9}$$

$$b \rhd v_1 \rhd \ldots \rhd v_n = a. \tag{4.10}$$

Substituting the b on the LHS of Equation 4.10 with the LHS of Equation 4.9, 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 4.9, we get $a = b$. $\square$

Proof of Theorem 1. 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 4.3, 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 11, 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 $\preceq$. $\square$

```
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 4.2: API for domain-specific fuzzing in pseudocode.

4.3 Implementation

We implemented the core FuzzFactory algorithm (Algorithm 4) as an extension to AFL [185], and inherits many of its implementation details (as in Section 3.3).

To create domain specific fuzzing maps dsf_i used in this algorithm, the program under test calls some FuzzFactory API functions that manipulate the values in dsf_i. In the applications, we created LLVM instrumentation passes that automatically add instrumentation to populate these dsf_i maps in a structured way. We describe the details of these passes in the corresponding subsections of Section 4.4. However, test programs can also be instrumented using any other tool, such as Intel's Pin [125]. In fact, domain-specific fuzzing applications can also be implemented by manually editing test programs to add code that calls the FuzzFactory API.

Figure 4.2 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 applications where K is a set of program locations $\mathbb{L}$ (Sections 4.4.1, 4.4.2), 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 (Section 4.4.3), 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.

The function `new_domain` returns a handle to the DSF map, which is then used in subsequent APIs listed in Figure 4.2, 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 4.1 and 4.2, which in turn guarantee monotonic aggregation (Theorem 1). API functions that start with '`dsf_`' manipulate the DSF map. The argument `key` must be in the range `[0, key_size)`.

4.4 Evaluation

We demonstrate the applicability of FuzzFactory by instantiating some domain-specific fuzzing applications. For each application, we describe the goals, the instrumentation used to create it, and evaluate its value. This dissertation highlights three of the domain-specific fuzzing applications described in the OOPSLA'19 paper on FuzzFactory [150], in increasing order of complexity:

1. `mem`: An application for generating inputs that maximize dynamic memory allocations.
2. `cmp`: A domain for smoothing hard comparisons. Although a lot of prior work address this application, our particular solution strategy is novel.
3. `diff`: A novel application for incremental fuzzing after code changes in a test program.

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. 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.

Composition A key advantage of FuzzFactory is that it enables us to naturally compose multiple domain-specific fuzzing applications with no extra effort. In Section 4.4.4, 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.

Implementation Traditionally, implementing each such domain would require non-trivial effort in modifying a fuzzing tool such as AFL to achieve a domain-specific objective. With FuzzFactory, implementing the `mem`, `cmp`, and `diff` domains required only 29, 355, and 146 lines of C++ code. These implementations basically consist of creating the LLVM passes that insert calls to the FuzzFactory API in order to create the domain-specific feedback maps for each problem.

Experimental Evaluation For our experiments, we use six benchmark programs from the Google fuzzing test suite [86]. This suite contains specific historical versions of programs that have been thoroughly fuzzed using the OSS-Fuzz infrastructure [85]. The six benchmarks we use include: (1) libpng-1.2.56, (2) libarchive-2017-01-04, (3) libjpeg-turbo-07-2017,

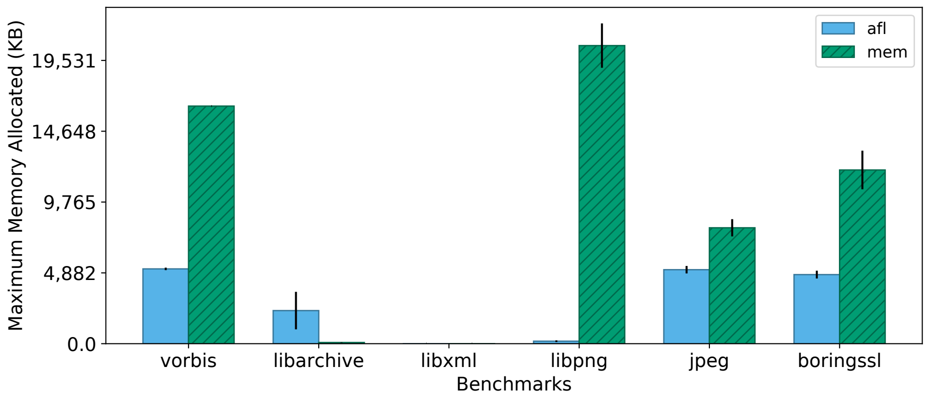


Figure 4.3: Maximum amount of dynamic memory allocated (in KB) due to inputs generated by baseline (afl) and domain-specific fuzzing application (mem). Higher is better.

(4) libxml2-v2.9.2, (5) vorbis-2017-12-11, and (6) boringssl-2016-02-12.[1] The benchmarks are written in C or C++. Benchmarks (1)–(4) were chosen because they are commonly used in the fuzzing literature [114, 115, 153, 54, 55, 158]. Benchmarks `vorbis` and `boringssl` were chosen because they expect markedly different input formats.

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.

4.4.1 `mem`: Exacerbating Memory Allocations

First, we describe `mem`, a application whose goal is to generate inputs that exacerbate memory allocations. There are several use cases for such a fuzzer: 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.

For this memory allocation domain, we have $K = \mathbb{L}, V = \mathbb{N}, A = \mathbb{N}, a_0 = 0,\ a \rhd v = \max(a, v)$. That is, the keys are program locations, and the values are the positive integers, which are aggregated with max. We instrument the test program with calls to `dsf_increment` so that whenever it 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.

[1]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.

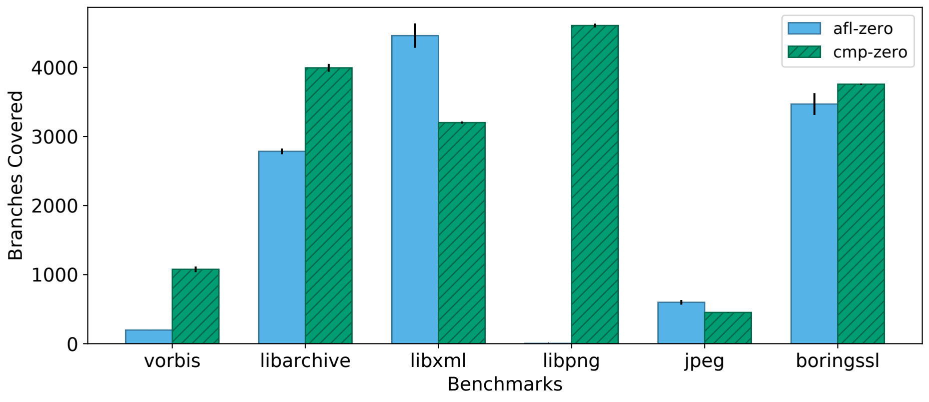


Figure 4.4: Branch coverage, as achieved by inputs generated by baseline (afl-zero) and domain-specific fuzzing application (cmp-zero). The suffix `zero` indicates that seed inputs were simply strings of zeros. Higher is better.

Experimental evaluation Figure 4.3 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 4.4.4, 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`.

Note that the `mem` application was not effective on `libarchive`. This is the only benchmark in our suite where the initial seed input leads to an early exit due to a validation error. Further, `libarchive` contains many "hard" comparisons, like checksum checks. Thus, it is difficult for the fuzzer to get into code where potentially data-controlled memory allocations lie. We will see in Section 4.4.4 that adding feedback to get through these hard comparisons can help us find memory allocation bugs on this benchmark.

4.4.2 `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 4.1, 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 [110, 172, 161, 117, 153, 184]. 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.

Using FuzzFactory, we build a solution that does not rely on the domain knowledge in seed inputs or on expensive symbolic analysis. Instead, we simply instrument all comparison operations in the program, treating locations of these comparisons as our *dsf* keys ($K = \mathbb{L}$), and keeping track of the number of bits in common between the two operands being compared as the value ($V = \mathbb{N}$). The feedback is aggregated using the max reduce operator, with $a_0 = 0$ as usual. 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.

At each comparison location that effectively performs an operation `a == b`, we add a call to `dsf_set` with the value $\mathit{bits\_in\_common}(\mathtt{a}, \mathtt{b})$. In addition to integer equality, we also instrument string comparisons and `switch-case` statements. The inserted code populates the DSF map entries corresponding to their program location with the maximum observed count of common bits between their operands.

Experimental evaluation Figure 4.4 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

```
 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 4.5. $\langle x,y\rangle$ designates an executed basic block transition 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 4.5: Motivating post-diff basic block transitions as DSF for incremental fuzzing.

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.

4.4.3 `diff`: Incremental Fuzzing

Fuzzing tools are 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 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 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 [45] address this application, but can require several hours of static analysis to pre-compute distances to target program locations[2].

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

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

the set of modified lines of code as the *diff*. To measure the variety of paths executed by the inputs, we will focus on basic block transitions (BBTs) rather than basic blocks alone. This notion of basic block transition is identical to the notion used by AFL (ref. Chapter 2).

Consider the example program given in Figure 4.5a. 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 4.5b shows some inputs and the execution paths they take through this program. The execution path is represented as the sequence of BBTs 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 ϟ.

Consider the three inputs in Figure 4.5b. 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 BBTs 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 BBTs executed after hitting the diff as *post-diff BBTs*; the newly exercised post-diff BBTs are highlighted in blue in Figure 4.5b. Since input i_3 covers a new post-diff BBT, 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 waypoints. It does so by populating the DSF map with the number of times each BBT is executed *after* the diff code has been executed (i.e., it must keep track of the BBTs after the ϟ). 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 $\{\langle 4, 6 \rangle \mapsto 1, \langle 6, 7 \rangle \mapsto 1\}$.

Since we keep track of basic block transitions rather than simply basic blocks, $K = \mathbb{L} \times \mathbb{L}$. To better approximate paths, the DSF map collects order-of-magnitude aggregation of BBT execution counts. Thus, $A = 2^{\mathbb{N}}$, $a_0 = \emptyset$, and the reducer function is $a \rhd v = a \cup \log_2(v)$. The instrumentation adopts AFL's BBT tracking logic, described in Section 2.1.

To make sure that we only track *post-diff* BBTs, 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 BBT $\langle p, c \rangle$ is only incremented (with a call the function `dsf_increment`) if `hits_diff` is `true`, effectively counting only post-diff BBTs.

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 the set of saved inputs from one of our 24-hour runs of AFL as the new starting set of test inputs, S_0. To find a relevant code change, we then advance the code

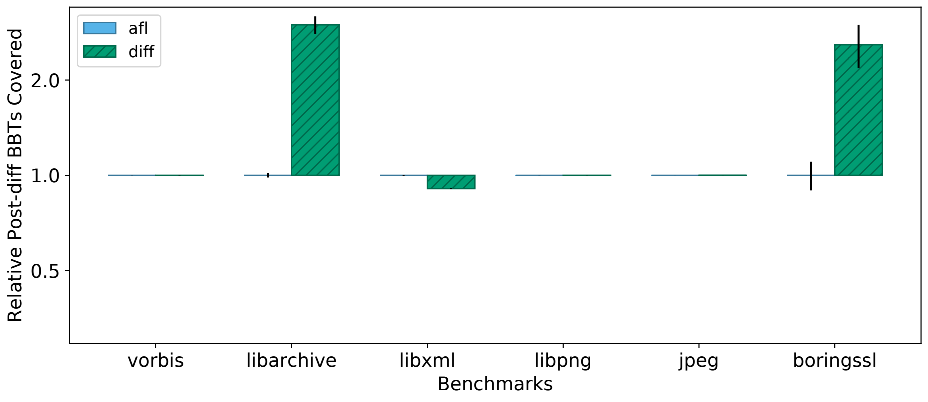


Figure 4.6: Relative (compared to AFL) coverage of basic block transitions after five minutes of incremental fuzzing with the domain-specific `diff` front-end.

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 S_0. 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 evaluate the coverage achieved by any input AFL generated that hit the diff in the five minute run.

Figure 4.6 contains the results of our 5-minute incremental fuzzing evaluation. The figure plots means and standard errors of the number of post-diff BBTs hit by all generated inputs, 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`.

4.4.4 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,

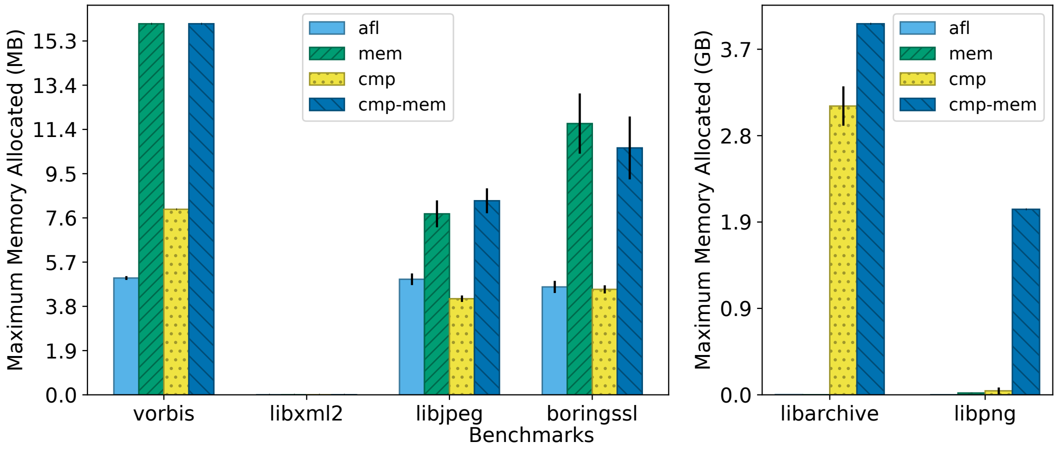


Figure 4.7: Evaluation of composing `cmp` and `mem` into the `cmp-mem` domain. Bars show the maximum dynamic memory allocated—in MB on the left and in GB on the right—at a single program location. Higher is better.

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 4.7 shows the results of our experiments with a composite application that smooths hard comparisons (ref. Section 4.4.2) and maximize dynamic memory allocations (ref. Section 4.4.1). 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`). Note the difference in y-axis between the four benchmarks on the left-hand-side (maximum value listed is 15.3 MB) and right-hand-side (maximum value listed is 3.7 *GB*) of Figure 4.7.

In four benchmarks, the `cmp-mem` composition is able to generate inputs that allocate more memory those those generated by `cmp` or `mem` individually. That is, the composite application performs better than the sum of its parts. 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 4.4.1). However, when combined with the application that smooths hard comparisons, we were able to quickly generate valid ZIP archives and eventually generated a ZIP bomb: a small input that when decompressed 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 4.4.1), the PNG bomb generated by `cmp-mem` relies on compressed data. This input demonstrates that simply capping an image's geometric dimensions is not sufficient for limiting memory usage.

Memory leak discovery Our experiments with `cmp-mem` led to the discovery of a memory leak in `libarchive`. 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. To our surprise, we not only found memory leaks in the January 2017 snapshot of `libarchive`, but also in the latest (March 2019) version of `libarchive`, using inputs generated while fuzzing the old version. The project developer fixed the bug after we reported it.

4.5 Discussion

FuzzFactory allows developers and researchers to control the process of fuzz testing by defining a strategy to selectively save intermediate inputs. However, it does not 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 [46] or FairFuzz [115] to work with any of the various domain-specific fuzzing applications described herein. The main contribution of this chapter is the separation of concerns between the fuzzing algorithm and the choice of feedback from the instrumented program under test.

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 code coverage domain. FuzzFactory allows users to disable this default domain via an environment variable.

Since the completion of the experiments for the OOPSLA'19 paper on FuzzFactory [150], even more specialized fuzzers that fit our abstraction of *waypoints* have appeared: e.g. (1) Coppik et al. [61] save inputs that read/write new values to input-dependent memory addresses, and (2) Nilizadeh et al. [140] discover side-channel vulnerabilities by saving inputs whose execution paths maximally differ from a reference path. The continued appearance of these domain-specific fuzzers strengthens the *key result* from both this chapter and the last. By adding feedback that expands the notion of "interesting input" in coverage-guided fuzzing, we can consistently find specialized bugs whose appearance is not captured by branch coverage alone.

Part II

Structured Mutations

Chapter 5

FairFuzz: Mutation Masking for Deeper Coverage

The last part presented two projects, PerfFuzz and FuzzFactory, which changed the feedback component of coverage-guided fuzzing in order to achieve diverse testing goals. However, both of those projects did not touch another important aspect of coverage-guided fuzzing: the mutations used to create inputs. This part—Chapters 5 and 6—explores the impact of mutations on the effectiveness of coverage-guided fuzzing in more detail.

As mentioned in the introduction, AFL and other coverage-guided fuzzing (CGF) algorithms create new inputs by performing byte-level mutations on existing, saved inputs. For programs expecting highly structured inputs—like a compiler or an XML linter—rather than binary file formats, this can result in many malformed inputs which are unable to explore the program very deeply. Overall, this limits the overall coverage achieved by CGF to the parsing or input validation stages of the program under test.

In Section 4.4.2 of Chapter 4, we saw the `cmp` approach to getting deeper program coverage. Effectively, this approach gave breadcrumbs to the fuzzer for making progress through so-called *hard comparisons.* These hard comparisons, sometimes called *magic byte* comparisons, are ones that rely on a subsequence or function of the input being *exactly equal* to a given value. Many works have tried to modify CGF to increase the probability of getting through these types of magic bytes comparisons [46, 161, 117, 54].

The `cmp` approach increases the probability of generating inputs which satisfy these conditions, and thus, increase the coverage of the program under test. However, it still relies on getting lucky with random mutations: the approach does not alter the mutation strategy, but simply saves inputs if, by luck, a mutant gets further through the comparison. This process could be much more effective by targeting the mutations in a manner that does not ruin the program exploration progress made by the parent input.

This chapter presents a lightweight technique, called FairFuzz, which helps increase the coverage of the program under test by targeting mutations in such a manner. This technique requires *no extra instrumentation* beyond regular CGF instrumentation, preserving CGF's ease-of-use. While the focus of FairFuzz is on branch coverage, a similar concept could be used to target other kinds of coverage and testing objectives, including the `cmp` feedback.

FairFuzz works in two main steps. First, it identifies the program branches that are rarely hit by previously generated inputs. We call such branches *rare branches*. These rare branches guard under-explored functionalities of the program. By generating more random inputs hitting these rare branches, FairFuzz greatly increases the coverage of the parts of the code guarded by them.

Second, FairFuzz uses a novel lightweight mutation technique to increase the probability of hitting these rare branches. The mutation strategy is based on the observation that certain parts of an input already hitting a rare branch are crucial to satisfy the conditions necessary to hit that branch. Therefore, to generate more inputs hitting the rare branch via mutation, the parts of the input that are crucial for hitting the branch should not be mutated. FairFuzz identifies these crucial parts of the input by performing a number of small mutation experiments. Later, in test input generation, it avoids mutating these crucial parts of the input. This mutation strategy is orthogonal to approaches that to help CGF pass magic byte comparisons (like the `cmp` approach mentioned above), and can be combined with them to increase code coverage even further.

We begin with an overview of the FairFuzz algorithm, by studying a particular example on which its core innovation brings benefits.

5.1 Motivation

FairFuzz is built on top of American Fuzzy Lop (AFL) [185], a popular coverage-guided fuzz tester. As discussed in Chapter 2, AFL uses coverage feedback to guide its input generation, saving inputs which cover new edges in the control flow graph of the program under test.

While AFL's search strategy is guided by coverage, AFL often fails to cover some important functionalities of the program under test. And, if a program region is not covered, there is no way AFL can find bugs or crashes in that region.

Consider the code fragment shown in Figure 5.1. It is adapted from the `parser.c` file used in `libxml2`'s `xmllint` utility. We ran AFL on this benchmark 20 times, each time for 24 hours. Only in one of these 24-hour runs did AFL produce an input passing Line 1. Even then, AFL failed to explore the contents of any of the if statements in Lines 6-30. As such, it failed to explore the large quantity of code after Line 31 (mostly omitted in Figure 5.1). Since this code is not even *covered*, then AFL simply cannot find any bugs in it.

The key reason AFL is unable to produce inputs covering any of this code—even after discovering an input containing `<!ATTLIST`—is that AFL mutates inputs paying no attention to which byte values are required to cover particular parts of the program. For example, after having produced the input `<!ATTLIST BD`, AFL will not prioritize mutation of the bytes after `<!ATTLIST`. Instead, it is as likely to produce the mutants `<!CATLIST BD`, `<!!ATTLIST BD`, or `???!ATTLIST BD` as it is to produce `<!ATTLIST ID`. However, to explore the code in Figure 5.1, once AFL discovers `<!ATTLIST BD`, it should not mutate the `<!ATTLIST` part of this input. To see why, suppose that the production of an input like `<!ATTLIST ID`—with the token

```
if (CMP9(ptr,'<','!','A','T','T','L','I','S','T')) {
  ptr += 9;
  /* some processing code omitted */
  while ((ptr != '>') && (ptr != EOF)){
   int type = 0;
   if (CMP5(ptr,'C', 'D', 'A', 'T', 'A')){
       ptr += 5;
       type = XML_ATTRIBUTE_CDATA;
   } else if (CMP6(ptr,'I', 'D', 'R', 'E', 'F', 'S')){
       ptr += 6;
       type = XML_ATTRIBUTE_IDREFS\;
   } else if (CMP5(ptr,'I', 'D', 'R', 'E', 'F')){
       ptr += 5;
       type = XML_ATTRIBUTE_IDREF;
   } else if ((ptr == 'I') && ((ptr+1)== 'D')){
       ptr += 2;
       type = XML_ATTRIBUTE_ID;
   } else if (CMP6(ptr,'E','N','T','I','T','Y')){
       ptr += 6;
       type = XML_ATTRIBUTE_ENTITY;
   } else if (CMP8(ptr,'E','N','T','I','T','I','E','S')){
       ptr += 8;
       type = XML_ATTRIBUTE_ENTITIES;
   } else if (CMP8(ptr,'N','M','T','O','K','E','N','S')){
       ptr += 8;
       type = XML_ATTRIBUTE_NMTOKENS;
   } else if (CMP7(ptr,'N','M','T','O','K','E','N')){
       ptr += 7;
       type = XML_ATTRIBUTE_NMTOKEN;
   }
   if (type == 0) {ptr++; break;}

   /* more omitted code */

   if (CMP9(ptr,'#','R','E','Q','U','I','R','E','D')) {
     ptr += 9;
     default_decl = XML_ATTRIBUTE_REQUIRED;
   }
   if (CMP8(ptr,'#','I','M','P','L','I','E','D')) {
     ptr += 8;
     default_decl = XML_ATTRIBUTE_IMPLIED;
   }
   if (CMP6(ptr,'#','F','I','X','E','D')) {
     ptr += 6;
     default_decl = XML_ATTRIBUTE_FIXED;
     if (!IS_BLANK_CH(ptr)) {
       xmlFatalErrorMsg("Space required after '#FIXED'");
     }
   }
    ptr++;
  }
}
```

Figure 5.1: Code fragment based off the `libxml` file `parser.c` showing many nested if statements that must be satisfied to explore erroneous behavior.

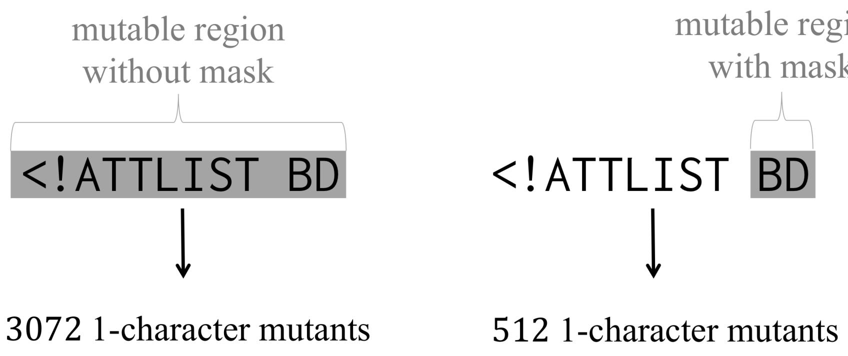


Figure 5.2: Preventing AFL from mutating the `<!ATTLIST` part of this input increases the probability of generating `<!ATTLIST ID` by at least 6×.

"`ID`"—is required to pass the processing code omitted in Line 3 of Figure 5.1. Preventing the modification of `<!ATTLIST` increases AFL's probability of generating `<!ATTLIST ID` by at least 6×. Figure 5.2 illustrates how restricting mutations to only the last two characters of the input yields to a smaller space of mutants to explore, and thus, a higher probability of discovering an input that gets deeper into the program.

5.1.1 Overview of FairFuzz

FairFuzz is a two-pronged approach that addresses this concern and can be smoothly integrated into AFL or other coverage-guided fuzzers. It works as follows.

The first part of FairFuzz is the identification of statements like the if statement in Line 1 of Figure 5.1, which potentially guard large unvisited regions of code. For this, we utilize the observation that such statements are usually hit by very few of AFL's generated inputs (i.e. they are *rare*), and can thus be easily identified by keeping the track of the number of inputs which hit each branch. Intuitively, the code guarded by a branch hit by few inputs is much less likely to have been thoroughly explored than the code guarded by a branch hit by a huge percentage of generated inputs.

Having identified these rare branches as targets, FairFuzz modifies the input mutation strategy in order to keep the condition of the rare branch satisfied. Specifically, it uses a deterministic mutation phase to approximately determine the parts of the input that cannot be mutated in order to hit the rare branch. The subsequent mutation stages are not allowed to mutate these crucial parts of the input. This significantly increases the probability of generating new inputs that hit the rare branch, increasing the probability of exploring the code guarded by the branch, and thus, exploring the program under test more deeply. While FairFuzz uses this mutation masking strategy to target rare branches, the mutation masking is general and can be applied to other testing targets.

Algorithm 5 The FairFuzz algorithm. Differences from regular coverage-guided fuzzing are highlighted in gray.

```
Input: an instrumented test program $p$, a set of initial seed inputs $S_0$
Output: a corpus of automatically generated inputs $\mathcal{S}$
 1: $\mathcal{S} \leftarrow S_0$
 2: $totalCoverage \leftarrow$ initCoverage($S_0$)
 3: repeat
 4:   for $input$ in $\mathcal{S}$ do
 5:     with probability fuzzProb($input$) do
 6:       $target \leftarrow$ getRarestBranch($input$)
 7:       $mask \leftarrow$ computeMask($p, input, target$)
 8:       for $1 \leq i \leq$ numChildren($input, mask$) do
 9:         $input' \leftarrow$ mutateWithMask($input, mask$)
10:         $coverage \leftarrow$ execute($p, input'$)
11:         if $coverage \not\subseteq totalCoverage$ then
12:           $\mathcal{S} \leftarrow \mathcal{S} \cup \{input'\}$
13:           $totalCoverage \leftarrow totalCoverage \cup coverage$
14: until given time budget expires
15: return $\mathcal{S}$
```

5.2 The FairFuzz Algorithm

Algorithm 5 outlines the FairFuzz algorithm and how it differs from the coverage-guided fuzzing algorithm (Algorithm 1). The essential differences are that inputs are selected based on whether or not they hit a rare branch (Line 5), then a mutation mask is computed for the rarest branch hit by the selected input (Lines 6, 7), and the mutation mask is used to filter and bias mutations towards the target branch (Lines 8, 9). Notice that unlike the changes to the coverage-guided fuzzing algorithm studied in Part I, there is no change to the feedback used to judge interesting inputs (Lines 10-13).

We begin with an abstract treatment of the mutation masking technique.

5.2.1 Mutation Masking

In this section we introduce the mutation mask for a given input, x, and a given testing target, T. We say *satisfies*(x, T) is true if input x satisfies T.

Definition 12. A *mutation* is a tuple (c, m), where m is the number of bytes impacted by the mutation and c is one of the following mutation categories:

O: *overwrites* m bytes starting at position k with some values,

I: *inserts* some sequence of m bytes at position k,

D: *deletes* m bytes starting at position k.

To fully specify mutations with $c \in \{O, I\}$, we must also specify the values to be inserted. Given an input x, a mutation $\mu = (c, m)$, and a position $0 \leq i < |x| - m$, let $\mathit{mutate}(x, \mu, i)$ denote the input produced by applying mutation μ on x at position i.

Definition 13. The *mutation mask* for an input x and a testing target T is a function $\mathit{mask}_{x,T} \colon \mathcal{N} \to \mathcal{P}(\{O, I, D\})$ which takes a position i in the input x and returns a subset of $\{O, I, D\}$. We say that a mutation category $c \in \mathit{mask}_{x,T}(i)$ if $\mathit{satisfies}(\mathit{mutate}(x, (c, 1), i), T)$ is true. That is, if c is in the set $\mathit{mask}_{x,T}(i)$, then after applying a mutation of category c at position i on x, the resulting input will satisfy the target T.

Intuitively, the mutation mask specifies whether the input produced from mutating x at position i will (likely) reach the testing target. With this mask, given a mutation $\mu = (c, m)$ at position k, we can compute

$$\textsc{okToMutate}(\mathit{mask}_{x,T}, \mu, k) = \bigwedge_{i=k}^{k+m-1} c \in \mathit{mask}_{x,T}(i).$$

We describe the algorithm to compute $\mathit{mask}_{x,T}(i)$ in Section 5.2.2.2.

5.2.1.1 Biasing Mutation with the Mutation Mask

FairFuzz uses $\textsc{okToMutate}(\mathit{mask}_{x,T}, \mu, k)$ to bias mutations towards the testing target.

As discussed in Section 2.1, there are two main stages in AFL's mutation algorithm. Biasing mutations with the mask is done a somewhat differently in both of these stages.

First, in the deterministic mutation stages, any mutation μ which does not satisfy okToMutate is simply skipped. That is, FairFuzz simply does not apply that mutation to the parent input. For a given mutation type μ and position i, the mutant $x' = \mathit{mutate}(x, \mu, i)$ is generated only if $\textsc{okToMutate}(\mathit{mask}_{x,T}, \mu, i)$ is true. This has the effect of reducing the number of mutants created in the deterministic mutation stages, thereby reducing the number of children inputs created from the parent inputs (Line 8 of Algorithm 5).

Second, recall that in the havoc stage, mutants are created by choosing a random mutation and random position at which to apply it. FairFuzz selects the random mutation $\mu = (c, m)$ as AFL does (Algorithm 2, Line 5). However, instead of selecting the position at random between 0 and $|\mathit{newinput}| - m - 1$ Algorithm 2, Line 6, FairFuzz chooses the position randomly from the subset of ok-to-mutate positions. Precisely, it replaces that line with a call to $\textsc{randomOkToMutate}(\mathit{mask}_{x,T}, \mu)$, defined as:

$$\mathit{sampleUniform}(\{i \in [0, |x| - m - 1] : \textsc{okToMutate}(\mathit{mask}_{x,T}, \mu, i)\}).$$

If the set of ok-to-mutate positions is empty, FairFuzz skips the mutation and chooses a new $\mu = (c, m)$ at the next iteration of the havoc mutation loop (Algorithm 2, Line 4).

5.2.2 Targeting Rare Branches

So far we have kept the testing target abstract. Now, we concretize it by elaborating the definition of *rare branches* and giving the concrete algorithm which FairFuzz uses to compute the mutation mask for rare branches.

5.2.2.1 Selecting Inputs to Mutate

To bias input generation towards rare branches, FairFuzz selects only inputs that hit rare branches for mutation. First, we formalize the concept of a rare branch.

Definition 14. We say that an input x *hits* a branch b, denoted $\mathit{hits}(x, b)$, if the execution of the program on x exercises the branch b at least once.

The *hit count* of a branch is the number of produced inputs i which have exercised the branch. More formally,

Definition 15. Let $\mathcal{I}$ be the set of all inputs produced by fuzzing so far. The *hit count* of branch b is

$$\mathit{numHits}[b] = |\{x \in \mathcal{I} : \mathit{hits}(x, b)\}|.$$

To establish *numHits*, FairFuzz runs one round of the coverage-guided fuzzing loop on the seed input with no masking.

A natural idea is to designate the n branches hit by the fewest inputs as rare, or the branches hit by less than p percent of inputs to be rare. After some initial experiments, we rejected these methods as (a) they can fail to capture what it means to be rare (e.g. if $n = 5$ and the two rarest branches are hit by 20 and 15,000 inputs, both would be "rare"), and (b) these thresholds need to be modified for different benchmarks. Instead, we define a rare branch as one whose hit count is smaller than a dynamic rarity cutoff as follows. Let B be the set of all branches in the program.

Definition 16. Let $B_v = \{b \in B : \mathit{numHits}[b] > 0\}$. A *rare branch* is a branch b such that

$$\mathit{numHits}[b] \leq \mathit{rarity\_cutoff}$$

where

$$\mathit{rarity\_cutoff} = 2^i \text{ such that } 2^{i-1} < \min_{b' \in B_v}(\mathit{numHits}[b']) \leq 2^i.$$

For example, if the branch hit by the fewest inputs has been hit by 17 inputs, any branch hit by $\leq 2^5$ inputs is rare.

To determine whether an inputs hits a rare branch, FairFuzz computes the *rarest branch* hit by the input:

Algorithm 6 Computing the mutation mask in FairFuzz.

```
1: procedure computeMask($p$, $input$, $branch$)
2:     $mask \leftarrow$ initWithEmptySet($|input|$)
3:     for $0 \leq i < |input|$ do
4:         $inputO \leftarrow$ mutate($input$, $flipByte$, $i$)
5:         if $branch \in$ branchesHitBy($p$, $inputO$) then
6:             $mask[i] \leftarrow mask[i] \cup \{O\}$
7:         $inputI \leftarrow$ mutate($input$, $addRandomByte$, $i$)
8:         if $branch \in$ branchesHitBy($p$, $inputI$) then
9:             $mask[i] \leftarrow mask[i] \cup \{I\}$
10:        $inputD \leftarrow$ mutate($input$, $deleteByte$, $i$)
11:        if $branch \in$ branchesHitBy($p$, $inputD$) then
12:            $mask[i] \leftarrow mask[i] \cup \{D\}$
       return $mask$
```

Definition 17. Let $branches(x) = \{b \in B : hits(x, b)\}$. Then the *rarest branch* hit by input x is the branch b^* such that

$$b^* = \underset{b \in branches(x)}{\arg\min}\ numHits[b].$$

Then, FairFuzz selects only inputs whose rarest branch is a rare branch for mutation. That is, fuzzProb(x) in Line 5 of Algorithm 5 is 1 if the rarest branch b^* hit by input x is rare (i.e. is $\leq$ *rarity_ cutoff*), and 0 otherwise.

5.2.2.2 Computation of the Mutation Mask

Algorithm 6 outlines how FairFuzz computes $mask_{x,b}$ for a given input x and rare branch b. This algorithm could be easily adapted to other testing targets by replacing $hits(x^c, b)$ with $satisfies(x^c, T)$. The algorithm works as follows.

For each position i in the x, FairFuzz produces the mutants x^O by flipping the byte at position i (Line 4 of Algorithm 6), x^I by adding a random byte at position i (Line 7), and x^D by deleting the byte at position i (Line 10). Then, for each x^c, FairFuzz determines whether $hits(x^c, b)$ by running x^c through the program (captured in branchesHitBy on Lines 5, 8, 11). Finally, if x^c hits b, FairFuzz notes the position i as overwritable (O), insertable (I), or deletable (D), respectively (Lines 6, 9, 12). While the calculation is illustrated as separate from the deterministic mutation stages in Algorithm 5, the two are integrated in the implementation. Since the mask computation adds only two new deterministic mutation types to AFL (byte-flipping is a default mutation type), the computation adds negligible overhead to stock AFL.

Of course, this computation of $O \in mask_{x,b}(i)$ and $I \in mask_{x,b}(i)$ is approximate—FairFuzz doesn't check whether every value overwritten or inserted results in b being hit.

Unfortunately, trying all possible values to insert or write is too expensive and produces too many redundant inputs. Empirically we find this approximation produces an effective mutation mask (see Section 5.3.3).

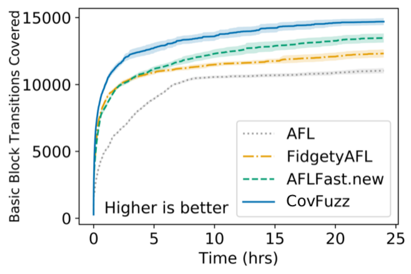
5.2.3 Trimming Inputs for Testing Targets

AFL's efficiency depends on large part on its ability to quickly produce and modify inputs [189]. Thus, it is important to make sure the deterministic mutation stage—and in FairFuzz, mutation mask computation—is efficient. Since the runtime of the computation is linear in the length of the selected input, FairFuzz needs to keep the length of the inputs in the queue short. AFL has two techniques for keeping inputs short: (1) prioritizing short inputs when selecting inputs for mutation and (2) trimming (an efficient approximation of delta-debugging [191]) the parent input before mutating it. This trimming is omitted from Algorithms 1 and 5 for clarity. Trimming attempts to minimize the input to mutate with the constraint that the minimized input hits the same path (set of (*branch ID*, *branch hits*)) as the unminimized one. However, this constraint is not good enough for reducing the length of inputs when very long inputs are chosen. FairFuzz may do this since it selects inputs based only on whether they hit a rare branch. We found that we can make inputs shorter in spite of this if we relax the trimming constraint. In particular, we relax the constraint to require that the minimized input hits only the target branch of the original input, instead of the same path as the original input. Similar relaxation could be done for other testing targets. We refer to FairFuzz with this relaxed constraint as FairFuzz *with trimming*.

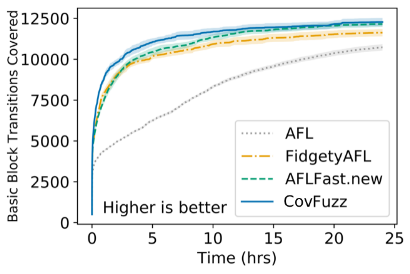
5.3 Evaluation

FairFuzz is implemented as an open source tool built on top of AFL. The implementation adds around 600 lines of C code to the file containing AFL's core implementation.

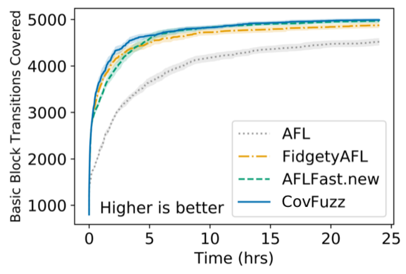
We evaluated FairFuzz on 9 different real-world benchmarks. We selected these from those favored for evaluation by the AFL creator (`djpeg` from libjpeg-turbo-1.5.1, and `readpng` from libpng-1.6.29), those used in AFLFast's evaluation (`tcpdump -nr` from tcpdump-4.9.0; and `nm`, `objdump -d`, `readelf -a`, and `c++filt` from GNU binutils-2.28) and a few benchmarks with more complex input grammars in which AFL has previously found vulnerabilities (`mutool draw` from mupdf-1.9, and `xmllint` from libxml2-2.94). Since some of these input formats had AFL dictionaries and some did not, we ran all the evaluation without dictionaries to level out the playing field. In each case we seeded the fuzzing run with the inputs in the corresponding AFL `testcases` directories (except `c++filt`, which was seeded with the input "`_Z1fv\n`"); for PNG we used only `not_kitty.png`.



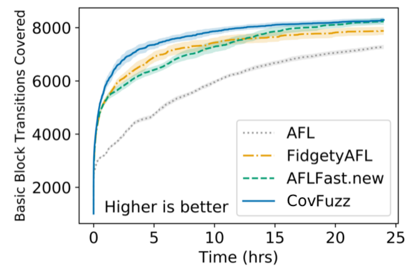
(a) tcpdump



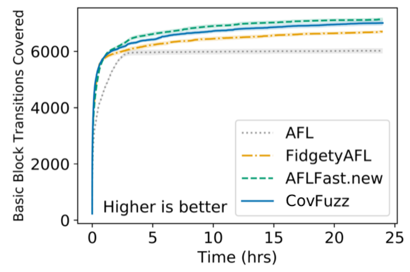
(b) readelf



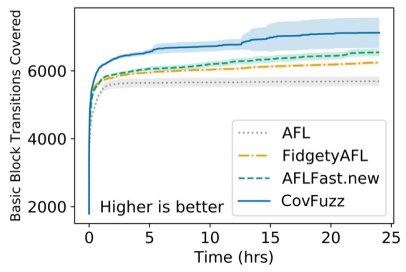
(c) nm



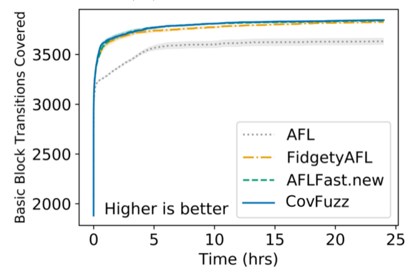
(d) objdump



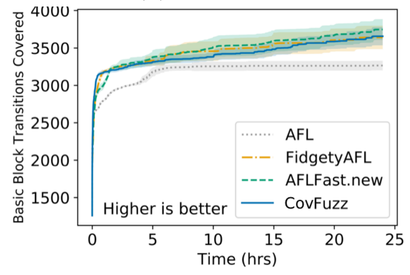
(e) c++filt



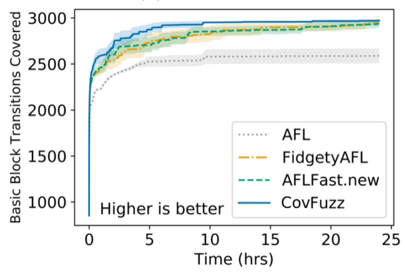
(f) xmllint



(g) mutool draw



(h) djpeg



(i) readpng

Figure 5.3: Number of basic block transitions (AFL branches) covered by different AFL techniques averaged over 20 runs (bands represent 95% C.I.s).

5.3.1 Coverage Compared to Prior Techniques

In this section of evaluation we compare three popular versions of AFL against FairFuzz, all based off of AFL version 2.40b.

1. "AFL" is the vanilla AFL available from AFL's website.

2. "FidgetyAFL" [188] is AFL run without deterministic mutations.

3. "AFLFast.new" [42] is AFLFast run without deterministic stage and with the cut-off-exponential exploration strategy.

We ran FairFuzz with input trimming for the testing target and omitting all deterministic stages except those necessary to compute the mutation mask.

We ran each technique for 24 hours (on a single core) on each benchmark. We repeated each 24 hour experiment 20 times for each benchmark. We ran our experiments for 24 hours as the fuzzing process does not have a defined end-time and this is a runtime used in prior work [46, 172, 161, 117]. We repeated our experiments 20 times because fuzz testing is an inherently non-deterministic process, and so is its performance.

All machines ran Ubuntu 16.04. We ran `mutool`, `djpeg` and `readpng` on a machine with four 2.27 GHz Intel(R) Xeon(R) E7- 4860 processors (40 cores) with 264GB of RAM; we ran `c++filt` on a machine with four 2.27 GHz Intel(R) Xeon(R) X7560 processors (32 cores) with 264GB of RAM; we ran `objdump` and `nm` on a machine with four 2.40GHz Intel(R) Xeon(R) E5-4640 processors (32 cores) and 264GB of RAM; and we ran `readelf`, `xmllint`, and `tcpdump` on two machines, each with a single 3.00 GHz AMD Ryzen 7 1700 processor and 16 GB of RAM. We ran instances of the same AFL technique in at the same time (1×20 runs for `mutool`, `djpeg` and `readpng`, 2×10 runs for `c++filt`, `objdump`, and `nm`, and 5×4 runs for `readelf`, `c++filt`, and `tcpdump`), and different techniques in sequence.

5.3.1.1 Overall Branch Coverage Achieved

We begin by analyzing coverage achieved by different techniques through time. The main metric we report is basic block transitions covered, which is close to the notion of branch coverage used in real-world software testing.

Figure 5.3 plots, for each benchmark and technique, the average number of branches covered over all 20 runs at each time point (dark central line) and 95% confidence intervals in branches covered at each time point (shaded region around line) over the 20 runs for each benchmark. For the confidence intervals we assume Student's t distribution.

From Figure 5.3, we see that that on all benchmarks except `c++filt`, FairFuzz achieves the upper bound in branch coverage, generally showing the most rapid increase in coverage at the beginning of execution.

Note that while FairFuzz keeps a sizeable lead on the `xmllint` benchmark (Figure 5.3f), it does so with wide variability. Closer analysis reveals that one run of FairFuzz on `xmllint` was buggy, and no inputs were selected for mutation—this run covered no more than 6160 branches. However, FairFuzz had two runs on `xmllint` covering an exceptional 7969 and 10990 branches, respectively.

Figure 5.4, shows, at every hour, for how many benchmarks each technique has the *lead* in coverage. By *lead* we mean its average coverage is above the confidence intervals of the other techniques, and no other technique's average lies within its confidence interval. We say two techniques are tied if one's average lies within the confidence interval of the other. If techniques tie for the lead, the benchmark is counted for both techniques in Figure 5.4, which is why the number of benchmarks at each hour may add up to more than 9. This figure shows that FairFuzz *quickly achieves a lead in coverage* on nearly all benchmarks and *is not surpassed* in coverage by the other techniques in our time limits.

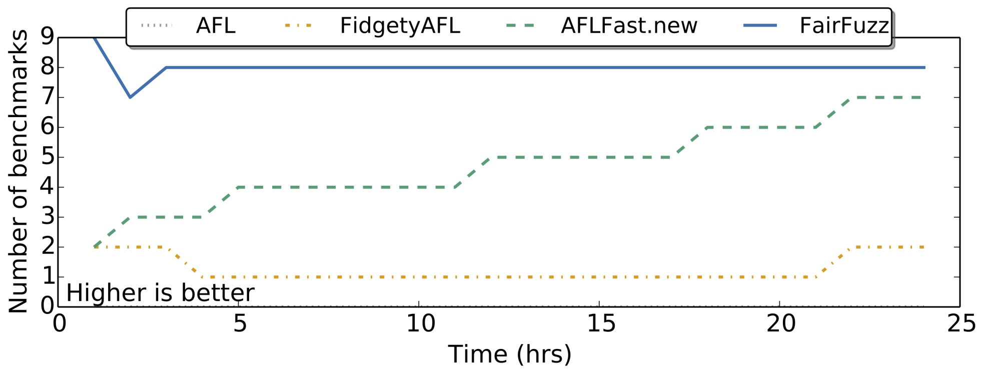


Figure 5.4: Number of benchmarks on which each technique has the lead in coverage at each hour. A benchmark is counted for multiple techniques if two techniques are tied for the lead.

5.3.1.2 Detailed Analysis of Coverage Differences.

Figure 5.3 shows there are three benchmarks (`c++filt`, `tcpdump`, and `xmllint`) on which one technique achieves a statistically significant lead in AFL's branch coverage after 24 hours (with AFLFast.new leading on `c++filt` and FairFuzz on the other two). A natural question is what these increases corresponded to at the source code level.

Since AFL saves all inputs that achieve new program coverage to disk, we can replicate what program coverage was achieved in each run by replaying these saved inputs through the programs under test. Since each benchmark was run 20 times, we take the union (over each technique) of saved inputs for all 20 runs. We ran the union of the inputs for each technique through their corresponding programs and then ran `lcov` on the results to reveal coverage differences. Using the union is a generous approach, revealing only which regions are uncoverable by the different techniques over all the 20 runs.

`xmllint` The bulk of the coverage gains on `xmllint` were in the main `parser.c` file. The key trend in increased coverage appears to be FairFuzz's increased ability to discover keywords.

For example, both AFL and FairFuzz have higher source code coverage than FidgetyAFL and AFLFast.new as they discovered the patterns `<!DOCTYPE` and `<!ATTLIST` in at least one run. However, FairFuzz also produced inputs satisfying *all the other conditionals* illustrated in Figure 5.1, which meant discovering all the keywords used in the comparisons. The produced inputs included:

```
<!DOCTYPET@[ <!ATTLIST?D T NMTOKENS
<!DOCTYPE?[ <!ATTLIST D T ENTITY
<!DOCTYPE\[ <!ATTLISTíD T ID #REQUIRED^@^P
```

We believe the mutation masking technique is directly responsible for the discovery of these. Consider the `<!ATTLIST` block covered by the inputs above, whose code is outlined in in Figure 5.1. While both AFL and FairFuzz had a run discovering the sequence `<!ATTLIST`,

Table 5.1: Number of runs producing an input with the given sequence in 24 hours.

	Technique			
Sequence	AFL	FidgetyAFL	AFLFast.new	FairFuzz
`<!A`	7	15	18	17
`<!AT`	1	2	3	11
`<!ATT`	1	0	0	1

of all the saved inputs for AFL in that run, only 0.4% of them (18) visited Line 2 of Figure 5.1, resulting in 18 hits of the line. In contrast, 12.3% (1169) of the saved inputs produced by FairFuzz in the run where it discovered `<!ATTLIST` visited Line 2 of Figure 5.1, resulting *2124* hits of the line. With two orders of magnitude more hits of this line, it is obvious that FairFuzz was better able to explore the code in Figure 5.1. The orders of magnitude difference is likely due to the mutation mask.

To confirm the effect, we also look at the number of runs which produced subsequences of `<!ATTLIST`. This is illustrated in Table 5.1. The decrease in the number of runs discovering `<!AT` from the number of runs discovering `<!A` in this table shows the mutation mask in action, with 11 of FairFuzz' runs discovering `<!AT`, compared to 1, 2, and 3 for AFLFast.new, FidgetyAFL, and AFLFast.new, respectively.

Finally, as may be clear from the example inputs above, although FairFuzz discovered more keywords, the inputs it produced were not necessarily more well-formed. Nonetheless, these inputs allowed the FairFuzz to explore more of the program's fault modes. This is reflected in the coverage of a large case statement differentiating 57 error messages in `parser.c`. Both FidgetyAFL and AFLFast.new cover only 22 of these cases, AFL covers 33, and FairFuzz covers 39.

`tcpdump` Like `xmllint`, `tcpdump` has extensive nested structure. Coverage for `tcpdump` differs a bit for all four techniques over a variety of different files, but the biggest gains for FairFuzz are in three files printing certain packet types (`print-forces.c`, `print-llc.c`, and `print-snmp.c`).

The coverage gains in these files suggest FairFuzz is better able to automatically detect sequences in the inputs necessary to increase program coverage. For example, unlike the other three techniques, FairFuzz was able to create files that have legal ForCES (RFC 5810) packet length. FairFuzz was also able to create IEEE 802.2 Logical Link Control (LLC) packets with the organizationally unique identifier (OUI) corresponding to RFC 2684, and subsequently explore the many subtypes of this OUI. Finally, in the Simple Network Management Protocol parser, FairFuzz was able to create inputs corresponding to Trap PDUs, as well as some inputs with a correct SNMPv3 user-based security message header.

We note these gains in coverage seem less impressive than those of FairFuzz on `xmllint`, even though the performance in Figure 5.3 looks similar. This appears to be because

FairFuzz gets consistently higher coverage of `tcpdump` instead of covering parts of the program wholly uncoverable by the other techniques. We can see this by looking at the number of branches covered *by at least one of* the 20 runs (the union over the runs) and the number of branches covered *at least once in all* the 20 runs (the intersection over the runs). For `tcpdump`, FairFuzz has a consistent increase in the intersection of coverage (FairFuzz's contains 11,293 branches compared to AFLFast.new's 10,724), but a smaller gain in the union (FairFuzz's is 16,129, while AFLFast.new's is 15,929). On the other hand, the intersection of coverage for `xmllint` is virtually the same for all techniques except stock AFL (5,876 for FidgetyAFL, 5,778 for AFLFast.new and 5,884 for FairFuzz, maybe because of the buggy run mentioned in Section 5.3.1.1), but FairFuzz's union of coverage (11,681) contains over 4,000 more branches than that of AFLFast.new (7,222).

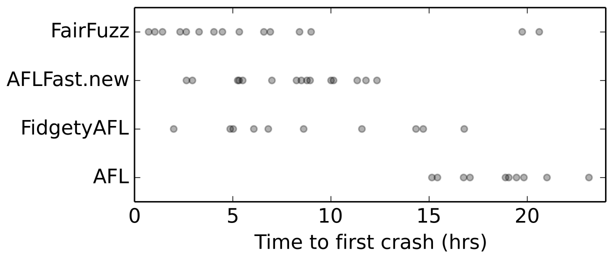
`c++filt` The differences in terms of source code coverage between techniques were much more minimal for `c++filt` than for `tcpdump` or `xmllint`. For example, FairFuzz covers 3 lines in `cp-demangle.c` that AFLFast.new does not, related to demangling binary components when the operator has a certain opcode. On the other hand, AFLFast.new covers a branch where `xmalloc_failed(INT_MAX)` is called when a length bound comparison fails, while FairFuzz fails to produce an input long enough to violate the length bound. FairFuzz also fails to cover a branch in `cxxfilt.c` taken when the length of input read into `c++filt` surpasses the length of the input buffer allocated to store it.

FairFuzz's inability to produce very long inputs may be related to the second round of trimming FairFuzz does. Or, it could be because `c++filt` has highly recursive structure, so full branch coverage is not as good a exploration heuristic for this program. A testing target other than hitting rare branches may be better suited for programs like `c++filt`.

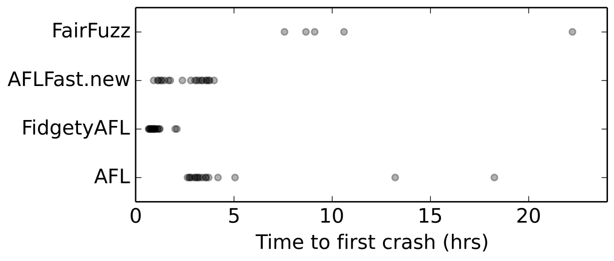
The pattern which emerges from this analysis is that FairFuzz is better able to automatically discover input constraints and keywords—special sequences, packet lengths, organization codes—and target exploration to inputs which satisfy these constraints than the other techniques. It is likely the targeting of rare branches shines the most in the `tcpdump` and `xmllint` benchmarks since these programs are structured with many nested constraints, which the other techniques are unable to properly explore over the time budget (and perhaps even longer) without extreme luck.

5.3.2 Crashing Compared to Prior Techniques

While FairFuzz's goal is to target deeper program coverage by targeting rare branches, it is natural to examine whether this goal has an effect on finding bugs, or crashes, in the program under test. Crash finding is hard to evaluate in practice, due to the sparsity of natural bugs and the effort required to properly deduplicate crashes. Of the 9 benchmarks we tested on, crashes were only found on `c++filt` and `readelf`. We did not do in-depth deduplication of these crashes, instead comparing the techniques on time to first crash.



(a) readelf



(b) cxxfilt

Figure 5.5: Time to find first crash for each run of the different techniques. Each point represents the time to first crash for a single run.

Figure 5.5 shows the time to find the first crash for each of the techniques on each of the 20 24-hour runs. FairFuzz seems to find crashes a bit earlier on `readelf` than the other techniques, but much later on the `c++filt` benchmark. This contrast also appears when looking at the percent of runs finding crashes for each benchmark. On `readelf`, both AFL and FidgetyAFL find crashes in 50% of runs while AFLFast.new and FairFuzz find crashes in 75% of runs. On `c++filt`, however, FidgetyAFL and AFLFast.new find crashes in 100% of the runs, AFL in 85% of them, and FairFuzz in only 25%.

Given that some of the lines missed by FairFuzz in `c++filt` were about buffer length (refer to the previous section), input length may have been factor in this performance discrepancy. Many of the crashing inputs found by AFLFast.new and FidgetyAFL were very long, more than 20KB, with one crashing input generated by AFLFast.new being 130KB long. Crashing inputs of this length were found in only one of the five FairFuzz runs which found crashes in `c++filt`. Thus, a key factor in FairFuzz's poor performance on crash finding on this `c++filt` may be that the method does not encourage the creation of huge inputs. It is also possible that the structure of `c++filt` may be better suited to a path-based exploration strategy (like that of AFL/FidgetyAFL and AFLFast.new) than a branch-based one. For example, demangling function argument types is done in a loop, so FairFuzz is less likely to prioritize many new function arguments than path-based prioritization does. Overall, the results here are inconclusive: FairFuzz may improve crash exposure in some cases (`readelf`), but does not when the crashes are most easily exposed by large inputs.

5.3.3 Can Masking Effectively Target Branches?

The final point of FairFuzz's evaluation is whether the mutation mask strategy effectively biases mutation towards the testing target. In FairFuzz, the target was hitting the same rare branch as the parent input. We conducted the following experiment on a subset of our benchmarks to evaluate the effect of the mask.

We added a *shadow mode* to FairFuzz. When running in shadow mode, every time an input is selected for mutation, FairFuzz first performs all mutations without the influence

Table 5.2: Average % of mutated inputs hitting target branch for one queueing cycle.

(a) Cycle without trimming.

	Det. mask	**Det. plain**	**Hav. mask**	**Hav. plain**
`xmllint`	92.8%	46.5%	31.8%	6.6%
`tcpdump`	99.0%	74.0%	34.2%	9.3%
`c++filt`	97.6%	64.1%	41.4%	14.4%
`readelf`	99.7%	82.7%	57.7%	14.9%
`readpng`	99.1%	34.6%	24.3%	2.4%
`objdump`	99.2%	70.2%	42.4%	9.0%

(b) Cycle with trimming.

	Det. mask	**Det. plain**	**Hav. mask**	**Hav. plain**
`xmllint`	90.3%	22.9%	32.8%	2.9%
`tcpdump`	98.7%	72.8%	36.1%	9.0%
`c++filt`	96.6%	14.8%	34.4%	1.1%
`readelf`	99.7%	78.2%	55.5%	11.4%
`readpng`	97.8%	39.0%	24.0%	2.4%
`objdump`	99.2%	66.7%	46.2%	7.6%

of the mutation mask (the *shadow* run). Then, for the same input, FairFuzz performs all mutations again, using the mutation mask filtering and bias.

This shadow run allows us to compute the difference between the percentage of generated inputs hitting the target with and without the mutation mask *for each parent input.* Since some target branches may be easier to hit than others, this gives us a better idea of how effective the masking technique is in general. In our experiments, we ran FairFuzz with the shadow run on a subset of our benchmarks. For each benchmark we ran a cycle with target branch trimming and one without.

Table 5.2 presents the target branch hit percentages for the deterministic and havoc stages. These percentages are the averages—over all inputs selected for mutation in the first queueing cycle—of the percentage of children inputs hitting the target.

Overall, Table 5.2 shows that the mutation mask largely increases the percentage of mutated inputs hitting the target branch. The hit percentages for the deterministic stage are strikingly high. This is not unexpected because in the deterministic stage the mutation mask simply prevents mutations at locations likely to violate the target branch. Thus, the gain percentage of inputs hitting the target branch in the havoc stage is most impressive. In spite of the use of the mutation mask in the havoc stage being heuristic, we consistently see the use of the mutation mask causing a 3×-10× increase in the percentage of inputs hitting the target branch. As for trimming, it appears that extra trimming reduces the number of inputs hitting the target branch when the mutation mask is disabled but has minimal effect when the mutation mask is enabled.

5.4 Discussion

While FairFuzz was evaluated with a variety of input formats accomplishing different tasks, the results in terms of increased coverage may not generalize to other programs. Re-evaluations of FairFuzz have found that it indeed achieves higher branch coverage on some benchmarks, but not universally across all benchmarks [55]. However, those same evaluations have found that FairFuzz finds an increased number of bugs compared to baseline AFL.

The foremost limitation of using rare branches as a testing target in FairFuzz is the fact that branches that are never hit by any fuzzer-generated or seed input cannot be targeted by this method. So, it confers little benefits to discovering a single long magic number when progress towards matching the magic number does not result in new coverage. The FairFuzz mutation masking algorithm could be used in conjunction with methods targeting the magic number issue [161, 117, 172] to build a more effective fuzzer.

Nonetheless, this chapter demonstrates how modifying the mutation strategy of a fuzzer is key to enabling deeper exploration of the program. The effect would likely be more powerful used in conjunction with a feedback strategy that gives intermediate rewards for some increased depth of exploration, like the `cmp` feedback module described in Chapter 4. However, as can be seen in the inputs examined in this chapter, FairFuzz is still limited in the space of inputs it can explore. It finds inputs getting through rare branches, but they may not be well-formed in the end. The next chapter looks at how bringing in a more structured input mutation strategy truly enables fuzzing to explore programs beyond the parser.

Chapter 6

Zest: Using Generators for Higher-Level Mutations

The last chapter showed evidence of how powerful mutations which preserve some input structure can be, especially with respect to increasing the overall coverage of the program under test. However, the technique presented in FairFuzz, though lightweight and flexible, still suffers from some of the drawbacks of classical coverage-guided fuzzing. The inputs it creates via byte-level mutations remain more well-suited to stress-testing the parsing or *syntactic analysis* stages of programs.

Programs under test which expect highly structured inputs often consist of both a *syntactic analysis stage*, which parses raw input, and a *semantic analysis stage*, which conducts checks on the parsed input and executes the core logic of the program. If the structural constraints which must be satisfied to pass the syntactic analysis stage are easily ruined by byte-level mutations—e.g., tags matching in an XML document—then classical coverage-guided fuzzing will struggle to explore the semantic analysis stage of the program under test.

Another popular form of fuzzing, generator-based fuzzing, can overcome this drawback. Generator-based fuzzing, discussed in the introduction, relies on a hand-written *input generator*, which can be called repeatedly in order to create inputs. If the domain of inputs returned by the generator closely matches the space of all semantically valid inputs to the program under test, then generator-based fuzzing is very effective at exploring the behavior of the program under test [181, 122]. The problem is that oftentimes, the domain of inputs returned by the generator is not so well-tuned to the program under test, and so, sampling inputs from the generator will not effectively explore the semantic analysis stage of the test program.

Part III explores how to overcome this drawback of generator-based fuzzing in more detail. This chapter focuses on a more targeted problem: whether input generators, and the structure they encode, can be used to perform higher-level mutations in coverage-guided fuzzing. Further, this chapter explores whether these higher-level mutations can then be used to better explore the *semantic analysis stage* of programs. The approach discussed in this chapter, Zest, shows the answer to both of these questions is the affirmative.

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

Figure 6.1: A test driver exercising the `runModel` function.

6.1 Motivation

Suppose a developer of the Maven [9] build system wants to fuzz test the core functionality that turns Maven build files into models, the `runModel` function. Figure 6.1 shows a test harness for this function, called in Line 4. The `testProgram` function accepts as input a string, which is parsed as a Maven document in the call to `ModelReader.readModel` in Line 8. `ModelReader.readModel` does the main work of turning the input string into a model, unless the input is syntactically invalid XML (Line 10) or a semantically invalid Maven file (Line 12). The assumption at Line 3 exits early, but without error, if the input was syntactically or semantically invalid and did not parse into a model. If the input passes the assumption, then the main `runModel` function is finally called parsed model (Line 4).

Now, the developer could simply use the `testProgram` as the program under test for a coverage-guided fuzzer. Unfortunately, most of the inputs generated by coverage-guided fuzzing will never get past the syntactic validity check in Line 10, resulting instead in inputs that look like: `<a b>ac&#84;a>`. Thus, although `testProgram` is being invoked repeatedly by the fuzzer, few inputs will ever get to Line 4, and so, `runModel` will barely be covered. This does not help the developer who wants to test the inner workings of `runModel`.

Another option is for the developer to use generator-based fuzzing. Since a Maven document is a type of XML document, they could use the generator of XML documents in Figure 6.2 to generate inputs for `testProgram`.

At a high level, this generator works as follows. 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 3). Then, `genElement` uses repeated calls to methods of `random` to generate the element's tag name (Line 9), any embedded text (Lines 17, 18, and in `genString`), and the number of children (Line 12); it recursively calls `genElement` to generate each child node. Finally, the `XMLDocument` object is converted to a string before `generate()` returns.

```java
class XMLGenerator {
    public String generate(Random random) {
        XMLElement root = genElement(random, 1);
        XMLDocument doc = new XMLDocument(root);
        return doc.toString();
    }
    private XMLElement genElement(Random random, int depth) {
        // Generate element with random name
        String name = genString(random);
        XMLElement node = new XMLElement(name);
        if (depth < MAX_DEPTH) { // Ensures termination
            int n = random.nextInt(MAX_CHILDREN);
            for (int i = 0; i < n; i++) {
                node.appendChild(genElement(random, depth+1));
            }
        }
        if (random.nextBool()) {  // Maybe insert text inside element
            node.addText(genString(random));
        }
        return node;
    }
    private String genString(Random random) {
        // Generate string of random length and characters
        int len = random.nextInt(1, MAX_STRLEN);
        String str = "";
        for (int i = 0; i < len; i++) {
            str += random.nextChar();
        }
        return str;
    }
}
```

Figure 6.2: A simplified XML document generator.

While such a generator is easy for the developer to write, it will, unfortunately, not help them in their quest to test the `runModel` function! In fact, when running generator-based fuzzing with the generator in Figure 6.2 and the test harness `testProgram` in Figure 6.1, only 0.09% of the generated inputs make it through to the assumption in Line 3, this time because most of the inputs are semantically invalid (i.e., get to Line 12).

To get better results, the developer could sit down and write a specialized generator for Maven documents, whose output is more likely to get through the assumption in Line 3. Unfortunately, the effort required to write such a specialized generator can be quite significant, and may discourage developers from using generator-based fuzzing. Luckily, the approach proposed in the chapter, Zest, is able to use even a basic generator, like that in Figure 6.1, to effectively generate inputs which are both syntactically *and* semantically valid, and thus, explore the behavior of the `runModel` function.

6.2 The Zest Technique

Zest has two key algorithmic components. First, Zest leverages input generators to conduct higher-level structural mutations on inputs by converting a random-input generator into an equivalent deterministic *parametric generator*. Second, Zest uses a *validity-guided parameter search* to guide the input search towards inputs that are not only syntactically, but also *semantically*, valid. At its core, this search augments the CGF algorithm by keeping track of the code coverage achieved by valid inputs.

6.2.1 Parametric Generators

Consider a particular execution of the generator in Figure 6.2, focusing on the calls to `nextInt`, `nextBool`, and `nextChar`. The following sequence of calls will be our running example:

Call → result	**Context**
`random.nextInt(1, MAX_STRLEN)` → 3	Root: `name` length (Line 24)
`random.nextChar()` → ‘`f`’	Root: `name[0]` (Line 27)
`random.nextChar()` → ‘`o`’	Root: `name[1]` (Line 27)
`random.nextChar()` → ‘`o`’	Root: `name[2]` (Line 27)
`random.nextInt(MAX_CHILDREN)` → 2	Root: # children (Line 12)
`random.nextInt(1, MAX_STRLEN)` → 3	Child 1: `name` length (Line 24)
`random.nextChar()` → ‘`b`’	Child 1: `name[0]` (Line 27)
`random.nextChar()` → ‘`a`’	Child 1: `name[1]` (Line 27)
`random.nextChar()` → ‘`r`’	Child 1: `name[2]` (Line 27)
`random.nextInt(MAX_CHILDREN)` → 0	Child 1: # children (Line 12)
`random.nextBool()` → True	Child 1: embed text? (Line 17)
`random.nextInt(1, MAX_STRLEN)` → 2	Child 1: `text` length (Line 24)
`random.nextChar()` → ‘`H`’	Child 1: `text[0]` (Line 27)
`random.nextChar()` → ‘`i`’	Child 1: `text[1]` (Line 27)
`random.nextInt(1, MAX_STRLEN)` → 3	Child 2: `name` length (Line 24)
`random.nextChar()` → ‘`b`’	Child 2: `name[0]` (Line 27)
`random.nextChar()` → ‘`a`’	Child 2: `name[1]` (Line 27)
`random.nextChar()` → ‘`z`’	Child 2: `name[2]` (Line 27)
`random.nextInt(MAX_CHILDREN)` → 0	Child 2: # children (Line 12)
`random.nextBool()` → False	Child 2: embed text? (Line 17)
`random.nextBool()` → False	Root: embed text? (Line 17)

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

$x_1 =$ `<foo><bar>Hi</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:

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

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 document 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.

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

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

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:

$$x_2 = \texttt{<}\underline{\texttt{W}}\texttt{oo><bar>Hi</bar><baz /></}\underline{\texttt{W}}\texttt{oo>}.$$

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:

$$\sigma_3 = \texttt{0000 0010} \ \ldots \quad \underbrace{\texttt{0000 0001}}_{\texttt{nextInt(MAX\_CHILDREN)}\rightarrow 1} \quad \ldots \ \texttt{0000 0000}.$$

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

$$x_3 = \texttt{<foo><bar>Hi</bar>■</foo>}$$

(■ 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>Hi</bar>`—is identical to the one in x_1. The parametric generator thus enables a structured mutation in the DOM tree —here, 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.

As a final example, suppose σ_1 is mutated as follows:

$$\sigma_4 = \texttt{0000 0011} \ \ldots \ \underbrace{\texttt{0000 0001}}_{\texttt{nextBool()}\rightarrow\text{True}} \ \underbrace{\texttt{0000 0000}}_{\texttt{nextInt(1,...)}\rightarrow 1} \ .$$

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, to generate the content of the embedded text, the generator needs more parameter values than σ_4 contains. Zest deals with this by appending pseudo-random values to the end of the parameter sequence on demand. It uses a fixed random seed to maintain determinism. For example, suppose the sequence is extended as:

$$\sigma_4' = \texttt{0000} \ \ldots \ \ \texttt{0001 0000 0000} \ \underbrace{\texttt{0100 1100}}_{\texttt{nextChar()}\rightarrow\texttt{'H'}} \ \ \underbrace{\texttt{0000 0000}}_{\texttt{nextBool()}\rightarrow\text{False}}$$

Then the parametric generator would produce the test-input:

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

6.2.2 Feedback-Directed Parameter Search

Algorithm 7 shows the Zest algorithm, which guides parametric generators to produce inputs that get deeper into the semantic analysis stage of programs using *validity-guided parameter*

Algorithm 7 The Zest algorithm, pairing parametric generators with validity-guided parameter search. Additions to Algorithm 1 highlighted in grey.

```
Input: program $p$, generator $q$
Output: a corpus of automatically generated test inputs
1: $\mathcal{S} \leftarrow \{\textsc{random}\}$
2: $g \leftarrow \textsc{makeParametric}(q)$
3: $\textit{totalCoverage} \leftarrow \textsc{initCoverage}(g(\mathcal{S}))$
4: $\textit{validCoverage} \leftarrow \textsc{initCoverage}(\{g(s) | s \in \mathcal{S} \wedge p(g(s)).\textit{result} = \textsc{Valid}\})$
5: repeat
6:   for $\textit{param}$ in $\mathcal{S}$ do
7:     with probability $\textsc{fuzzProb}(\textit{param})$ do
8:       for $1 \leq i \leq \textsc{numChildren}(\textit{param})$ do
9:         $\textit{candidate} \leftarrow \textsc{mutate}(\textit{param}, \mathcal{S})$
10:        $\textit{input} \leftarrow g(\textit{candidate})$
11:        $\textit{coverage}, \textit{result} \leftarrow \textsc{Execute}(p, \textit{input})$
12:        if $\textit{coverage} \not\subseteq \textit{totalCoverage}$ then
13:          $\mathcal{S} \leftarrow \mathcal{S} \cup \{\textit{candidate}\}$
14:          $\textit{totalCoverage} \leftarrow \textit{totalCoverage} \cup \textit{coverage}$
15:        if $\textit{result} = \textsc{Valid}$ and $\textit{coverage} \not\subseteq \textit{validCoverage}$ then
16:          $\mathcal{S} \leftarrow \mathcal{S} \cup \{\textit{candidate}\}$
17:          $\textit{validCoverage} \leftarrow \textit{validCoverage} \cup \textit{coverage}$
18: until given time budget expires
19: return $g(\mathcal{S})$
```

search. Zest builds on Algorithm 1, but acts on parameter sequences rather than the raw inputs to the program and keeps track of the coverage achieved by *semantically valid inputs*. Again, the differences between Algorithms 7 and 1 are highlighted 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 it automatically converts to a parametric generator g (Line 2). 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; see Section 6.3 for details. New inputs are generated by running the sequences through the parametric generator (Line 10). The program p is then executed on each input. 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. An input is considered invalid if it leads to a violation of any assumption in the test harness (e.g. Line 3 in Figure 6.1), which is how we capture application-specific semantic validity. More generally, the exit code of the program under test can be used to determine validity.

As in Algorithm 1, a newly generated parameter sequence is added to the set $\mathcal{S}$ at Lines 12–14 of Algorithm 7 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 Zest adds the parameter sequence to $\mathcal{S}$ and updates the cumulative valid coverage variable *validCoverage* at Lines 15–17. Adding these new-coverage valid parameter sequences to $\mathcal{S}$ ensures Zest keeps mutating valid inputs that exercise core program functionality. This should bias the search towards generating even more valid inputs, and in turn, increase code coverage in the semantic analysis stage.

6.3 Implementation

Zest is implemented on top of the open-source JQF platform [148], which provides a framework for specifying algorithms for feedback-directed fuzz testing of Java programs. JQF dynamically instruments Java classes in the program under test using the ASM bytecode-manipulation framework [146] via a `javaagent`. The instrumentation allows JQF to observe code coverage events, e.g. the execution of program branches and invocation of virtual method calls.

Fuzzing "guidances" can plug into JQF to provide inputs and register callbacks for listening to code coverage events. JQF includes the guidances `AFLGuidance` and `NoGuidance`, which we use in our evaluation in Section 6.4. `AFLGuidance` uses a proxy program to exchange program inputs and coverage feedback with the external AFL tool; the overhead of this inter-process communication is a negligible fraction of the test execution time. `NoGuidance` randomly samples inputs from `junit-quickcheck` [98] generators without using coverage feedback. We implement `ZestGuidance` in JQF, which biases these generators using Algorithm 7.

`junit-quickcheck` provides a high-level API for making random choices in the generators, such as generating random integers or selecting random items from a collection. All these methods indirectly rely on the underlying JDK method `java.util.Random.next(int bits)`, which returns bits from a pseudo-random stream. Zest replaces this pseudo-random stream with a stored parameter sequence, which it extends on-demand.

Since `java.util.Random` polls byte-sized chunks from its underlying stream of pseudo-random bits, Zest performs mutations on the parameter sequences (Algorithm 7, 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.

6.4 Evaluation

We evaluate Zest by measuring its effectiveness in testing the semantic analysis stages of five benchmark programs. We compare Zest with two baseline techniques: AFL and `junit-quickcheck`, referred to as simply *QuickCheck*. AFL is known to excel in exercising the syntax analysis stage via coverage-guided fuzzing of raw input strings. We use version 2.52b, skipping the deterministic mutation stages. 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 The evaluation uses following Java benchmarks as test program:

1. Apache Maven [9] (99k LoC): The test reads a `pom.xml` file and converts it into an internal `Model` structure. 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 [10] (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 [11] (89k LoC) compiles JavaScript to Java bytecode. The test driver calls `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 [121]. An input is valid if BCEL finds no errors up to Pass 3A verification.

Experimental Setup We make the following design decisions in our experiments.

Duration: We run each test-generation experiment for *3 hours*. Researchers have used various timeouts to evaluate random test generation tools, from 2 minutes [147, 75] to 24 hours [46, 106]. We chose 3 hours as a middle ground. 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.

Table 6.1: 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`

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 [162].

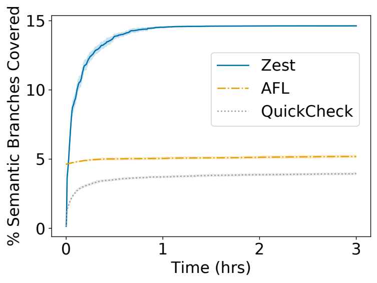
Researchers have found success using varying strategies ranging from large seed corpora to single empty files [106]. In our evaluation, we provide AFL one valid seed input per benchmark that covers various domain-specific semantic features. For example, in the Closure and Rhino benchmarks, we use the entire React.JS library [12] 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.

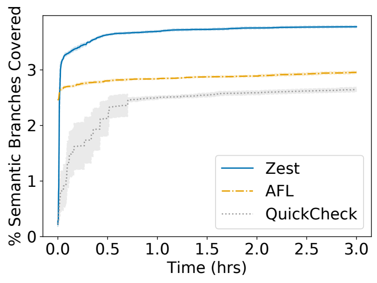
Generators: Zest and QuickCheck use hand-written input generators. For Maven and Ant, we use an XML document generator similar to Figure 6.2, of around 150 lines of Java code. It generates strings for tags and attributes by randomly choosing strings from a list of string literals scraped from class files 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 by Rohan Padhye in less than two hours each. Although these generators produce syntactically valid inputs, no effort was made to produce semantically valid inputs; doing so can be a complex and tedious task [181].

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.

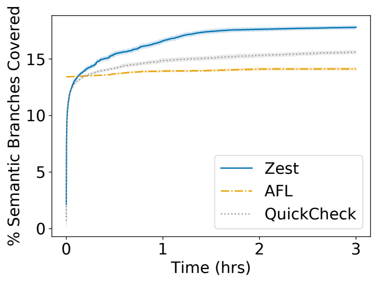
Syntax and Semantic Analysis Stages in Benchmarks Zest is specifically designed to exercise the semantic analysis stages of programs. To evaluate Zest's effectiveness in this regard, we manually identify the components of our benchmark programs which correspond to syntax and semantic analysis stages. Table 6.1 lists prefix patterns that we match on the fully-qualified names of classes in our benchmarks to classify them in either stage. Section 6.4.1



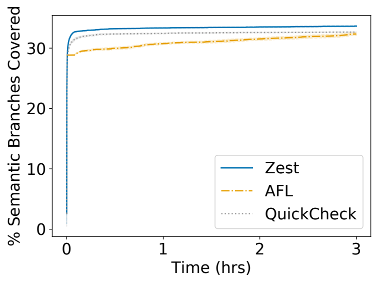
(a) maven



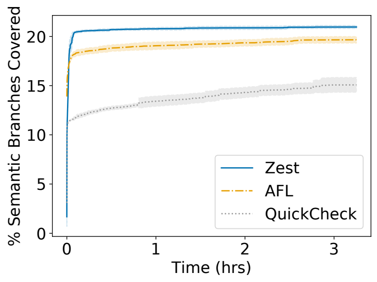
(b) ant



(c) closure



(d) rhino



(e) bcel

Figure 6.3: Percent coverage of *all* branches in semantic analysis stage of the benchmark programs. Lines designate means and shaded regions 95% confidence intervals.

evaluates the code coverage achieved within the classes identified as belonging to the semantic analysis stage. Section 6.4.2 evaluates the bug-finding capabilities of each technique for bugs that arise in the semantic analysis classes. Section 6.5 discusses some findings in the *syntax* analysis classes, whose testing is outside the scope of Zest.

6.4.1 Coverage of Semantic Analysis Classes

Instead of relying on our own instrumentation, we use a third party tool, the widely used Eclemma-JaCoCo [96] library, for measuring code coverage in our Java benchmarks. Specifically, we measure *branch coverage* within the semantic analysis classes from Table 6.1; 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 6.3 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× as many semantic branches covered by the best baseline technique for Maven (Figure 6.3a). When looking at our JavaScript benchmarks, we see that Zest's advantage over QuickCheck is more slight in Rhino (Figure 6.3b) than in Closure (Figure 6.3c). 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 through feedback-driven parameter search.

Note that in some benchmarks AFL has an edge in coverage over QuickCheck (Figure 6.3a, 6.3b, 6.3e), and vice-versa (Figure 6.3c, 6.3d). For BCEL, this may be because the input format is a compact syntax, on which AFL 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 baselines in all these cases suggests that it is more robust to generator quality than QuickCheck.

6.4.2 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 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. Further, 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 6.1). Of the 20 unique bugs we found, 10 were syntactic and 10 were semantic.

Table 6.2: The 10 new bugs found in the semantic analysis stages of benchmark programs. Zest, AFL, and QuickCheck (QC) are evaluated on the *mean time to find* (MTF) each bug across the 20 3-hour experiments as well as their *reliability*, which is the percentage of the 20 experiments in which the bug was triggered at least once. For each bug, the circled tool is statistically significantly more effective at finding the bug than uncircled 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%

We evaluate Zest on *semantic bug* discovery. Table 6.2 enumerates the 10 semantic bugs found across our benchmark programs. Each bug has a unique ID—represented as a circled letter—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: has 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 does not find the other 5 within the 3-hour time limit. This is likely because AFL's mutation strategy results in 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.

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 right after a `break` statement is unreachable code, variable declarations in JavaScript are hoisted and thus cannot be removed. Zest is the only technique that finds this bug, with inputs like:

```
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 issue is perhaps rooted in Closure's attempt to evaluate `undefined[undefined](...)` at compile time. The developers acknowledged the bug. 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 its validity-guided parameter search.

6.5 Discussion

Zest and plain generator-based fuzzing tools such as QuickCheck make use of generators in order to synthesize 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, 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 6.1: 3 in Maven, 6 in BCEL, and 1 in Rhino. These bugs were triggered by syntactically invalid inputs, which Zest and QuickCheck do not produce.

This is a notable contrast between Zest and FairFuzz, the latter of which still generates syntactically invalid inputs, as discussed in Chapter 5. One approach to taking the best of both worlds would be to mainly conduct Zest's mutation strategy, but occasionally perform the byte-level mutation strategy in order to explore the syntactic analysis stages of programs.

Overall, this result, paired with the results from Chapter 5, further highlights the key observation of Part II: more structured input mutations are required to explore the so-called "deeper" stages of programs, which often contain the core logic of the program under test. This chapter also introduced the notion guided generator-based fuzzing: Zest is effectively generator-based fuzzing guided by coverage and validity. The next and final part of this dissertation dives deeper into the power of guided generator-based fuzzing, and its applicability to search problems beyond testing.

Part III

Distribution Tuning of Generators

Chapter 7

RLCheck: Valid Inputs via Reinforcement Learning

Chapter 6 illustrated how integrating generator-based and coverage-guided fuzzing enabled better coverage of program under test's core logic. From the lens of coverage-guided fuzzing, bringing in the input generator enabled higher-level mutations, and thus, broader exploration.

However, there is another lens from which to view Zest: that of guided generator-based fuzzing. Generator-based fuzzing, also called *property-based testing*, is a popular approach to validate the core logic of the program. Unlike generic coverage-guided fuzzing, which is typically used to validate that a program does not violate universal program correctness oracles (e.g., no array-out-of-bounds reads), generator-based fuzzing is typically used to validate a particular logical property of the program under test, i.e. that $\forall x \in \mathcal{X} : P(x) \Rightarrow Q(x)$.

Recall from the introduction that using a generator-based fuzzer requires two main steps. First, the user needs to write a *parameterized test driver*, i.e., the programmatic representation of $P(x) \Rightarrow Q(x)$. Second, the user needs to specify the *generator* for $x \in \mathcal{X}$. A generator for $\mathcal{X}$ is a non-deterministic program returning inputs $x \in \mathcal{X}$.

For this testing to be effective, the generator must produce a *diverse* set of inputs $x \in \mathcal{X}$ satisfying the validity constraint $P(x)$. This gives rise to a central conflict in generator-based fuzzing [111]. On the one hand, a simple generator is easier for the user to write, but not necessarily effective. On the other hand, a generator that produces diverse valid inputs is good for testing, but very tedious to write. Further, generators specialized to a particular validity function $P(x)$ cannot be reused to test other properties on $\mathcal{X}$ with different validity constraints, say $P'(x)$. We thus want to solve the following problem: given a generator G of inputs $x \in \mathcal{X}$ and a validity function $P(x)$, *automatically* guide the generator to produce a variety of inputs x satisfying $P(x)$.

In the previous chapter, we saw an approach that utilized coverage-guided fuzzing with additional semantic validity feedback to create more inputs x satisfying $P(x)$. In this chapter, we look at the problem with a more fine-grained approach. First, we formalize the problem of guiding the random choices made by a generator for effective testing as the *diversifying guidance problem*. Second, we notice that the diversifying guidance problem is similar to problems solved by reinforcement learning: given a sequence of prior choices (state), what is the next choice (action) that the generator should make, in order to maximize the probability of generating a new x satisfying $P(x)$ (get high reward)? We thus explore whether reinforcement learning can solve the diversifying guidance problem. We present an on-policy table-based approach, *RLCheck* [163], which adapts its choices on-the-fly during testing time to quickly produce a diversity of inputs x satisfying $P(x)$.

7.1 Motivation

Let us first consider a concrete example that illustrates the problem solved by *RLCheck*.

The test driver in Figure 7.1, `test_insert` (Line 13), takes a binary tree `tree` and an integer `to_add` as input. If `tree` is a binary search tree (Line 14), the driver inserts `to_add` into `tree` (Line 15) and asserts that `tree` is still a binary search tree after the insert (Line 16). The **`assume`** at Line 14 terminates the test silently if `tree` is not a binary search tree. The **`assert`** at Line 16 is violated if `tree` is not a binary search tree after the insertion. Thus, the test driver implements $P(x) \Rightarrow Q(x)$ for the validity constraint $P(x) =$"x is a binary search tree" and the post-condition $Q(x) =$"after inserting `to_add`, x is a binary search tree"—by raising an assertion failure when $P(x) \Rightarrow Q(x)$ is falsified.

The user must now specify how to generate random inputs for this driver. To do this, they must write or select a *generator*, a non-deterministic function that returns a random input of a given type each time it is executed. Generator-based fuzzing frameworks typically provide generators for basic types such as primitive types and predefined containers of primitives (e.g. `generate_int` in Figure 7.1). If the test function takes a user-defined data structure, such as the `tree` in Figure 7.1, Line 13, the user writes their own generator. For many types, writing a basic generator is fairly straightforward. In Figure 7.1, `generate_tree` generates a random binary tree by (1) choosing a value for the root node (Line 4), (2) choosing whether or not to add a left child (Line 6) and recursively calling `generate_tree` (Line 7), and (3) choosing whether or not to add a right child (Line 8) and recursively calling `generate_tree` (Line 9). We have deliberately kept this generator simple in order to have a small motivating example.

The user can now run `test_insert` on many different trees to try and validate $P(x) \Rightarrow Q(x)$. The **`assume`** in Line 14 effectively filters out invalid (non-BST) inputs. Unfortunately, this rejection sampling is not an effective strategy if $P(x)$ is too strict. If the generator has no knowledge of $P(x)$, it will, first of all, very rarely generate valid inputs. So, in a fixed time budget, very few valid inputs will be generated. The second issue is that the generator may not generate very diverse valid inputs. That is, the only valid inputs the generator has a non-negligible probability of generating may be *very small* valid inputs; these will not exercise a variety of behaviors in the code under test. For example, out of 1000 generated binary trees, the generator in Figure 7.1 only generates 20 binary search trees of size ≥ 3, and only one binary search tree of size 4 and 5, respectively. Overall, the generator has very low probability of generating complex valid inputs, which greatly decreases the efficacy of the generator-based fuzzing.

```
1  from generators import generate_int
2
3  def generate_tree(depth=0):
4      value = random.Select([0, 1, ..., 10])
5      tree = BinaryTree(value)
6      if depth < MAX_DEPTH and random.Select([True, False]):
7          tree.left = generate_tree(depth+1)
8      if depth < MAX_DEPTH and random.Select([True, False]):
9          tree.right = generate_tree(depth+1)
10     return tree
11
12 @given(tree = generate_tree, to_add = generate_int)
13 def test_insert(tree, to_add):
14     assume(is_BST(tree))
15     BST_insert(tree, to_add)
16     assert(is_BST(tree))
```

Figure 7.1: Test driver and generator for generator-based fuzzing. `generate_tree` generates a random binary tree, and `test_insert` tests whether inserting a given integer into a given tree preserves the binary search tree property. `random.Select(D)` returns a random value from `D`.

Fundamentally, the input generated by a generator is governed by the choices taken at various *choice points*. For example, in Figure 7.1, at Line 4, the generator makes a *choice* of which integer value to put in the current node, and it chooses to make a left or right child at Lines 6 and 8, respectively. Depending on the prior sequence of choices taken by the generator, only a subset of the possible choices at a particular choice point may result in a valid input. For example, if $P(x)$ is the binary search tree invariant, when generating the right child of a node with value n, the only values for the child node that can result in a valid BST are those greater than n. While narrowing the choice space in this manner is straightforward for BSTs, manually encoding these restrictions is tedious and error-prone for complex real-world validity functions.

Overall, we see that the problem of guiding G to produce many valid inputs boils down to the problem of narrowing the choice space at each choice point in the generator. We call this the *diversifying guidance problem*. We formalize this problem in Section 7.2 and propose a reinforcement-learning-based solution in Section 7.3.

7.2 Problem Definition

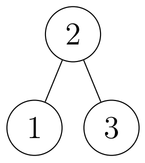
In generator-based fuzzing, a generator G is a non-deterministic program returning elements of a given space $\mathcal{X}$. For example, in Figure 7.1, $\mathcal{X}$ is the set of binary trees of depth up to `MAX_DEPTH` with nodes having integer values between 0–10, inclusive.

In particular, a generator G's non-determinism is entirely controlled by the values at returned at different *choice points* in the generator. A choice point p is a tuple (ℓ, C) where $\ell \in \mathbb{L}$ is a program location and $C \subseteq \mathcal{C}$ is a finite domain of choices. For example, there are three choice points in the generator in Figure 7.1:

- (Line 4, `[0, 1, ..., 10]`): the choice of node value;
- (Line 6, `[True, False]`): whether to generate a left child; and
- (Line 8, `[True, False]`): whether to generate a right child.

During execution, each time the generator reaches a choice point (ℓ, C), it makes a choice $c \in C$. Every execution of the generator, and thus, every value produced by the generator, corresponds to a sequence of choices made at these choice points, say $c_1, c_2, \ldots, c_n$.

For example, the execution through `generate_tree` in Figure 7.1 which produces the tree

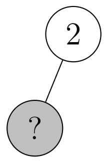


corresponds to the following sequence of choices $c_1, c_2, \ldots c_9$:

Choice Index	Choice Taken	Choice Point
c_1	`2`	(Line 4, `[0, 1, ..., 10]`)
c_2	`True`	(Line 6, `[True, False]`)
c_3	`1`	(Line 4, `[0, 1, ..., 10]`)
c_4	`False`	(Line 6, `[True, False]`)
c_5	`False`	(Line 8, `[True, False]`)
c_6	`True`	(Line 8, `[True, False]`)
c_7	`3`	(Line 4, `[0, 1, ..., 10]`)
c_8	`False`	(Line 6, `[True, False]`)
c_9	`False`	(Line 8, `[True, False]`)

As the generator executes, each time it reaches a choice point $p = (\ell, C)$, it will have already made some choices $c_1, c_2, \ldots c_k$. Traditional random generators, like the one in Figure 7.1, will simply choose a random $c \in C$ at choice point p regardless of the prefix of choices $c_1, c_2, \ldots c_k$.

Going back to our running example, suppose the generator has reached the choice point choosing the value of the left child of 2, i.e. choosing what to put in the ? in this tree:



That is, the generator has made the choices $[c_1 = $ `2`$, c_2 = $ `True`$]$, and must now choose a value from 0−10 at the choice point in Line 4. The generator is equally likely to pick any number in this range. Since only 2 of the 11 numbers from 0−10 are smaller than 2, it has at most an 18% chance of producing a valid BST.

To increase the probability of generating valid inputs, the choice at this point should be made not randomly, but according to a *guide*. In particular, according to a guide which restricts the choice space to only those choices which can result in a binary search tree. First, we formalize the concept making choices according to a guide.

Definition 18 (Following a Guide)**.** We say that a generator G *follows* a guide $\gamma : \mathcal{C}^* \times P \times \mathbb{N} \to C$ if: during its t^{th} execution, given a sequence of past choices $\sigma = c_1, c_2, \dots, c_k$, and the current choice point $p = (\ell, C)$, the generator G makes the choice $\gamma(\sigma, p, t)$.

Suppose we have a validity function $\nu : \mathcal{X} \to \{\texttt{True}, \texttt{False}\}$ which maps elements output by the generator to their validity. For example, `is_BST` is a validity function for the generator in Figure 7.1. The *validity guidance problem* is the problem of finding a guide that leads the generator to produce valid elements of $\mathcal{X}$:

Definition 19 (Validity Guidance Problem)**.** Let G be a generator producing elements in space $\mathcal{X}$. Let $\nu : \mathcal{X} \to \{\texttt{True}, \texttt{False}\}$ be a validity function. The *validity guidance problem* is the problem of creating a guide γ such that:

$$\text{if } G \text{ follows } \gamma, \text{ then } \nu(x) = \texttt{True} \text{ for any } x \in \mathcal{X} \text{ generated by } G.$$

Note that a solution to the validity guidance problem is not necessarily useful for testing. In particular, the guide γ could simply hard-code a particular sequence of choices through the generator which results in a valid element $x \in \mathcal{X}$. Instead, we want to generate valid inputs with diverse *characteristics*. For example, we may want to generate unique valid inputs, or valid inputs of different lengths, or valid inputs that exercise different execution paths in the test program. We use the notation $\xi(x)$ to denote an input's characteristic of interest, such as identity, length, or code coverage.

Definition 20 (diversifying guidance problem)**.** Let G be a generator producing elements in space $\mathcal{X}$. Let $\nu : \mathcal{X} \to \{\texttt{True}, \texttt{False}\}$ be a validity function and ξ be a characteristic function. The *diversifying guidance problem* is the problem of creating a guide γ such that:

$$\text{if } G \text{ follows } \gamma \text{ and } X_T \subseteq \mathcal{X} \text{ is the set of inputs generated by } G \text{ after } T \text{ executions,}$$
$$|\{\xi(x) : x \in X_T \wedge \nu(x)\}| \text{ is maximized.}$$

If ξ is the identity function, then a solution to the diversifying guidance problem is a guide which maximizes the number of unique valid inputs generated.

7.3 The *RLCheck* Algorithm

In this section we describe *RLCheck*, which solves the diversifying guidance problem by using reinforcement learning to guide the choices. We begin with background on Monte Carlo Control [173], and then describe how Monte Carlo Control can be used to solve the diversifying guidance problem.

7.3.1 Reinforcement Learning

We first define a version of the problem solved by reinforcement learning which is relevant to our task at hand. We use a slightly nontraditional notation for consistency with the previous and next sections. What we call *choices* are typically called *actions*, and what we call a *learner* is typically called an *agent*.

We assume an learner in some environment. The learner can perceive the *state* s of the environment, where s is in some set of states $\mathcal{S}$. At the first point in time, the learner is at an initial state $s_0 \in \mathcal{S}$. At each point in time, the learner can make a choice $c \in \mathcal{C}$ which will bring it to some new state $s' \in \mathcal{S}$. Eventually, the agent gets into some terminal state $s_{term} \in \mathcal{S}$, indicating the end of an *episode*. An episode is the sequence of *(state, choice)* pairs made from the beginning of time up to the terminal state, i.e.:

$$e = (s_0, c_0), (s_1, c_1), \dots (s_T, c_T),$$

where the choice c_T in state s_T brings the learner to the terminal state s_{term}. Finally, we assume we are given a reward r for a given episode e. A larger reward is better.

The problem to solve is the following. Given a state space $\mathcal{S}$, choices $\mathcal{C}$, and reward r, find a policy π which maximizes the expected reward to the learner. That is, find a π such that if the learner, at each state $s \in \mathcal{S}$, makes the choice $c = \pi(s)$, then the expected reward $\mathbb{E}_{\pi,e}[r]$ from the resulting episode e is maximized.

7.3.1.1 Monte Carlo Control

One approach to solving the policy-learning problem above is by on-policy *Monte Carlo Control* [173]. The technique is *on-policy* because the policy the learner is optimizing is the same one it is using to control its actions. Thus, a Monte Carlo Control learner L defines both a policy π, where $\pi(s)$ outputs a choice c for the given state s, as well as an update procedure that improves π after each episode.

Algorithm 8 shows pseudocode for a Monte Carlo Control (MCC) learner L. In the algorithm, we subscript the choice space, state space, and Q and *counts* with L to emphasize these are independent for each MCC learner. We will drop the subscript L when talking about a single learner. The basic idea is as follows.

We are trying to learn a policy π for state space $\mathcal{S}$ and choices $\mathcal{C}$. The policy is ε-greedy: with probability ε it makes random choices (Line 7), otherwise it makes the choices that will maximize the value function, Q (Line 9).

Algorithm 8 A Monte Carlo Control learner L. Implements a policy π_L and an update function UPDATE_L which updates π_L towards the optimal policy after each episode.

```
Input: choice space $\mathcal{C}_L$, state space $\mathcal{S}_L$, and $\varepsilon_L$
 1: $e_L \leftarrow []$                                          ▷ initialize episode
 2: for $(s, c) \in \mathcal{S}_L \times \mathcal{C}_L$ do
 3:     $counts_L[s, c] \leftarrow 0$
 4:     $Q_L[s, c] \leftarrow 0$
 5: procedure $\pi$(state $s$)
 6:     if $\text{UNIFORMRANDOM}() < \varepsilon$ then
 7:         $c \leftarrow \text{RANDOM}(\mathcal{C})$
 8:     else
 9:         $c \leftarrow \arg\max_{c \in \mathcal{C}_L} Q_L[s, c]$          ▷ break ties arbitrarily
10:     $e_L \leftarrow \text{APPEND}(e_L, (s, c))$
11:     return $choice$
12: procedure UPDATE(reward $r$)
13:     $T \leftarrow \text{LEN}(e_L)$
14:     for $0 \leq t < T$ do
15:         $s, c \leftarrow e_L[t]$
16:         $Q_L[s, c] \leftarrow \frac{r + Q_L[s,c] \cdot (counts_L[s,c])}{counts_L[s,c]+1}$          ▷ update avg. reward
17:         $counts_L[s, c] \leftarrow counts_L[s, c] + 1$
        $e_L \leftarrow []$
```

The value function $Q[s, c]$ models the expected reward at the end of the episode from the choice c in state s. It is initialized to 0 for each (s, c) pair (Line 4), so the first episode follows a totally random policy. $Q[s, c]$ is exactly the *average rewards* seen for each episode e containing (s, c). Thus, at the end of each episode e, for each $(s, c) \in e$ (Line 15), the running average for the rewards observed with (s, c) is updated to include the new reward r (Line 16).

If the reward function producing r is *stationary* (i.e., fixed), this update procedure always improves the policy. That is, if π is the original policy, and π' is the policy after the update, the expected reward from a learner following π' is greater than or equal to the expected reward from a learner following π. Sutton and Barto [173] provide a proof.

Algorithmic changes We make a few algorithmic changes on top of regular MCC. First, we update an episode with a single reward r which is distributed to all state action pairs. This is because, as will be seen in later sections, we only observe rewards at the end of an episode i.e there are no intermediate rewards provided in our method. Second, we do not use a discount factor on the reward r. This is because the sequence of choices in an input generation, do not lend themselves to a natural absolute ordering. We cannot assume later decisions are more important than earlier ones, which the discount factor implicitly does.

7.3.2 *RLCheck*: MCC with Diversity Reward

We now return to our problem space of generating inputs with a generator G. Notice that the *guides* we defined in Definition 18 have a similar function to the learners in Section 7.3.1: given some state (σ, p, t), make a choice c.

This leads to the natural idea of implementing a guide as an MCC learner, rewarding the learner with some $r(x)$ after the generator produces input x. However, note that for the guide, at each choice point $p = (\ell, C)$, only a subset of choices $C \subseteq \mathcal{C}$ can be taken. Further, each choice point has a unique task: for example, choosing whether to generate a left child (Figure 7.3, Line 8) or a right child (Figure 7.3, Line 11). Thus, it is natural to define a separate learner L_p for each choice point p, and call $\textsc{update}_{L_p}$ once for each learner after every execution of the generator.

Note that Section 7.2 defined a guide as making choices based on a sequence $\sigma \in \mathcal{C}^*$, while Section 7.3.1 assumed a finite set of states $\mathcal{S}$. Thus, we need a state abstraction function:

Definition 21 (State Abstraction Function)**.** A *state abstraction function* $A : \mathcal{C}^* \to \mathcal{S}$ *for a* generator G is a deterministic function mapping an arbitrary-length choice sequence σ to a finite state space $\mathcal{S}$. A can rely on G to retrieve, for any $c_i \in \sigma$, the choice point p at which c_i was made.

With this, we can define a Monte Carlo Control Guide:

Definition 22 (Monte Carlo Control Guide)**.** Assume a generator G producing inputs in $\mathcal{X}$, a state abstraction function A, and a reward function $r : \mathcal{X} \to \mathbb{R}$. A Monte Carlo Control Guide γ consists of a set of Monte Carlo control *learners*, $\{L_p\}$. Each learner L_p is associated with a choice point $p = (\ell, C)$ in G.

Let $\pi^{(t)}_{L_p}$ be L_p's policy after $t-1$ calls to $\textsc{update}_{L_p}$ (ref. Algorithm 8). Then γ is:

$$\gamma(\sigma, p, t) = \pi^{(t)}_{L_p}(A(\sigma)).$$

Finally, after G produces an input x, the guide γ calls $\textsc{update}_{L_p}(r(x))$ for each learner L_p.

Now, to use a Monte Carlo Control guide (MCC guide) to solve the diversifying guidance problem, only (1) the state abstraction function A (ref. Section 7.3.3) and (2) the reward function r need to be specified. We construct a reward function as follows.

Let ν be the validity function and ξ the characteristic function of interest. If X be the set of inputs previously generated by G, then let $\Xi = \{\xi(x') : x' \in X\}$ be the set of characteristics of all the previously generated inputs. Then the reward function r is:

$$r(x) = \begin{cases} R_{unique} \text{ if } \nu(x) \wedge \xi(x) \notin \Xi \\ R_{valid} \text{ if } \nu(x) \wedge \xi(x) \in \Xi \\ R_{invalid} \text{ if } \neg\nu(x) \end{cases} \tag{7.1}$$

Our technique, ***RLCheck***, is thus: make a generator G follow an MCC Guide with the reward function r above.

Note that this reward function is *non-stationary*, that is, it is not fixed across time. If $X = \emptyset$, then generating any $x \in \mathcal{X}$ such that $\nu(x)$ holds will result in the reward R_{unique}; re-generating the same x in the next step will only result in the reward R_{valid}. This means the assumptions underlying the classic proof of policy improvement do not hold [173]. Thus, *RLCheck*'s guide is not guaranteed to improve to an optimal policy. Instead, it practices a form of online learning, adjusting its policy over time.

7.3.3 State Abstraction

A key element in enabling MCC to solve the diversifying guidance problem is the state abstraction function (Definition 21), which determines the current state given a sequence of past choices. The choice of A impacts the ability of the MCC guide to learn an effective policy. On one extreme, if A collapses all sequences into the same abstract state (e.g., $A(\sigma) = 0$), then a learner L_p essentially attempts to find a single best choice $c \in \mathcal{C}_{L_p}$ for choice point p, regardless of state. On the other extreme, if A is the identity function (i.e., $A(\sigma) = \sigma$), then the state space is infinite; so for every previously unseen sequence of choices σ, the learner's policy is random.

The ideal A is the abstraction function that maximizes expected reward. However, computing such an A is not tractable, since it requires inverting an arbitrary validity function $\nu(x)$. Instead, we apply the following heuristic: in many input generators, a good representation for the state S_n after making the n^{th} choice c_n is some function of a past subsequence of choices that *influence* the choice c_n. The meaning of influence depends on the type of input being generated and the nature of the validity function.

For example, on the left of Figure 7.2 is a partially generated binary tree. On the right are the choices made in the binary-tree generator (ref. Fig. 7.1) leading to this partial tree ($c_1 = 2$, $c_2 =$ `True`, $c_3 = 1$, $c_4 =$ `False`, $c_5 =$ `False`, $c_6 =$ `True`), arranged by *influence*. A choice in the construction of a child node is *influenced* by choices constructing its parent.

With this influence heuristic, the best value for the next choice c_7, which determines the value assigned to the right child, should depend on the choice c_1 (which decided that the root node had value 2) and the choice c_6 (which made the decision to insert a right child). The best value for this choice c_7 does not necessarily depend on choices c_2–c_5, which were involved in the creation of the left sub-tree. Therefore, the state S_6, in which the choice c_7 is to be made, can be represented as a sequence $[f_v(c_1), f_r(c_6)]$. Here, f_v is a function associated c_1's choice point (the node-value choice point at Line 4 of Fig. 7.1) and f_r is a function associated with c_6's choice point (the right-child choice point at Line 8 of Fig. 7.1). In Figure 7.2, the state S_6 after applying these functions is `[2, R]`; we will define the functions f_v and f_r for this figure later in this section.

An additional consideration when representing state as a sequence derived from past choices is that such sequences can become very long. We need to make sure the state space is finite. Again, a reasonable heuristic is to use a trimmed representation of the sequence, which incorporates information from up to the last w choices that influence the current choice. w is a fixed integer that determines the size of a sliding *window*.

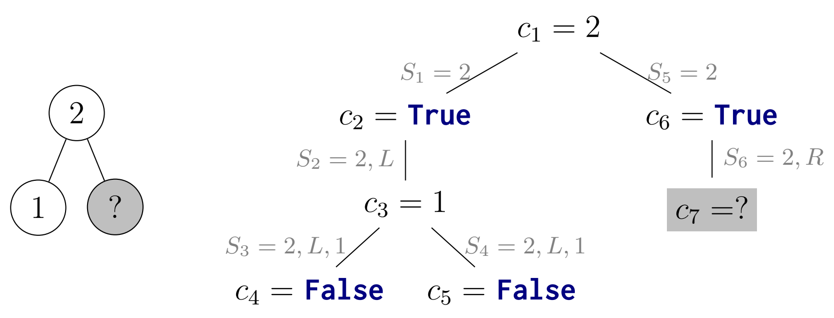


Figure 7.2: A partially-generated binary tree (left) and its corresponding choice sequence arranged by influence (right).

We can build a state abstraction function that follows these considerations in the following manner. First, build a choice abstraction function f_p for each choice point p, which maps each c to an abstract choice. Then, for $\sigma = c_1, c_2, \ldots, c_n$, build $S_n = A(\sigma)$ so that:

$$S_n = \begin{cases} \emptyset \text{ if } \sigma = \emptyset \\ tail_w(S_k :: f_p(c_n)) \text{ for some } k < n \text{ otherwise,} \end{cases}$$

where :: is the concatenation operator and $tail_w(s)$ takes the last w elements of s. Assume c_n was taken at choice point p.

We can build both very basic and very complex state abstractions in this manner.

For example, we can get $A(\sigma) = c_{n-w+1}, \ldots, c_{n-1}, c_n$ by taking $f_p = id$ for all and choosing $k = n-1$ always. This would be a simple sliding window of the last w choices.

The states S_1-S_6 that annotate the edges in Figure 7.2 are derived using the choice point abstraction functions $f_v(c) = c$ for the value choice point, $f_r(c) = R$ for the right child choice point, and $f_l(x) = L$ for the left child choice point. The k is chosen as $k =$"largest $k < n$ which is a choice from the parent node". While programatically deriving this k from a choice sequence σ is tedious, it is quite easy to do inline in the generator. The generator Figure 7.3 shows a modified version of the generator from Figure 7.1, which updates an explicit state value at each to compute exactly this state abstraction function (Lines 6, 8, 11); it also uses guides to select arbitrary values (Lines 5, 8, 11).

7.3.3.1 Case study

We evaluate the effect the state abstraction function has on the ability of *RLCheck* to produce unique valid inputs for the BST example. We evaluate three state abstraction functions:

- *Sequence*, the sliding window abstraction which retains choices from the sibling nodes, i.e. $A(\sigma) = c_{n-w+1}, \ldots, c_{n-1}, c_n$.
- *Tree L/R*, the abstraction function illustrated in Figure 7.2 and implemented in Figure 7.3.

```
def concat_tail(state, value):
  return (state + [value])[-WINDOW_SIZE:]

def gen_tree(state, depth=0):
  value = guide.Select([0, ..., 10], state, id=1)
  state = concat_tail(state, value)
  tree = BinaryTree(value)
  if depth < MAX_DEPTH and guide.Select([True, False], state, id=2):
    left_state = concat_tail(state, "L")
    tree.left = gen_tree(left_state,  depth+1)
  if depth < MAX_DEPTH and guide.Select([True, False], state, id=3):
    right_state = concat_tail(state, "R")
    tree.right = gen_tree(right_state, depth+1)
  return tree
```

Figure 7.3: Pseudo-code for a binary tree generator which follows `guide` and builds a tree-based state abstraction.

- *Tree*, which chooses k like Tree L/R but has $f_p = id$ for all choice points, and thus produces the same state for the left and right subtree of a node.

For example, taking $w = 4$ and the choices to be abstracted $c_1, \dots, c_6$ from Figure 7.2: *Sequence* will give `[1, False, False, True]`, *Tree* state will give `[2, True]`, and *Tree L/R* will give `[2, "R"]`.

We evaluate each of these abstraction techniques for generating BSTs with maximum depth 4 (i.e., 4 links), with $\varepsilon = 0.25$ and rewards (Eq. 7.1) $R_{invalid} = -1, R_{valid} = 0$, and $R_{unique} = 20$. We set $w = 4$ for the abstraction function: since there are at least two elements in the state for each parent node, this means the learners cannot simply memorize the decisions for the full depth of the tree.

Results Figures 7.4 and 7.5 show the results for our experiments. In each experiment we let each technique generate 100,000 trees. The results show the averages and standard errors over 10 trials. We compare to a baseline, *Random*, which just runs the generator from Figure 7.1. Figure 7.4 illustrates that no matter the state abstraction function chosen, *RLCheck* generates many more valid and unique valid inputs than the random baseline; *Tree L/R* generates 10× more unique valid inputs than random. Within the abstraction techniques, *Tree* generates the fewest unique valid inputs. *Sequence* appears to be better able to distinguish whether it is generating a left or right child than *Tree*, probably because the *Tree* state is identical for the left and right child choice points.

Tree L/R generates the fewest valid inputs, but the most unique valid inputs, 36% more than *Sequence*. These unique valid inputs are also more complex those generated with other

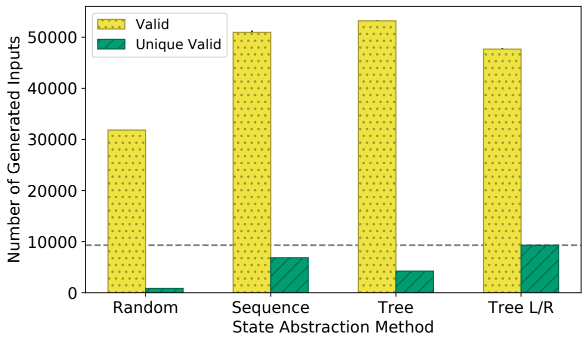


Figure 7.4: Number of (unique) valid inputs generated, by state abstraction. "Random" is a no-RL baseline.

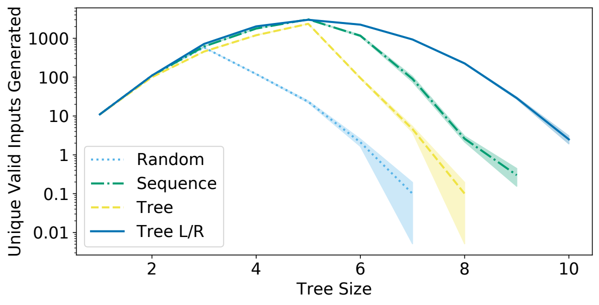


Figure 7.5: Distribution of unique valid tree sizes, by state abstraction. "Random" is a no-RL baseline.

state abstractions. Figure 7.5 shows, for each technique, the average number of unique valid trees generated of each size. Note the log scale. The tree size is the number of nodes in the tree. We see that *Tree L/R* is consistently able to generate orders of magnitude more trees of sizes > 5 than the other techniques. Since we reward uniqueness, the *RLCheck* is encouraged to generate larger trees as it exhausts the space of smaller trees. These results suggest that *Tree L/R* has enough information to generate valid trees, and then combine these successes into more unique valid trees.

Overall, we see that even with a naïve state abstraction function, *RLCheck* generates nearly an order of magnitude more unique valid inputs than the random baseline. However, a well-constructed influence-style state abstraction yields more diverse valid inputs.

7.4 Evaluation

In this section we evaluate how *RLCheck*, our MCC-based solution to the diversifying guidance problem, performs. In particular, we focus on the following research questions:

RQ1 Does *RLCheck* quickly find many diverse valid inputs for real-world benchmarks compared to state-of-the-art?

RQ2 Does *RLCheck* find valid inputs covering many different behaviors for real-world benchmarks?

RQ3 Does *RLCheck* have better bug-finding capabilities than the baselines?

RQ4 Does adding coverage feedback improve the ability of *RLCheck* to generate diverse valid inputs for real-world benchmarks?

Implementation To answer these research questions, we implemented Algorithm 8 in Java, and *RLCheck* on top of the open-source JQF [148] platform. JQF, highlighted in last chapter's implementation section, provides a mechanism for customizing input generation for QuickCheck-style property tests.

Baselines We compare *RLCheck* to two different methods: (1) junit-quickcheck [98], or simply QuickCheck, the baseline generator-based testing technique which calls the generator with a randomized guide; and (2) Zest discussed in Chapter 6. Unlike *RLCheck* and QuickCheck, Zest is a *greybox* technique: it relies on program instrumentation to get code coverage from each test execution.

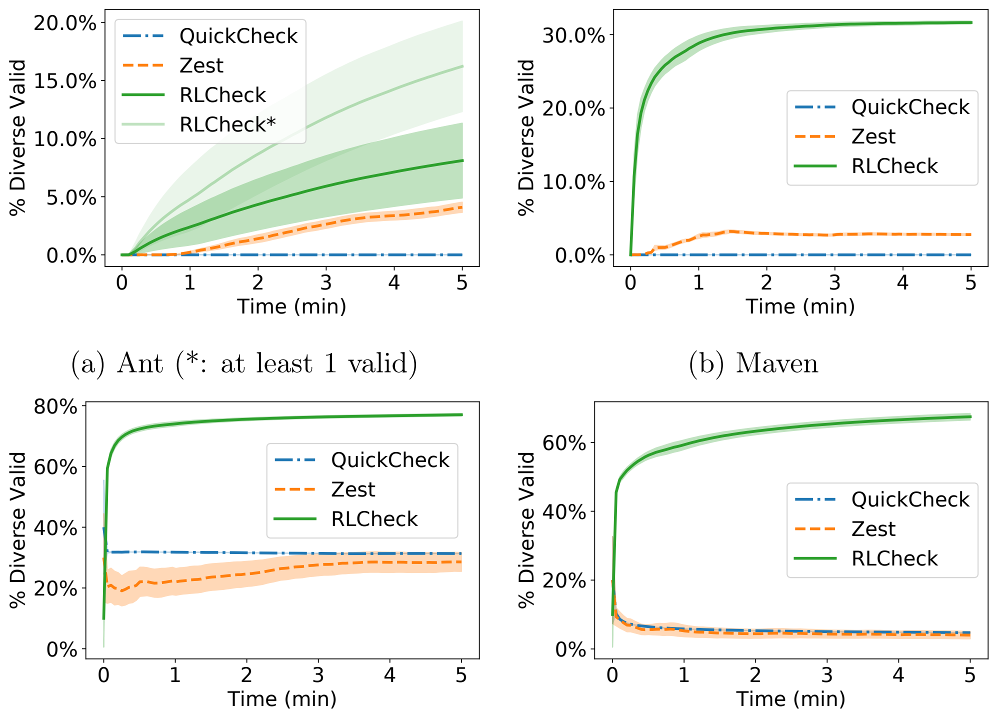
Benchmarks We compare the techniques on four Java benchmarks described in the prior chapter: Apache Ant, Apache Maven, Google Closure Compiler, and Mozilla Rhino. These benchmarks rely on two generators: Ant and Maven use an XML generator, whereas Closure and Rhino use a generator for JavaScript ASTs. The validity functions for each of these four benchmarks is distinct: Ant expects a valid `build.xml` configuration, Maven expects a valid `pom.xml` configuration, the Closure expects an ES6-compliant JavaScript program that can be optimized, and Rhino expects a JavaScript program that can be statically translated to Java bytecode. Overall, Ant has the strictest validity function and Rhino has the least strict validity function.

Design Choices In our main evaluation, we simply use identity as the characteristic function ξ. Thus, *RLCheck* simply tries to maximize the number of unique valid inputs. This allows us to run *RLCheck* at full speed without instrumentation, and generate more inputs in a fixed time budget. In Section 7.4.4 we compare this choice to a greybox version of *RLCheck*, where $\xi(x)$ takes into account the branch coverage achieved by input x.

We instantiate our reward function (Eq. 7.1) with $R_{unique} = 20$, $R_{valid} = 0$ and $R_{invalid} = -1$. This incentivizes *RLCheck* to prioritize exploration of new unique valid inputs, while penalizing strategies that lead to producing invalid inputs. Additionally, we set $\varepsilon = 0.25$ in Algorithm 8, which allows *RLCheck* to explore at random with reasonably high probability.

We first modified the base generators provided by JQF for XML and JavaScript to transform choice points with infinite domains to finite domains. These are the generators we use for evaluation of Zest and QuickCheck. We then built guide-based generators with the same choice points as these base generators. For the guide-based generators, we built the state abstraction inline, like it is built in Figure 7.3. For each benchmark, the state abstraction function is similar to that in Figure 7.3 as it maintains abstractions of the parent choices. We set $w = 5$ for the state window size.

Experiments We sought to answer our research questions in a property-based testing context, where we expect to be able to run the test generator for a short amount of time. Thus, we chose 5 minutes as a timeout. To account for variability in the results, we ran



(a) Ant (*: at least 1 valid) (b) Maven

(c) Rhino (d) Closure

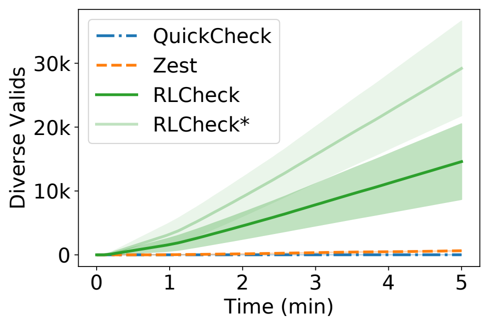
Figure 7.6: Percent of total generated inputs which are diverse valids (i.e. have different traces). Higher is better.

10 trials for each technique. The experiments in Section 7.4.1 and 7.4.2 were run on GCP Compute Engine using a single VM instance with 8vCPUs and 30 GB RAM. The experiments in Section 7.4.4 were run on a machine with 16GB RAM and an AMD Ryzen 7 1700 CPU.

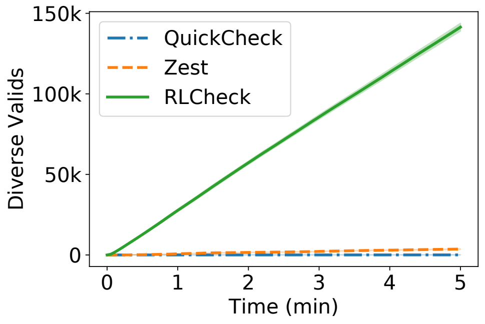
7.4.1 Generating Diverse Valid Inputs

To answer RQ1, we need to measure whether *RLCheck* generates a higher number of unique, valid inputs compared to our baselines. On these large-scale benchmarks, where the test driver does non-trivial work, simple uniqueness, at the byte or string level, is not the most relevant measure of input diversity.

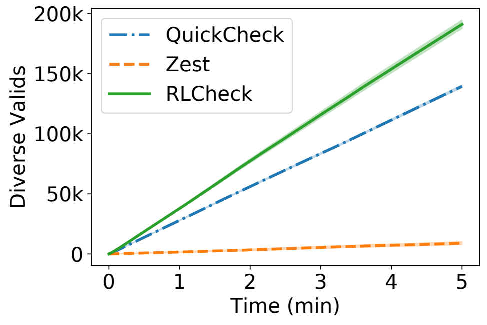
What we are interested in is inputs with diverse coverage. So, we measure inputs with different *traces*, a commonly-used metric for input coverage diversity in fuzzing literature [189] (as discussed in Section 2.1, these traces are sometimes called "paths", but this is a misnomer). The trace of an input x is a set of pairs (b, c) where b is a branch and c is the number of times that branches is executed by x, bucketed to base-2 orders of magnitude. Let $\xi(x)$ give the trace of of x. If x_1 takes the path A, B, A, then $\xi(x_1) = \{(A, 2), (B, 1)\}$. If x_2 takes the path



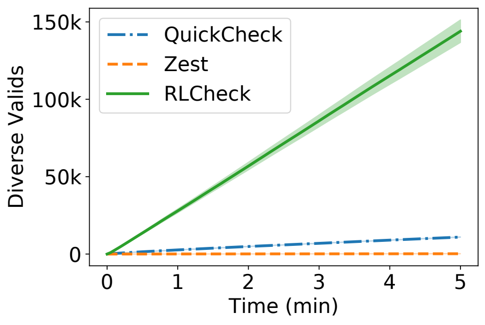
(a) Ant (*: at least 1 valid)



(b) Maven



(c) Rhino



(d) Closure

Figure 7.7: Number of diverse valid inputs (i.e. inputs with different traces) generated by each technique. Higher is better.

A, A, A, B, then A is hit the same base-2 order-of-magnitude times, so $\xi(x_2) = \{(A, 2), (B, 1)\}$. We call valid inputs with different traces *diverse valid* inputs.

Figures 7.6 and 7.7 show the results. Figure 7.6 shows, at each time, the *percentage* of all generated inputs that are diverse valid inputs. For techniques that are only able to generate a fixed number of diverse valid inputs, this percentage would steadily decrease over time. In Figures 7.6c and 7.6d, we see an abrupt decrease at the beginning of fuzzing for Zest and QuickCheck, and for Closure we see a continuing decrease in the percentage over time for these techniques. In Figures 7.6b, 7.6c, and 7.6d see that *RLCheck* quickly converges to a high percentage of diverse valid inputs being generated, and maintains this until timeout.

RLCheck also generates a large *quantity* of diverse valid inputs. Figure 7.7 shows the total number of diverse valid inputs generated by each technique: we see that *RLCheck* generates multiple order of magnitude more diverse valid inputs compared to our baselines. The exception is on Rhino (Figure 7.7), *RLCheck* only has a 1.4× increase over QuickCheck. Rhino's validity function is relatively easy to satisfy: most JavaScript ASTs are considered valid inputs for translation; therefore, speed is the main factor in generating valid inputs for this benchmark. Consequently, the blackbox techniques *RLCheck* and QuickCheck outperform the instrumentation-based Zest technique on the Rhino benchmark.

On both metrics, the increase in Ant is less pronounced, and very variable. The variation in percentage for Ant is quite wide because it is hard to get a first valid input for *RLCheck* (and QuickCheck), and in some cases *RLCheck* did not get this within the five-minute time budget. For an understanding of the effect on the results, *RLCheck** shows the results for only those runs that find at least one valid input. The mean value for *RLCheck** is much higher, but the high standard errors remain because these runs find the first valid input being at different times. For such extremely strict validity functions, *RLCheck* has difficulty finding a first valid input compared to coverage-guided techniques. This is a limitation of *RLCheck*: a good policy can only be found after some valid inputs have been discovered.

For completeness, we also ran longer experiments of 1 hour, to see if Zest or QuickCheck would catch up to *RLCheck*. In 1 hour, *RLCheck* generates between 5-15$\times$ more diverse valid inputs than Zest on all benchmarks and outperforms QuickCheck on all benchmarks. Furthermore, *RLCheck* continues to generate a higher percentage of generated diverse valid inputs after one hour. In particular, the large improvements that are seen in Figures 7.6 are all maintained at roughly the same rate except for Rhino. In the case of Rhino, Zest improves its percentage of diverse valid inputs from 40% to 67% after one hour, while *RLCheck* continues to generate 78% diverse valid inputs throughout.

RQ1: *RLCheck* quickly converges to generating a high percentage of diverse valid inputs, and on most benchmarks generates orders of magnitude more diverse valid inputs than our baselines.

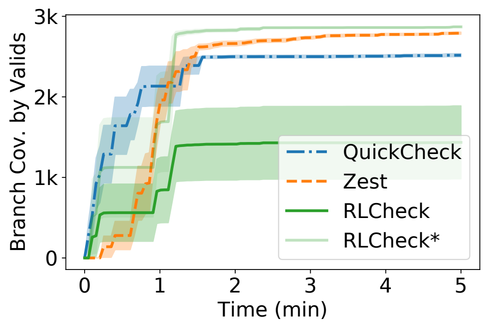
7.4.2 Covering Different Valid Behaviors

Section 7.4.1 shows that *RLCheck* generates many more diverse valid inputs than the baselines, i.e. solves the diversifying guidance problem more effectively. A natural question is whether the valid inputs generated by each method cover the same set of input behaviors (RQ2). For this, we can compare the cumulative branch coverage achieved by the valid inputs generated by each technique.

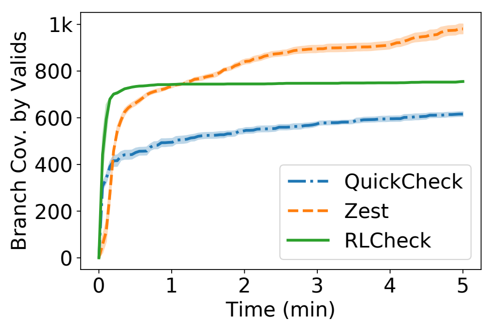
Figure 7.8 shows the coverage achieved by all valid inputs for each benchmark over time. The results are much more mixed than those in Section 7.4.1. On the Closure benchmark (Fig. 7.8d), QuickCheck and *RLCheck* achieve the same amount of branch coverage by valid inputs. On Rhino (Fig. 7.8c) QuickCheck dominates slightly. On Maven (Fig. 7.8b), *RLCheck* takes an early lead in coverage but Zest's coverage-guided algorithm surpasses it at timeout.

On Ant (Figure 7.8a), *RLCheck* appears to perform poorly, but this is mostly an artifact of *RLCheck*'s bad luck in finding a first valid input. Again, for comparison's sake, *RLCheck** shows the results for only those runs that generate valid inputs: we see that *RLCheck*'s branch coverage is slightly above Zest's on these runs.

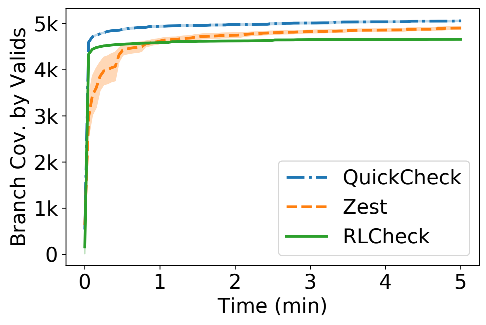
The overall clearest trend from Figure 7.8 is that *RLCheck*'s branch coverage seems to quickly peak and then flatten compared to the other techniques. This suggests that our MCC-based algorithm, while it is exploring diverse valid inputs, may still be tuned too much



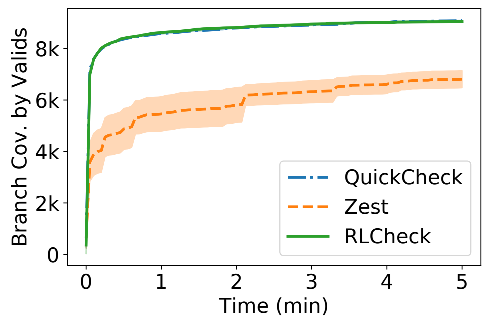
(a) Ant



(b) Maven



(c) Rhino



(d) Closure

Figure 7.8: Number of branches covered by valid inputs. Higher is better.

towards exploiting the knowledge from the first set of valid inputs it generates. We discuss in Section 7.5 some possible avenues to explore in terms of the RL algorithm.

RQ2: No method achieves the highest branch coverage on all benchmarks. *RLCheck*'s plateauing branch coverage suggests that it may be learning to generate diverse inputs with similar features rather than discovering new behavior.

7.4.3 Bug-Finding Ability

In answering RQ1, we established that *RLCheck* was able to generate orders-of-magnitude more diverse valid inputs. A natural question is whether *RLCheck* also has an increased ability to find bugs.

During our evaluation runs, the techniques found a subset of the bugs described in the previous chapter. Table 7.1 lists, for each bug that was discovered during our evaluation runs, the mean time to find (MTF) and reliability (percent of runs on which the bug was found) for each method. Bugs are deduplicated, as in Chapter 6, by exception type.

We see that on the Ant, where RLCheck found 1000× more diverse valid inputs than QuickCheck, it found bug (#1) 4× faster and 5× more often than QuickCheck. It was also

Table 7.1: Mean time to find (MTF) and Reliability (Rel.)—the percentage of runs on which the bug was found—for bugs found by each technique during our experiments. Bugs are deduplicated by benchmark and exception type. Dash "-" indicates bug was not found.

	RLCheck		**QuickCheck**		**Zest**	
Bug ID	MTF	Rel.	MTF	Rel.	MTF	Rel.
Ant, (#1)	41s	50%	178s	10%	123s	90%
Closure, (#2)	1s	100%	1.2s	100%	23s	60%
Rhino, (#3)	95s	90%	62s	70%	276s	10%
Rhino, (#4)	11s	100%	1s	100%	30s	100%
Rhino, (#5)	-	-	3s	100%	80s	100%
Rhino, (#6)	-	-	96s	20%	-	-

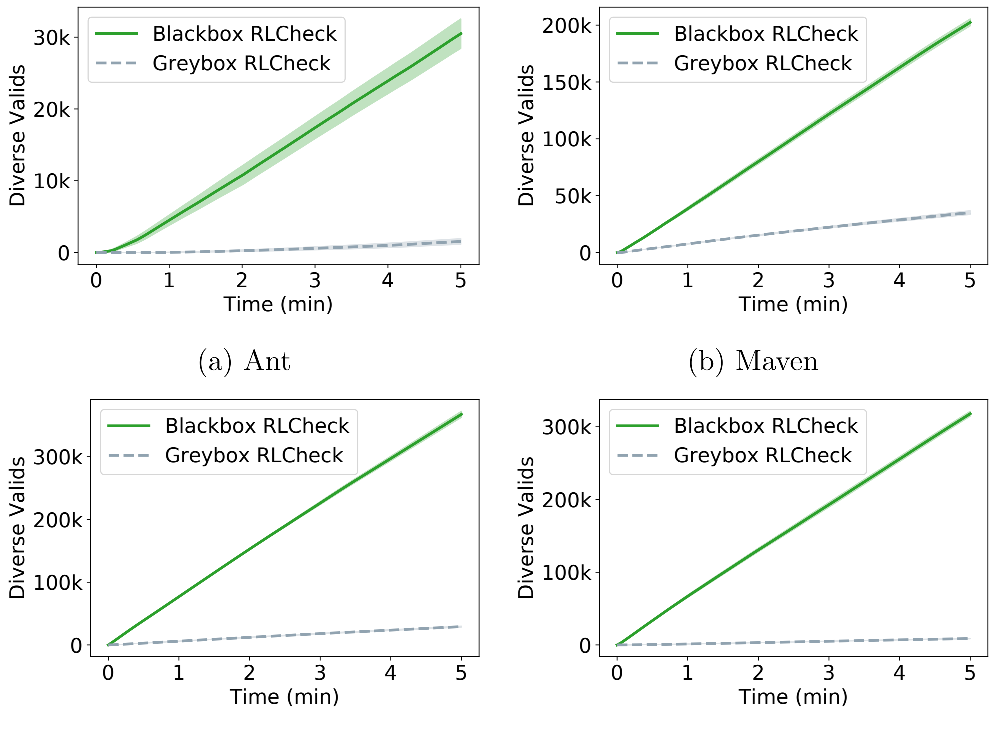
faster than Zest. On Closure, where RLCheck found 60× more diverse valid inputs than Zest, it was also 20× faster at finding fault (#2). In contrast, on Rhino, RLCheck only found 1.4× more unique valid inputs than QuickCheck. In fact, as shown in Figure 7.6c, over 30% of generator-generated inputs already satisfied the validity function. Thus, on this benchmark, the plain generator-based approach (QuickCheck) had the best fault discovery of the three methods. This benchmark is representative of situations where the generator is already fairly well-tuned for the validity function of the program under test. While the results are too sparse to be conclusive, we can make the following observation:

RQ3: Order-of-magnitude increases in input diversity seem to relate to better bug discovery; on the benchmark where *RLCheck* did not reach orders-of-magnitude increases, its ability to discover bugs was worse than the baseline.

7.4.4 Greybox Information

Given that *RLCheck* is able to attain its objective as defined by the diversifying guidance problem (Section 7.4.1)—generating large numbers of unique valid inputs—, but does not achieve the highest branch coverage over all benchmarks (Section 7.4.2), a natural question is to ask whether choosing a different ξ, one that is coverage-aware, could help increase the diversity of behaviors discovered. This is what we seek to answer in RQ4.

For this experiment, we re-ran *RLCheck* both blackbox, i.e. with $\xi_{bb} = id$, and with greybox information, using $\xi_{gb}(x) =$ "the set of all branches covered by the input x". Thus, Greybox *RLCheck* is rewarded when it discovers a valid input that covers a distinct set of branches compared to all generated inputs. Note that this does not reward the guide more for generating an input which covers a wholly-uncovered branch, compared to an input that covers a new combination of already-seen branches. Again, we ran each method for 10 trials, timing out at 5 minutes.



(a) Ant (b) Maven

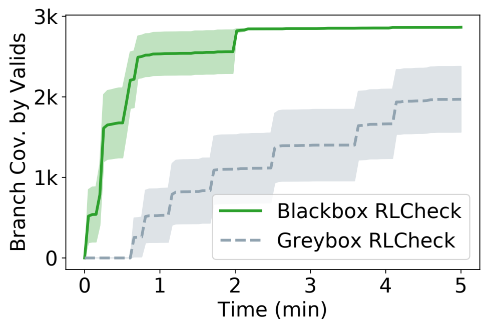
(c) Rhino (d) Closure

Figure 7.9: Number of diverse valid inputs generated by each technique. Higher is better.

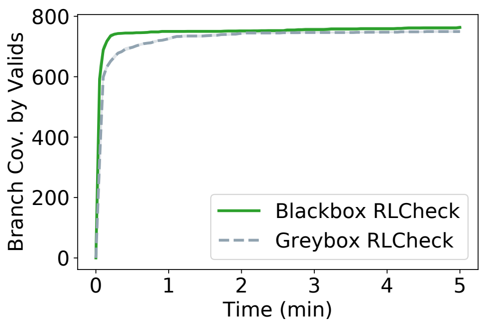
Figures 7.9 shows the number of diverse valid inputs generated the the blackbox and greybox versions of *RLCheck*, and Figure 7.10 shows the branch coverage by valid inputs for these two versions. We see universally across all benchmarks and both metrics that Blackbox *RLCheck* outperforms Greybox *RLCheck*. This suggests that the slowdown incurred by instrumentation is not worth the increased information *RLCheck* gets in the greybox setting. The difference is less striking for branch coverage than number of diverse valid inputs generated, because fewer inputs are required to get the same cumulative branch coverage.

We see much lower variation in Ant in this experiment because on all 10 runs, Blackbox *RLCheck* was able to generate at least one valid input for Ant. We chose random seeds at random in both experiments, so this is simply a quirk of experimentation.

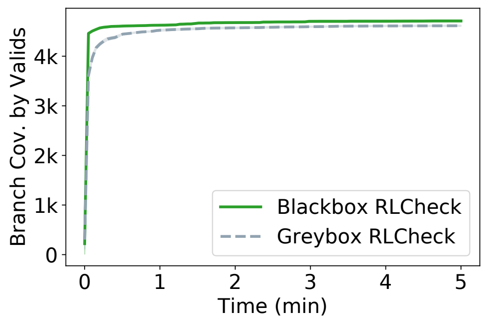
RQ4: Adding greybox feedback to the characteristic function ξ causes a large slowdown, but no huge gains in number of valid inputs or coverage achieved. Overall, *RLCheck* performs best as a black-box technique.



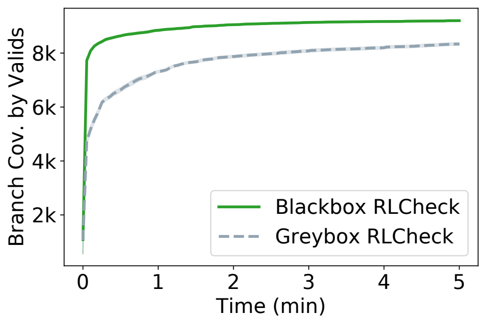
(a) Ant



(b) Maven



(c) Rhino



(d) Closure

Figure 7.10: Number of branches covered by valid inputs generated by each technique. Higher is better.

7.5 Discussion

Tabular methods such as ours do not scale well for large choice or state spaces. If S and C denote state and choice space sizes, the Monte Carlo control algorithm requires $\mathcal{O}(SC)$ space and $\mathcal{O}(C)$ time to evaluate the policy function π. This is because all the algorithmic decision-making is backed by a large Q-table with $S \times C$ entries. Because of these constraints we had to restrict our state and choice spaces. For example, in our JavaScript generator, when selecting integer values, we restricted our choice space to be in range from 0 to 10 rather than a larger range like between 0 and 100. Function approximation methods, such as replacing the Q-table with a neural network, may be necessary for dealing with larger, more complex, state and choice spaces.

In Section 7.4.1 we saw that *RLCheck* had difficulty generating a first valid input for very strict validity functions (Ant). This limitation could be overcome by allowing *RLCheck* to be *bootstrapped*, i.e. given a sequence of choices that produces a valid input at the beginning of testing. This choice sequence could be user-provided, as long as there exists a relatively short sequence of choices resulting in a valid input.

In Section 7.4.2 we observed that the branch coverage achieved by *RLCheck*-generated valid inputs tends to quickly plateau, even for benchmarks where the other methods could achieve higher branch coverage (Figs 7.8b, 7.8c). This suggests that even with a high ε, Monte Carlo Control may still be too exploitative for the diversifying guidance problem. One approach to increase exploration would be to allow the learners to "forget" old episodes so choices made early in the testing session that are not necessary to input validity do not persist throughout the session. Curiosity-based approaches, which strongly encourage exploration and avoid revisiting states [152], may also be applicable.

Overall, what we saw from this chapter is the great potential of general guided generator-based fuzzing. In particular, the casting of the problem as the diversifying guidance problem—rather than simply viewing guided generator-based fuzzing at the byte-parameter level—opens the door to more complex distribution-tuning approaches. The next chapter introduces a very different distribution-tuning approach, which brings generator-based search to the program synthesis domain.

Chapter 8

AutoPandas: Generator-Based Program Synthesis

All the previous chapters of this dissertation have focused on solving the same fundamental search problem: how to find, in a large search space of inputs, interesting inputs to a program under test. The previous chapter tackled the problem of generating more valid inputs by cleanly separating the generator, which encodes domain knowledge about the search space, from the *guide*, which controls the sampling from the generator.

This abstraction brings about the question: are there other search problems that could be solved with a smartly guided generator-based search? One particular search problem emerges as well-suited to this framework: that of synthesizing programs in a complex, real-world API.

Developers nowadays have to contend with a growing number of APIs. Many of these APIs are very useful to developers, increasing the ease of code re-use. API functions provide implementations of functionalities that are often more efficient, more correct, or produce better-looking results than what the developer can implement.

Unfortunately, it can be difficult to learn how to use an API. Many new APIs are wide, with hundreds of functions, some of which have overlapping semantics. Further, each function can have tens of arguments governing its behavior. For example, some Python APIs such as NumPy [94] use the flexible type system to define almost entirely different behavior for functions based on the type or arguments. The documentation of all of these factors is of varying quality. Further, modern APIs are frequently updated, so tutorials, blog posts, and other external resources on the API can quickly fall out of date. All these factors increase the difficulty of API use.

Oftentimes, however, when trying to use an APIs to conduct *data transformation*, novice developers know which transformation they want to perform. The popularity of online help forums such as StackOverflow has normalized the practice of creating an *input-output* (I/O) example that clearly illustrates the transformation. By I/O examples of a transformation, we mean examples where the input is the data before the transformation (e.g. the string "Garbledy Goop"), and the output is the desired transformed data (e.g. "Goop, G.").

With an I/O example, observe that the problem of synthesizing a program in the desired

	Date	Category	Location	Expense	Balance
0	2018-02-18	Social	Terrace	98.34	9971.66
1	2018-02-18	Lunch	Pox	245.63	9726.03
2	2018-02-24	Social	Gate 320	121.89	9604.14
3	2018-02-24	Lunch	Pox	248	9356.04

(a) An example input DataFrame.

	Lunch	Social
2018-02-18	245.63	98.34
2018-02-24	248	121.89

(b) Desired output.

Figure 8.1: A DataFrame input-output example.

API [70, 69, 68] becomes very similar to the testing search problem explored in the prior chapters of this dissertation. Instead of generating *inputs* to the program under test until a generated input *crashes the program*, the search problem is to generate *programs* and run them on the user-provided input until the output of a generated program *matches the user-provided output*.

This chapter explores the efficacy of a generator-based program synthesis technique for the Python `pandas` API. The program candidate generator encodes expert domain knowledge about the API to—as much as possible—synthesize programs in the API which are *valid*. For example, when producing arguments to a function, the generator should almost never produce argument combinations which cause the function to immediately error out. Given knowledge of the API, writing such a candidate program generator is a straightforward—although perhaps tedious—effort. This means such a generator can be written by any developer who knows the API, regardless of their familiarity with program synthesis techniques.

However, if the generator takes a long time to generate a p such that $p(\mathit{input}) = \mathit{output}$, this simple generator-based search—which just calls the generator until it yields such a p—becomes impractical. Unlike the testing scenario, where the search process can be run for hours, in order for a program synthesis tool to be practical, the search process can be run for minutes at most. So, instead of taking choices in the generator totally randomly, the choices in the generator will, much like in the previous chapter, be taken with respect to a smart *guide*. Ideally, this guide ensures that a generated program which matches the input-output example appears not too late in the search process.

Before describing the details of this guide, and the process of generator-based program synthesis, we will introduce a small example that motivates its necessity.

8.1 Motivation

Suppose a developer or data scientist, new to the Python library `pandas`, needs to use `pandas` in order to pre-process data for a statistical or ML pipeline. For example, suppose they need to transform an expense table, given in Figure 8.1a, into the dataframe in Figure 8.1b.

Suppose first that while the novice does not know `pandas`, they know some other data transformation tools. In particular, the novice knows that in Microsoft Excel, they can

```
1 def find_pivot_args(input_df: pandas.DataFrame,
2                     output_df: pandas.DataFrame):
3     while True:
4         cur_kwargs = generate_pivot_args(input_df, output_df)
5         cur_out = pandas.DataFrame.pivot(input_df, **cur_kwargs)
6         if cur_out == output_df:
7             return cur_kwargs
```

Figure 8.2: A procedure to find the arguments to the `pandas` function `pivot` that turn `input_df` into `output_df`.

perform the transformation in Figure 8.1 using the "pivot table" functionality. The novice also notices that `pandas` has a `pivot` function for dataframes, and thinks they can use it. But, for complex APIs like `pandas`, there are many arguments for each function, requiring substantial effort to master even one function. Resources explaining all the complexity of the API can overwhelm a novice.

Hence, novices often resort to asking experts for help on which arguments to use. Unfortunately, this is not a perfect solution. First, if no expert is around to answer the question, a novice can get stuck on the problem for a long time. Also, the expert finds themselves constantly answering a very similar question—what `pivot` arguments should be used to get `output` from `input`? Answering this question over and over again is not scalable. So nowadays, if a basic `pivot` or `merge` question is asked on `pandas` StackOverflow, it is not answered, and simply marked as a duplicate of a master answer diving into the details of these functions. As of May 2021, these master answers had 927 and 708 duplicates, respectively. [1]

Instead of simply redirecting novices to documentation, the API expert can write a generator that outputs valid argument combinations for `pivot` on the dataframe `df`, say `generate_pivot_args(df)` (Figure 8.3). The novice can then use `generate_pivot_args(df)` to enumerate the argument combinations, and save the one that works for their input-output example. Figure 8.2 shows pseudo-code to find the correct arguments for `pivot`, given the generator `generate_pivot_args(df)`. The code simply calls `generate_pivot_args(df)` (Line 4) until it returns an argument combination `kwargs` which satisfies the I/O example, i.e., such that `pivot(input_df, **kwargs) == output_df` (Line 6). This is essentially generator-based fuzzing, but instead of finding inputs that cause the program under test to crash, the search is finding argument combinations that satisfy the input-output example.

To make sure that all the argument combinations returned by `generate_pivot_args(df)` are valid, the expert has encoded in the generator basic constraints on the `pivot` arguments:

1. `arg_col` should be selected from the list of column names of `df`, `df.columns`.

[1]The current number can be determined with the query at `https://data.stackexchange.com/stackoverflow/query/edit/1024223`.

```python
@generator
def generate_pivot_args(input_df: DataFrame, output_df: DataFrame):
  context = (input_df, output_df)
  arg_col = Select(df.columns, context, id=1)
  arg_idx = Select({None} | df.columns - {arg_col}, context, id=2)
  if isinstance(df.index, pandas.MultiIndex) and arg_idx is None:
    arg_val = None
  else:
    arg_val = Select(df.columns - {arg_col, arg_idx}, context, id=3)

  return {'columns': arg_col, 'index': arg_idx, 'values': arg_val}
```

Figure 8.3: A generator of all valid arguments to the `pivot` function from the `pandas` API. `Select(D,c,i)` returns a single element from the domain `D`.

2. `arg_idx` is either `None`, or selected from the list of column names of `df`, except from the column name used in `arg_col` (`df.columns-{arg_col}`).
3. Finally, the `arg_val` argument should either be (1) selected from the list of column names except for the ones used in `arg_col` and `arg_idx`, or (2) `None`, in the case where `arg_idx` is `None` and `df` has a multi-level index.

These constraints are universal for the `pivot` function, and an expert can straightforwardly derive them from the documentation.

Figure 8.3 shows the implementation of `generate_pivot_args(df)`. The calls to `Select` return a single element from their domains `D`. Assume first that `Select(D,c,i)` is the naïve choice operator, which returns a random element from `D`. Section 8.2 formalizes the smartly guided `Select` and explains the arguments `c` and `i`.

Unfortunately, if there are many argument combinations, the basic search in Figure 8.2 may take some time to terminate. The problem gets worse if the exact function to use is not known. If the novice does not know whether to use `pivot`, `pivot_table`, `unstack`, etc., they would have to go through the argument combinations for each of these functions. If `generate_pivot_args` returns arguments in a totally random order, the correct argument combination is unlikely to show up early enough to ensure a practical synthesis time. The problem is exacerbated if sequences of multiple functions are required to perform the transformation, as the total number of possible argument combinations grows exponentially.

To make `generate_pivot_args` output the correct argument combination more quickly, the API expert *could* replace the calls to `Select(D,c,i)` with a particular enumeration order through codeD. The enumeration order would be based on some additional *heuristics*, e.g.:

1. The values in the column from `input_df` that is used as `arg_col` end up being column names in the `output_df`. Therefore, the generator should look at the output's column

names, and first try to use as `arg_col` any column from the input that shares values with the output's column names.

2. The values in the column from `input_df` that is used as the `arg_val` argument end up in the main data of the table. Hence, the generator should look at the output's data, and first try to use as `arg_val` any column whose values are the same as output's data cells. However the values argument also accepts `None` as a valid argument, in which case all the remaining column values are used as the main data of the output. Therefore the generator should take this into account as well.

3. ... *(more heuristics omitted)*

Designing such heuristics is error-prone. They are not guaranteed to be effective, especially if the I/O example provided by the user cannot actually be solved with a single call to `pivot`. Further, it is much more tedious for the expert to write a generator that uses these heuristics than it is to write a generator that encodes the basic validity constraints, like that in Figure 8.3. This is even harder than the problem described in Section 7.1 of Chapter 7: instead of writing heuristics to generate valid inputs, these heuristics must quickly generate a program satisfying the input-output example. It is almost as if the developer of a generator for testing needs to write the heuristics which will quickly result in a bug-finding input.

Instead of requiring a human to write these heuristics, we propose to have the generator follow a smart *guide*. For each **`Select`** statement, the guide first derives from the context `c` a probability distribution p over `D`. Then, it returns elements $d \in$ `D` in descending order of their probability $p(d)$. The distribution model is represented by a neural network, learned from a training set of inputs, programs, and their outputs. Over a validation set of (`input_df`, `output_df`) pairs where `output_df = pivot(input_df, **kwargs)` for some `kwargs`, our smart backend has 99% top-1 accuracy in retrieving the correct `kwargs`.

In order to synthesize full `pandas` programs which satisfy an `(input_df, output_df)` example, we take a similar approach. We implement a *program candidate* generator which outputs straight-line `pandas` programs that run without error on `input_df`. This generator follows a similar smart guide as described above. This generator-based synthesis engine, AutoPandas, supports 119 `pandas` functions and can form programs with multiple function calls. Given the I/O example in Figure 8.1, AutoPandas finds the correct program:

```
output_df = input_df.pivot(index='Date', columns='Category', values='Expense')
```

after checking only *one* program candidate.

In the next section, we formalize this generator-based program synthesis technique, as well as the smart guides which yield these results.

8.2 The AutoPandas Technique

Figure 8.4 shows the core AutoPandas technique of generator-based synthesis. The engine consists of two components — (1) a program candidate generator and (2) a checker that checks

```
def synthesize(input, output, max_len):
  while (True):
    candidate = generate_pandas_program(input, output, max_len)
    if candidate(input) == output:
          return candidate
```

Figure 8.4: Generator-Based Enumerative Synthesis Engine.

if the candidate program produces the correct output. The checker is rather straightforward to implement: we simply execute the program and test the exact match of its output to the target output. The bulk of the work is done by the program candidate generator.

A program candidate generator is a generator that, given an input-output example, generates program candidates. Figure 8.5 shows an excerpt of our program candidate generator for `pandas` programs. This generator produces straight-line programs, each of which is a sequence of up to `max_len` `pandas` function calls. The program at the end of Section 8.1 is one such candidate, consisting of a sequence of length 1.

The generator in Figure 8.5 generates candidate programs as follows. First, it picks a sequence of functions from a list of supported functions (Lines 3-4). Then, for each function in the sequence, the generator selects the arguments (Lines 10-32), and computes the result by running the function with the arguments and stores it as an *intermediate* (e.g. Line 33). Intermediates are the outputs produced by previous functions in the sequence. These are essential to allow the generator to generate meaningful multi-function programs, where a function can operate on the output of a previously applied function. Argument generation is done on a case-by-case basis depending on the given function. For example, for the function `pivot` (Lines 10-21), the generator follows the argument generation logic of Figure 8.3, applies the function with the selected arguments to a selected input or intermediate `df`, and stores the output as an intermediate. The program candidate generator can handle `pandas` functions that operate on multiple dataframes, e.g. `merge` on Lines 23-29, by selecting each dataframe from the set of input and intermediates (Lines 24-25).

Unlike the generators shown in the previous chapter, the generator in Figure 8.5 has choice point operators other than **`Select`**, which simply outputs a choice from its domain `D`. The three additional operators we support are: (1) **`Subset`**, (2) **`OrderedSubset`** and (3) **`Sequence`**. An informal description of their behavior is provided in Table 8.1. They can be understood as essentially syntactic sugar around **`Select`**, e.g. **`Subset`**`(D,c,i)` is simply a **`Select`** over the possible subsets of `D`.

More concretely, each operator Op is of the form $Op(\mathcal{D}, \mathcal{C}, id)$ where $\mathcal{D}$ is the domain passed to the operator; $\mathcal{C}$ is the context passed to the operator to control its behavior; and id is the unique static ID of the operator. The static ID of Op simply identifies each call to an operator uniquely based on its static program location. It is provided explicitly in Figure 8.3 for clarity but may be inserted automatically via a static instrumentation pass of

```
@generator
def generate_pandas_program(input, output, max_len):
  functions = [pivot, drop, merge, ...]
  function_sequence = Sequence(max_len)(functions,
                                        context=[input,output],id=1)

  intermediates = []
  for function in function_sequence:
    c = [input, *intermediates, output]
    if function == pivot:
      df = Select(input + intermediates, context=c, id=2)
      arg_col = Select(df.columns, context=[df, output], id=3)
      arg_idx = Select(df.columns - {arg_col}, context=[df, output],
                       id=4)

      if isinstance(df.index, pandas.MultiIndex) and arg_idx is None:
        arg_val = None
      else:
        arg_val = Select(df.columns - {arg_col, arg_idx},
                         context=[df, output], id=5)
        args = (df, arg_col, arg_idx, arg_val)

    elif function == merge:
      df1 = Select(input + intermediates, context=c, id=6)
      df2 = Select(input + intermediates, context=c, id=7)
      common_cols = set(df1.columns) & set(df2.columns)
      arg_on = OrderedSubset(common_cols, context=[df1, df2, output],
                             id=8)
      args = (df1, df2, arg_on)
                                   ⋮
    # Omitted code: case for each function
                                   ⋮
    intermediates.append(function.run(*args))

  return function_sequence
```

Figure 8.5: A Simplified Program Candidate Generator for pandas Programs.

Table 8.1: List of Available Choice Operators

Operator	**Description**
`Select(domain)`	Returns a single item from `domain`
`Subset(domain)`	Returns an unordered subset, without replacement, of the items in `domain`
`OrderedSubset(domain)`	Returns an ordered subset, without replacement, of the items in `domain`
`Sequence(len)(domain)`	Returns an ordered sequence, with replacement, of the items in `domain` with a maximum length of len

the generator code. The actual choice domain of the operator, $W(Op, \mathcal{D})$, depends on the operator, but is consistent with the description in Table 8.1:

$$W(Op, \mathcal{D}) = \begin{cases} \mathcal{D} \text{ if } Op = \mathit{Select} \\ \mathit{PowerSet}(\mathcal{D}) \text{ if } Op = \mathit{Subset} \\ \cup\{\text{Perms}(x) \mid x \in \mathit{PowerSet}(\mathcal{D})\} \text{ if } Op = \mathit{OrderedSubset} \\ \{(a_1, \cdots, a_k) \mid k \leq l, a_i \in \mathcal{D}\} \text{ if } Op = \mathit{Sequence}(l) \end{cases}$$

With these concepts in mind, we can again define the notion of a generator following a guide. This time, let a choice point $p = (Op, \mathcal{D}, \mathcal{C}, id)$ encode all the information about an operator call, and let P_G be the set of choice points for a generator G, and C the set of all possible choices.

Definition 23 (Following a Guide)**.** We say that a generator G *follows* a guide $\gamma : P_G \to C^* \times \mathbb{N} \to C$ if: during its t^{th} execution, given a sequence of past choices $\sigma = c_1, c_2, \ldots, c_k$, and the current choice point $p = (Op, \mathcal{D}, \mathcal{C}, id)$, the generator G makes the choice $\gamma(p)(\sigma, t) \in W(Op, \mathcal{D})$.

The parameter σ is necessary to separate different calls to the same operator in one generator invocation, and the parameter t distinguishes different invocations of the generator. We consider three different guides in this chapter:

- **Randomized.** The simplest case is for the generator to follow a *randomized* guide, resulting in the production of random programs. This can be achieved by simply having the choice operators return a random element from their domains. Essentially, for the choice point $p = (Op, \mathcal{D}, \mathcal{C}, id)$, $\gamma(p)(\sigma, t)$ simply returns a random element of $W(Op, \mathcal{D})$.
- **Exhaustive (Depth-First).** Another option is to have the generator follow an *exhaustive* guide, which systematically explores all possible combinations of choices. In this case, the guide uses the (σ, t) arguments to $\gamma(p)(\sigma, t)$ to ensure that the value chosen from $W(Op, \mathcal{D})$ results in a depth-first exploration of the space of all possible programs. For example, it will force the generator in Figure 8.3 to first explore all

possible values of the `Select` call at Line 9 before moving on to the next possible value for the `Select` call at Line 5.

- **Smart.** Note that neither the randomized nor exhaustive guides utilize the context $\mathcal{C}$ in making choices. The significance of this is the behavior of each choice operator is independent of the `(input_df, output_df)` passed to the generator. This is not suitable for tasks such as the one presented in Section 8.1, where the goal is to quickly find an argument combination to the `pivot` function such that when it is called on `input_df`, it produces the target output `output_df`. In this case, we want a *smart* guide which biases the generator choices towards the values that have a higher probability of creating a program satisfying `(input_df, output_df)`.

  A *smart* guide is essentially an exhaustive depth first generator. But, it forces a smart order of enumeration *dependent on the context*. For each operator $Op(\mathcal{D}, \mathcal{C}, id)$, it calls a function $Rank_{(Op,id)}(\mathcal{D}, \mathcal{C})$ which returns the values $W(Op, \mathcal{D})$ in an optimal exploration order given the context $\mathcal{C}$. Note that the *Rank* function is sub-scripted by (Op, id), implying that every operator call can have a separate ranking function.

  In the generator in Figure 8.3, the context passed at every operator call is the input and output dataframe. Therefore given suitable ranking functions $Rank_{(Select,1)}$, $Rank_{(Select,2)}$ and $Rank_{(Select,3)}$, the generator can be biased toward producing an argument combination that, when passed to the `pivot` function along with the input dataframe `input_df`, is likely to result in `output_df`.

The full AutoPandas technique thus consists of the enumerate-and-check loop from Figure 8.4, with the generator `generate_pandas_program` following a *smart*, neural-backed guide. The OOPSLA'19 paper on AutoPandas gives an alternate formalization of these guides, and their formal semantics [38].

8.3 Implementation

In AutoPandas, we use neural networks to define the *Rank* functions used by the smart guide to control the choices in our program candidate generator. We design a neural network model for each kind of operator (see Table 8.1). The first time an operator Op is called with a particular domain $\mathcal{D}$ and context $\mathcal{C}$, a query is constructed using $\mathcal{D}$ and $\mathcal{C}$. This query is passed to the neural network model, which returns a probability distribution over $W(Op, \mathcal{D})$, the domain of choices for the operator. The *Rank* function then uses this distribution to reorder the elements in $W(Op, \mathcal{D})$ in the decreasing order of probabilities. The smart guide uses this ranking to force the operator to return values in an order conditioned on the context. We now concretely define the query, its encoding, and the neural network architectures for each operator.

8.3.1 Query Encoding

The query $\mathcal{Q}$ to each neural network model, regardless of the operator, is of the form $\mathcal{Q} = (\mathcal{D}, \mathcal{C})$ where $\mathcal{D}$ and $\mathcal{C}$ are the domain and context passed to the operator.

Encoding this query into a neural-network suitable format poses several challenges. Recall that the context and the domain passed to operators in the `pandas` program candidate generator (Figure 8.5) contain complex structures such as dataframes. Dataframes are 2-D structures which can contain arbitrary Python objects as primitive elements. Even just considering strings or numbers, the set of possible primitive elements is infinite. This renders all common value-to-value map-based encoding techniques popular in machine learning, such as one-hot encoding, inapplicable. At the same time, the encoding needs to retain enough information about the context to generalize to unseen queries which may occur when the synthesis engine is deployed in practice. Therefore, simply abstracting away the exact values is not viable. In summary, a suitable encoding needs to (1) abstract away only irrelevant information and (2) be suitably structured for neural processing. To this end, we designed a graph-based encoding that possesses all these desirable properties.

Graph-Based Encoding. We encode the domain $\mathcal{D}$ and the context $\mathcal{C}$ as a graph, consisting of nodes, edges between pairs of nodes, and labels on nodes and edges. The overall rationale is that it is not the concrete values, but rather the *relationships* amongst values, that really encode the transformation at hand. That is, relationship edges should be sufficient for a neural network to learn from. For example, the essence of transformation represented by Figure 8.1 is that the values of the column 'Category' now become the columns of the pivoted dataframe, with the 'Date' column as an index, and the 'Expense' as values. The concrete names are immaterial.

Recall that the domain and context are essentially collections of elements. Therefore, we first describe how to encode each such element e individually as a graph G_e. Later we describe the procedure to combine these graphs into a single graph $G_\mathcal{Q}$ representing the graph-encoding of the full query $\mathcal{Q}$. Figure 8.6 shows the graph-encoding of the query generated as a result of the **`Select`** call at Line 4 in Figure 8.3 and will be used as a running example.

Encoding Primitives. If the element e is a primitive value (strings, integers, float, lambda, NaN etc.), its graph encoding G_e contains a single node representing e. This node is assigned a label based on the data-type of the element as well as the *source* of the element. The source of an element indicates whether it is part of the domain, or of one of the input-outputs in the I/O example, or of one of the intermediates, or none of these.

Encoding DataFrames. If the element e is a dataframe, each cell element in the dataframe is encoded as a node in the graph G_e. The label of the node includes the type of the element (string, number, float, lambda, NaN, etc.). The label also includes the source of the dataframe, i.e. whether the dataframe is part of the domain, input, output, intermediate, or none of these. We also add nodes to G_e that represent the schema of the dataframe, by creating a node for each row index and column name of the dataframe. Finally, we add a *representor* node to G_e that represents the whole of the dataframe. The label of this node contains the type "dataframe" as well as the source of the parent dataframe. Note that this additional representor node is not created when encoding primitive elements. The node representing the primitive element itself acts as its representor node.

The graph encoding of a dataframe also contains three kinds of edges to retain the structure of the dataframe. The first kind is adjacency edges. These are added between each pair of cell nodes, column name nodes or row index nodes that are adjacent to each other in the dataframe. We only add adjacency edges in the four cardinal directions. The second kind is indexing edges, which are added between each column name node (respectively, row index node) and all the cell nodes that belong to that column (respectively, row). Finally, the third kind of edge is a representation edge, between the representor node to all the other nodes corresponding to the contents of the dataframe.

Encoding the Query $\mathcal{Q}$. Finally, to encode $\mathcal{Q} = (\mathcal{D}, \mathcal{C})$, we construct G_e for each element in $\mathcal{D}$ and $\mathcal{C}$ as described above, and create a graph G containing these G_es as sub-graphs. Additionally, to capture relationships amongst these elements, we add a fourth kind of edge—*equality* edges between nodes originating in different G_es such that the elements they represent are equal. Formally, we add an equality edge between nodes n_1 and n_2 if $n_1 \in G_{e_i} \wedge n_2 \in G_{e_j} \wedge i \neq j \wedge V(n_1) = V(n_2)$ where V is a function that given n, retrieves the value encoded as n. For representor nodes, V returns the whole element it represents. For example, for a dataframe, V would return the dataframe itself for the representor node.

Equality edges are key to capturing relationships between the inputs and the output in the I/O example, as well as the domain $\mathcal{D}$ and the I/O example. The neural network then learns to extract these relationships and uses them to infer the required *Rank* probability distribution over $W(Op, \mathcal{D})$.

8.3.2 Operator-Specific Graph Neural Network Models

Given the graph-based encoding $G_{\mathcal{Q}}$ of a query $\mathcal{Q}$, we feed it to a graph neural network model. Each operator has a different model. These models are based on the gated graph neural network, introduced by Li et al.[118]. We base our model on the implementation by Microsoft [132, 27]. We first describe the common component of all the neural network models. Then, we provide an individual description for the neural network model corresponding to each operator listed in Table 8.1.

The input to all our network models is a undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{X})$. $\mathcal{V}$ and $\mathcal{X}$ characterize the nodes, where $\mathcal{V}$ is the set of nodes and $\mathcal{X}$ is the embedding $\mathcal{X} : \mathcal{V} \rightarrow \mathbb{R}^D$. Effectively, $\mathcal{X}$ maps each node to a one-hot encoding of its label of size D, where D is a hyper-parameter. $\mathcal{E}$ contains the edges, where each edge $e \in \mathcal{E}$ is a 3-tuple (v_s, v_t, t_e).

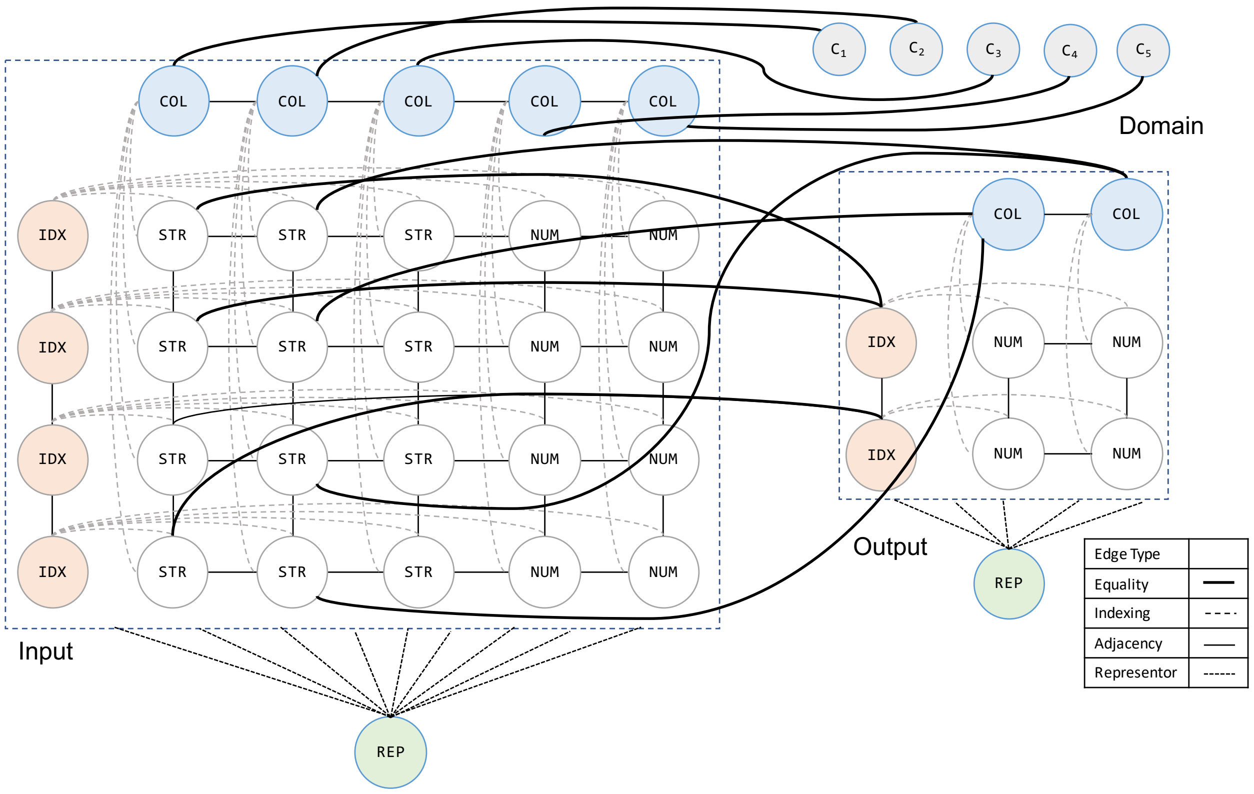
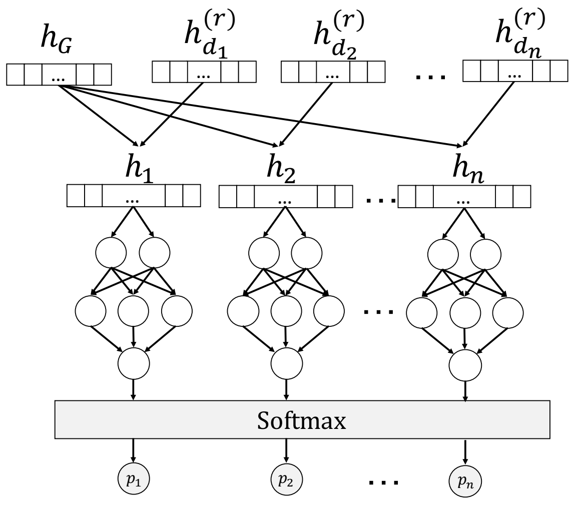


Figure 8.6: Graph encoding of the query passed to the first **Select** call in Figure 8.3, on the I/O example from Figure 8.1.

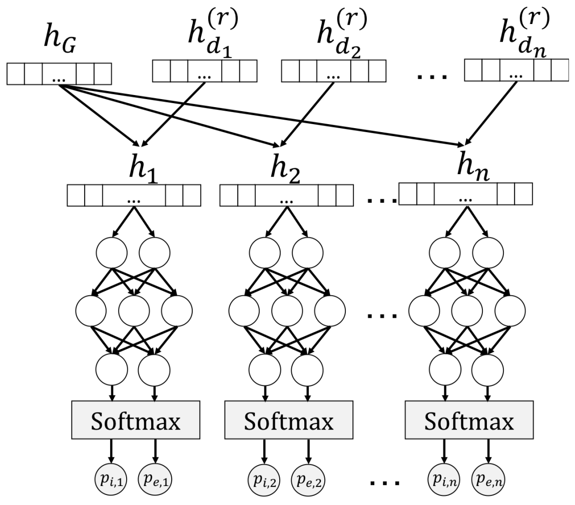
The source and target nodes are v_s and v_t, respectively. The type t_e of the edge is one of $\Gamma_e \equiv \{adjacency, indexing, representor, equality\}$ and is also one-hot encoded.

Each node v is assigned a state vector $h_v \in \mathbb{R}^D$. We initialize the vector to the node embedding $h_v^{(0)} = \mathcal{X}(v)$. The network then propagates information via r rounds of *message passing*. During round k $(0 \leq k < r)$, messages are sent across edges. In particular, for each edge $(v_s,\, v_t,\, t_e)$, v_s sends the message $m_{v_s \to v_t} = f_k(h_{v_s}^{(k)}, t_e)$ to v_t. Our $f_k : \mathbb{R}^{D+|\Gamma_e|} \to \mathbb{R}^D$ is a single linear layer. These are parameterized by a weight matrix and a bias vector, which are learnable parameters. Each node v aggregates its incoming messages into $m_v = g(\{m_{v_s \to v} \mid (v_s, v, t_e) \in \mathcal{E}\})$ using the aggregator g. In our case, we take g to be the element-wise mean of the incoming messages. The new node state vector $h_v^{(k+1)}$ for the next round is then computed as $h_v^{(k+1)} = GRU(m_v, h_v^{(k)})$ where GRU is the gated recurrent unit [57] with start state as $h_v^{(k)}$ and input m_v. We use $r = 3$ rounds of message passing, as we noticed experimentally that further increasing the number of message passing rounds did not increase validation accuracy.

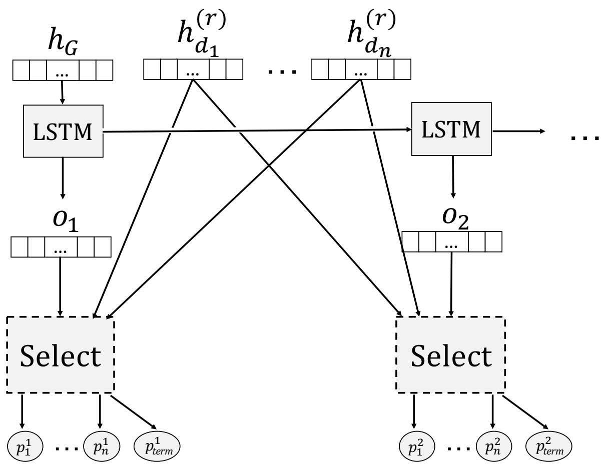
After message passing is completed, we are left with updated state vectors $h_v^{(r)}$ for each node v. Now depending on the operator, these node vectors are further processed in different



(a) Illustration of the **Select** model.



(b) Illustration of the **Subset** model.



(c) Illustration of the **OrderedSubset**/**Sequence** model. The box label "Select" expands to (a).

Figure 8.7: Operator-specific neural network architectures.

ways as described below to obtain the corresponding probability distributions over space of values defined by the operator $(W(Op, \mathcal{D}))$. A graphical visualization is provided in Figure 8.7

Select *:* We perform element-wise sum-pooling of the node state vectors $h_v^{(r)}$ into a graph state vector h_G. We now concatenate h_G with the node state vectors $h_{d_i}^{(r)}$ of the representor nodes d_i for each element in the domain $\mathcal{D}$ in the query $\mathcal{Q}$, to obtain vectors $h_i = h_G \circ h_{d_i}^{(r)}$. We pass the h_is though a multi-layer perceptron with one hidden layer and a one-dimensional output layer, and apply softmax over the output values for all the elements to produce a probability distribution over the domain elements $(p_1, \cdots, p_n)$. For inference, this distribution is returned as the result, while during training we compute cross-entropy loss w.r.t this distribution and the correct distribution where $p_i = 1$ for the correct choice i and $\forall j \neq i, p_j = 0$. Figure 8.7a shows an illustration of the model.

Subset *:* As in `Select`, we perform element-wise sum-pooling of the node state vectors and concatenate it with the state vectors of representor nodes to obtain the vectors $h_i = h_G \circ h_{d_i}^{(r)}$ for each element in the domain. However, we now pass the h_is though a multi-layer perceptron with one hidden layer and apply softmax activation on the output layer to obtain a distribution (p_{i_k}, p_{e_k}) over two label classes "include" and "exclude" for each of the domain element d_k individually. Recall that the space of possible outputs for the `Subset` operator is the power-set of the domain $\mathcal{D}$. The probability of these labels corresponds to the probability with which an element is included and excluded from the output set respectively. To compute the probability distribution, the probability of each possible output set is computed as simply the product of the "include" probabilities for the elements included in the set and the "exclude" probabilities for the elements excluded from the set. Again, this distribution is returned as the result during inference, while during training, loss is computed w.r.t this distribution and the correct individual distribution of the elements where $p_{i_k} = 1 \wedge p_{e_k} = 0$ if element d_k is present in the correct output, else $p_{i_k} = 0 \wedge p_{e_k} = 1$. Figure 8.7b illustrates the model.

OrderedSubset *and* ***Sequence*** *:* We perform element-wise sum-pooling of the node state vectors $h_v^{(r)}$ into a graph state vector h_G. We then pass h_G to an LSTM that is unrolled for $T + 1$ time-steps, where $T = |\mathcal{D}|$ for `OrderedSubset` and $T = l$ for `Sequence(l)` where `l` the max-length parameter passed to `Sequence`. The extra time-step is to accommodate a terminal token which we describe later. For each time-step t, the output o_t is concatenated with the node state vectors $h_{d_i}^{(r)}$ of the representor nodes d_is for each element in the domain passed to the operator to obtain vectors $h_i^t = o_t \circ h_{d_i}^{(r)}$. At time-step t, in a similar fashion as `Select`, a probability distribution is then computed over the domain elements plus an arbitrary terminal token $term$. The terminal token is used to indicate the end of a sequence/set. Now, to compute the probability distribution, the probability of each set or sequence $(a_0, \cdots, a_k)$ where $(k \leq T)$ is simply the product of probabilities of a_i at time-step i and the probability of the terminal token $term$ at time-step $k + 1$. As before, this distribution is directly returned during inference, while during training, loss is aggregated over individual time-steps; the loss for a time-step is computed as described in `Select`. Figure 8.7c illustrates the model.

All the network models are trained with the ADAM optimizer [105] using cross-entropy loss.

8.3.3 Training Neural-Backed Generators for Pandas

A Neural-Backed Generator consists of operators backed by *Rank* functions that are used by the smart guide to direct the order of exploration. We implement these *Rank* functions using neural networks, as described in Section 8.3.2. Training each of these networks for each call to an operator with static ID *id* requires training data consisting of tuples of the form $\mathcal{T}_{id} = (\mathcal{C}, \mathcal{D}, c)$ where c is the correct choice to be made by the operator call with static id *id*. Put another way, the neural network behind the operator call at location *id* is trained to predict the choice c with the highest probability given the context $\mathcal{C}$ and domain $\mathcal{D}$.

Unfortunately, such training data is not available externally as it is highly specific to our generators. Accordingly, we synthesize our training data automatically, i.e., synthesize a random tuple containing a context $\mathcal{C}$, domain $\mathcal{D}$, and the target choice c. This is a highly non-trivial problem, as there are two strong constraints that need to be imposed on $\mathcal{C}$, $\mathcal{D}$, and c for this tuple to be a useful training data-point. First, the random context, domain, and choice should be *valid*. That is, there should exist *an execution of the generator* for some input such that the operator call in question receives the random context and domain as arguments, and makes the same choice. Second, this tuple of context, domain, and choice should be *meaningful*, i.e., the choice should lead to progress on the task contained in the context. In our synthesis setting, this translates to the property that the generator makes a step towards producing a program that actually produces the output from the input as passed in the context. We rely on two key insights to solve these problems for our pandas program candidate generator.

Suppose we have tuples of the form $(\mathcal{I}, \mathcal{O}, \mathcal{P}, K)$ where $\mathcal{P}$ is a pandas program such that $\mathcal{P}(\mathcal{I}) = \mathcal{O}$ i.e. it produces $\mathcal{O}$ when executed on inputs $\mathcal{I}$. Also, K is the sequence of choices made by the operators in the generator such that the generator produces the program $\mathcal{P}$ when it is fed $\mathcal{I}$ and $\mathcal{O}$ as inputs. Then, it is straight-forward to extract training data tuples $(\mathcal{C}, \mathcal{D}, c)$ for each operator call by simply running the generator on $\mathcal{I}$ and $\mathcal{O}$ and recording the concrete context $\mathcal{C}$ and domain $\mathcal{D}$ passed to the operator, and forcing the operator to make the choice c. These tuples are *meaningful* by construction, as the operators make choices that lead to the generation of the program $\mathcal{P}$ which turns $\mathcal{I}$ into $\mathcal{O}$.

The second insight is that we can obtain these $(\mathcal{I}, \mathcal{O}, \mathcal{P}, K)$ tuples by using the generator itself. We generate random inputs $\mathcal{I}$ (DataFrames), and then run the generator on $\mathcal{I}$ using the randomized guide while recording the choices made as K. The program $\mathcal{P}$ returned by the generator is then run on $\mathcal{I}$ to yield $\mathcal{O}$.

The sheer size of APIs such as `pandas` presents another problem in this data generation process. The large number of functions yields a huge number of possible sequences of these functions (Lines 3-4 in Figure 8.5). Even when considering sequences of length ≤ 3, the total number of sequences possible from the 119 `pandas` functions we support is around 500,000. Generating enough examples for all function sequences to cover a satisfactory portion of all the possible argument combinations is prohibitively expensive and would result in dataset of enormous size that cannot be processed and learned from in reasonable time.

However, not all sequences actually occur in practice. Users of the API come up with sequences that are useful in solving real-world examples. So, we mine GitHub and StackOverflow to collect the function sequences used in the real-world. We were able to extract around 4,300 sequences from both these sources. Then, while generating the tuples $(\mathcal{I}, \mathcal{O}, \mathcal{P}, K)$ using randomized semantics, we tweak the semantics of the call to **`Sequence`** at Line 4 in

Figure 8.5 to randomly return sequences from only this mined set of sequences.

We implement this overall technique in a tool called AutoPandas. AutoPandas consists of 25k lines of Python code, and uses TensorFlow [25] to implement the neural network models. The code is available at `https://github.com/rbavishi/autopandas`.

8.4 Evaluation

We first evaluate the feasibility and effectiveness of our technique by evaluating the end-to-end ability of AutoPandas to synthesize solutions for real-world benchmarks. We then provide deeper insights into the performance of our neural network models and compare it with two baselines to demonstrate the usefulness of the models.

8.4.1 Training and Setup

We generated 6 million (input, output, program, generator choices) training tuples (as described in Section 8.3.3) containing 2 million tuples each for programs consisting of one, two, and three function calls. Similarly, we generate 300K validation tuples with 100K tuples each for the three function sequence lengths. From these tuples we extract training and validation data for the 320 operator calls in our program candidate generator for `pandas`, and train their respective models for 10 epochs on four NVIDIA Titan V GPUs. We finished training all the models in 48 hours. All our synthesis experiments are run on a single 8-core machine containing Intel i7-7700K 4.20GHz CPUs running Ubuntu 16.04.

8.4.2 Performance on Real-World Benchmarks

We evaluated AutoPandas on 26 benchmarks taken from StackOverflow questions containing the `dataframe` tag. We ran AutoPandas with a time-out of 20 minutes. For comparison, we also implement a baseline version of AutoPandas called Baseline that follows the exhaustive depth-first guide for all operator calls except the **`Sequence`** invocation. The rationale is that given the size of the search space, it is more meaningful to compare the performance of the models backing the exploration of function arguments given the same function sequences.

Table 8.2 shows the results. The column *Depth* contains the length of the function sequence used in the official solution for the benchmark. *Cand. Explored* denotes the number of candidates both approaches had to check for correctness before arriving at one which produces the target output. *Seq. Explored* contains the number of function sequences explored (by the **`Sequence`** call at Line 4 in Figure 8.5), while the *Time* column contains the time taken (in seconds) to produce a solution if any.

AutoPandas can solve 17 out of the 26 benchmarks. The Baseline approach solves only 14. Both approaches tend to miss the 20 minute mark more often on benchmarks with higher depths, which is expected as the space of possible programs grows exponentially with

Table 8.2: Performance on Real-World Benchmarks between AutoPandas (AP) and our baseline (BL). Dashes (-) indicate timeouts by the technique.

Benchmark	Depth	Candidates Explored		Sequences Explored		Solved		Time(s)	
		AP	BL	AP	BL	AP	BL	AP	BL
SO_11881165	1	**15**	64	1	1	Y	Y	**0.54**	1.46
SO_11941492	1	783	**441**	8	8	Y	Y	12.55	**2.38**
SO_13647222	1	**5**	15,696	1	1	Y	Y	**3.32**	53.07
SO_18172851	1	-	-	-	-	N	N	-	-
SO_49583055	1	-	-	-	-	N	N	-	-
SO_49592930	1	**2**	4	1	1	Y	Y	**1.1**	1.43
SO_49572546	1	**3**	4	1	1	Y	Y	**1.1**	1.44
SO_13261175	1	**39,537**	-	18	-	**Y**	N	**300.20**	-
SO_13793321	1	**92**	1456	1	1	Y	Y	**4.16**	5.76
SO_14085517	1	**10**	208	1	1	Y	Y	2.24	**2.01**
SO_11418192	2	158	**80**	1	1	Y	Y	**0.71**	1.46
SO_49567723	2	**1,684,022**	-	2	-	**Y**	N	**753.10**	-
SO_13261691	2	**65**	612	1	1	Y	Y	**2.96**	3.22
SO_13659881	2	**2**	15	1	1	Y	Y	**1.38**	1.41
SO_13807758	2	711	**263**	2	2	Y	Y	7.21	**1.81**
SO_34365578	2	-	-	-	-	N	N	-	-
SO_10982266	3	-	-	-	-	N	N	-	-
SO_11811392	3	-	-	-	-	N	N	-	-
SO_49581206	3	-	-	-	-	N	N	-	-
SO_12065885	3	**924**	2072	1	1	Y	Y	**0.9**	4.67
SO_13576164	3	**22,966**	-	5	-	**Y**	N	**339.25**	-
SO_14023037	3	-	-	-	-	N	N	-	-
SO_53762029	3	**27**	115	1	1	Y	Y	1.90	**1.50**
SO_21982987	3	8385	**8278**	10	10	Y	Y	30.80	**13.91**
SO_39656670	3	-	-	-	-	N	N	-	-
SO_23321300	3	-	-	-	-	N	N	-	-
Total						17/26	14/26		

the length of the function sequence being explored. The guided execution of the program candidate generator enabled by neural networks allows AutoPandas to search this enormous space in reasonable time.

Even on the benchmarks that are solved by both approaches, the lower numbers in the *Candidates Explored* column indicate that our neural-backed program candidate generator indeed learns to adapt to the synthesis task at hand, generating the solution faster than the baseline. Finally the number of sequences explored in both approaches is always at most 10, and often 1, suggesting that the sequence prediction component is quite effective. The difference in time between the two approaches is relatively smaller than in candidate numbers, because AutoPandas includes the time taken to query the neural network models and interpret its results. However we believe this is fundamentally an engineering issue. Performance could easily be improved by batching queries, parallelizing exploration and speculative execution of the generator while waiting for results from the models.

Most of the benchmarks on which AutoPandas fails to find a solution involve arithmetic functions. AutoPandas's encoding does not capture arithmetic relationships readily, so its function sequence prediction is not as accurate for these sequences.

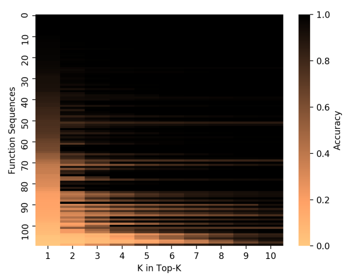
8.4.3 Analysis of Neural Network Models

Now we analyze whether the *Rank* function underlying the smart guides actually ranks choices in a useful order for synthesis.

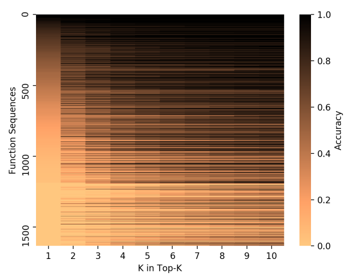
8.4.3.1 Function Sequence Prediction Performance

We single out the call to **Sequence** in our program candidate generator (Line 4 of Figure 8.5) as it is the component most critical to the performance of the generator, and dissect the performance of the neural network model backing it; on our synthetic validation dataset in Figure 8.8. In particular, we measure top-1 to top-10 accuracies on a per-sequence basis. Recall that these are the sequences mined from GitHub and StackOverflow. Figures 8.8a-8.8c show the performance of the model when predicting sequences of lengths 1, 2 and 3 respectively. As expected, the performance for shorter sequences is better as the logical distance between the input and output is lower, and therefore the encoding can capture sufficient information. Another reason for poorer accuracies at higher lengths is the fact that for large APIs like `pandas` functions often have overlapping semantics. Therefore multiple sequences may produce viable solutions for a given output example. This is reinforced by the results on real-world benchmarks in Table 8.2. In particular, the numbers in the "Sequences Explored" column for AutoPandas suggest that the model indeed predicts useful sequences, even if they don't match the ground-truth sequence.

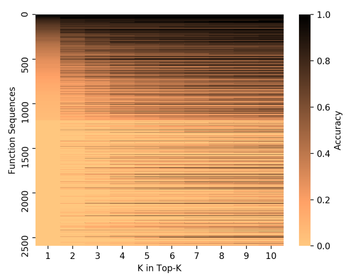
Figures 8.8d-8.8f present the expected accuracies of a purely random model on the same dataset. As expected, the accuracies are almost zero (there is a slight gradient in Figure 8.8d). The sheer number of possible sequences makes it improbable for a random model to succeed on this task; even our baseline benefited from the neural model's predictions.



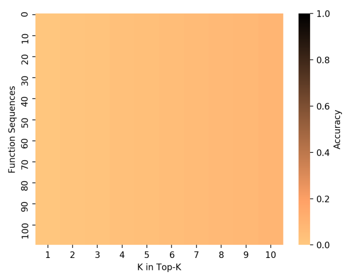
(a) Smart, Length-1



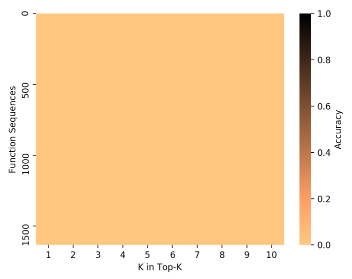
(b) Smart, Length-2



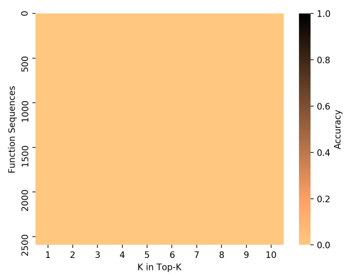
(c) Smart, Length-3



(d) Random, Length-1

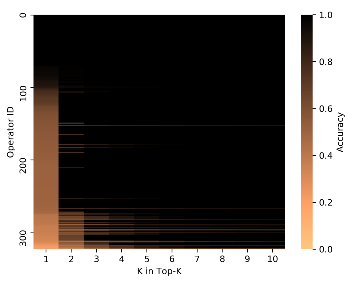


(e) Random, Length-2

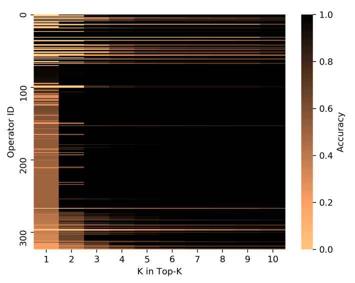


(f) Random, Length-3

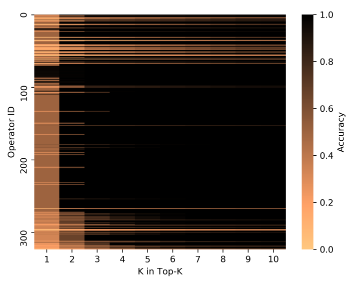
Figure 8.8: Smart Model Accuracies on Function Prediction Task, compared to a Random Baseline. Per-sequence Top-k accuracies provided. Color gives accuracy; darker is better. The color point (x, y) gives the top-x accuracy for sequence with ID y. Sequence IDs are sorted based on top-1 accuracy of the smart model.



(a) Smart Model



(b) Baseline-Deterministic



(c) Baseline-Randomized

Figure 8.9: Per-operator Top-k accuracies. Color gives accuracy; darker is better. The color point (x, y) gives the top-x accuracy for operator with ID y. Operator IDs are sorted based on top-1 accuracy of the smart model.

8.4.3.2 Comparison with Deterministic and Randomized Semantics

We demonstrate the efficacy of the smart guide by comparing the neural network models with deterministic and randomized baselines. In the deterministic baseline, the guide forces a certain fixed order of argument exploration (i.e. the exhaustive depth-first guide discussed in Section 8.2). In the randomized baseline, the guide returns values in a random order. We expect the smart neural-network-based guide to perform better than both these baselines as it is utilizing the context to influence its choices. Figure 8.9 shows the results.

We see that while a randomized approach smoothens results compared to the deterministic approach (ref. Figure 8.9c vs. Figure 8.9b), both still have significant difficulty on certain operator calls (top-left corners of all graphs). The neural network model performs quite well in comparison. There are operator calls where all the three approaches perform poorly or all perform well. The former can be attributed to insufficient information in the context. For example, if a `pandas` function supports two modes of operation which can both lead to a solution, the model may be penalized in terms of argument prediction accuracy, but may not affect its performance in the actual task. The latter case, where all approaches perform well, can be mostly attributed to small domains. For example, many `pandas` functions take an `axis` argument that can only take the value 0 or 1, which can be modeled as **`Select`**`({0,1})` in the generator. Hence the top-2 accuracy of all the approaches will be 100%.

Overall, we see that the neural-backed operators arrive at the correct choice much more quickly than their randomized or deterministic counter-parts, helping the generator as a whole to arrive at the solution more efficiently. In fact, the accuracies in Figure 8.9 are quite high for the neural-backed operators overall. We think this is a very encouraging result, as the networks are able to learn useful operator-level heuristics from the graph encodings of domain and context.

The contrast between the overall high accuracies in Figure 8.9 and the accuracies in Figure 8.8 suggests that the biggest bottleneck is predicting the correct function sequence. This and the previous observation are reinforced by the columns containing the number of candidates and function sequences explored in Table 8.2.

8.5 Discussion

Although the results suggest that the AutoPandas system as described works fairly well, the neural-backed program candidate generator we use is only one of the many possible generators in a large design space. At a high-level, our generator works by first predicting an entire sequence of functions, and then exploring the resulting space of argument combinations. However, predicting entire sequences is prone to error, especially at higher lengths, since the logical distance between the input and the target output may not allow our graph-based encoding to capture complete information about the transformation. Another possible approach is to predict only one function (along with its arguments) at a time, and make the next decision based on the output of running this function.

One of the key elements which allows neural-network backed execution of generators is our graph-based encoding of the domain and context, where relationships between elements are captured using edges. These edges can be thought of as dependency relationships. When considering dataframes as in our case, these edges (especially equality edges) capture the elements of the output that are dependent on certain elements of the input. This presents an opportunity for additional user interaction—the user, along with the input-output example, can provide additional help by pointing out the relationships between the cells of the input and output dataframe, which can be directly captured as edges in our encoding. This may give better results in cases where equality edges alone cannot distinguish the operation being performed (e.g., in distinguishing between arithmetic operations).

Overall, in this chapter, we saw that the notion of *guided* generator-based search has applications beyond fuzz testing. Much as in the prior chapter, the core of the AutoPandas technique is to cleanly separate the specification of the search space (the program candidate generator) from the technique used to sample inputs (the smart, neural network backed guide). As was shown in Section 8.4.3, the smart guide is very effective at pruning the space of programs for synthesis. But the results on real-world examples demonstrate there is room for growth. The key takeaway from Part III is that by separating the generator from its distribution, we can more efficiently solve search problems like blackbox validity fuzzing and program synthesis. Chapter 7 and this one open doors to further innovation in the efficient, automatic, distribution tuning of generators in order to tackle currently difficult search problems.

Chapter 9

Conclusion

This dissertation presented several fuzz testing methodologies, each of which had a particular goal in the scope of improvements to test-input generation and software quality more broadly.

In Part I, we saw first how a modification of the coverage-guided fuzzing algorithm—the multi-objective maximization search algorithm performed by PerfFuzz—enabled the creation of performance regression test suites, and the discovery of algorithmic complexity bugs. We saw an increased ability to discover known worst-case inputs compared to prior work, a fuzzing algorithm which used single-objective feedback and tried to bias mutations towards increasing the objective. But while Chapter 3 presented the PerfFuzz algorithm in reasonable generality, it only evaluated the algorithm for one performance objective: maximizing control-flow-graph-edge hit counts.

In Chapter 4, we introduced a reframing of this multi-objective maximization algorithm, pointing out the key notion of *waypoint* inputs to guide the fuzzer. This allowed us to generalize beyond the PerfFuzz algorithm for performance fuzzing to a general algorithm for *feedback-directed*—rather than simply coverage-guided—fuzzing. This chapter gave a better taste of the potential of feedback-directed fuzzing. First, with another performance fuzzing goal: showing how it could be used to find inputs that use unreasonable amounts of memory. Second, we saw how it could be used to target fuzzing to particular program components, e.g. hard comparisons in the code. Third, we saw the algorithm could also be used to target inputs towards newly-modified code. The FuzzFactory framework also neatly enabled us to combine these multiple objectives, in order to create truly generalized feedback-directed fuzzers.

In Part II we shifted away from the feedback component of coverage-guided fuzzing to examine the impact of mutations on the performance of fuzz testing. In Chapter 5 we presented a method to automatically identify branches guarding potentially under-explored regions of code, and target mutations towards these branches. FairFuzz's mutation targeting methodology—the mutation mask and its computation—could in fact be used to target objectives other than hitting a particular branch. We saw that this method was effective in increasing the coverage achieved by fuzz testing, especially on programs with highly nested structure. This highlighted the importance of structure-aware mutations in enabling deeper program exploration, but did not completely solve the problem of how to derive structure-aware mutations.

In Chapter 6, we explored the full potential of structure-aware mutations. Instead of simply filtering out mutations at the byte-level, as in FairFuzz, we leveraged user knowledge in the form of *random-input generators* to obtain structure-aware mutations. These random-input generators can be viewed as a mapping from an infinite sequence of bytes—created by a pseudo-random source—to a structured input. This view enables us to use byte-level mutations on the infinite byte sequence in order to obtain mutations that only step through the space of well-structured inputs. This, along with additional feedback on semantic validity, enabled the discovery of bugs deep in the semantic processing stages of programs.

In Part III, we took a deeper look at the potential behind random-input generators. Chapter 7 focused on a very similar problem to that addressed in Chapter 6: the increased generation of valid inputs from an over-approximate generator. However, Chapter 7 looked at controlling the behavior of the generator at a different level: instead of modifying an uninterpretable byte-level parameter sequence, *RLCheck* sought to create a guide which tuned the distribution of choices in the generator. We found that an adaptive guide backed by a reinforcement learning approach (Monte Carlo Control) could effectively narrow the generation of inputs the the space of valid inputs.

Finally, in Chapter 8, we saw an application domain of these search algorithms beyond automated test-input generation. Again, we leveraged the framework introduced in Chapter 7: guiding a generator at the level of "random" choices. We noticed that we could conduct program synthesis—find programs satisfying a user-provided input-output example—via essentially generator-based fuzzing, with a modified bug oracle. Instead of the generator generating program inputs, it generates programs which can be run on the user-provided input. Instead of tuning the distribution of the generator to a space of valid inputs, we used graph neural networks to bias it to producing programs satisfying the user specification early in the generation process.

The key achievement of this dissertation is revealed when taking a step back and looking at these works as a whole:

Three key components of modern fuzzing algorithms must be extended in order to use these algorithms to broadly improve software quality. A generalized notion of feedback enables coverage-guided fuzzers to find new types of bugs; well-structured mutations enable mutational fuzzers to explore deeper program states; and distribution learning methods enable the use of generator-based search for validity fuzzing and program synthesis.

This insight enables us to more rapidly adapt fuzzers and fuzz-testing-like search algorithms to different software quality problems. One direction going forward is to continue looking for software quality issues that could be solved with these strong search problems. However, there also remains much work to do around even well-known fuzzing algorithms.

While these fuzzing algorithms are very effective at finding bugs in programs on which they can be run, they cannot currently be run on all programs. Typically they assume the program under test is either a command-line tool taking in a file as input, or require a specialized side-effect-free test driver to be written. Writing these test drivers is not an easy task, and may discourage developers from using fuzz testing. Automating the creation of such drivers is key to bringing fuzz testing into the everyday developer's toolbox, and preliminary work shows great potential for impact in this direction [33, 102].

Further, as we saw in Chapters 6 and 7, when input structure information can be obtained, the efficiency of the fuzzer in exploring program states beyond the parser increases greatly. However, writing down these specifications of input structure can be another barrier to the usage of these more effective fuzz testers. Pairing fuzz testing with input structure inference—and developing input structure inference techniques that scale to real-world scenarios—may be the next great innovation in fuzz testing.

Finally, this dissertation did not explore what exactly developers should do with all the bugs with which they are presented. It is possible that some of the bugs fuzz testers find automatically do not fit the user's threshold of importance. Exploring of specifications of bug relevance that can be both understood by human developers and can effectively restrict the fuzz tester's search would help us reduce these false positives.

Solving all of these problems is highly non-trivial. However, first steps in these directions have shown that good solutions to these problems can have even broader impacts on the amelioration of software quality than the improvement of fuzz testing algorithms alone.

Bibliography

[1] CVE-2011-3414. Available from MITRE, 2011.

[2] CVE-2011-4858. Available from MITRE, 2011.

[3] CWE-400: Uncontrolled Resource Consumption. Available from MITRE, 2011. Accessed Jan 2018.

[4] CVE-2014-5265. Available from MITRE, 2014.

[5] CVE-2017-9804. Available from MITRE, 2017.

[6] `wf` - Simple word frequency counter, 2017. Accessed Jan 2018.

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

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

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

[10] Google Closure. `https://developers.google.com/closure/compiler`, 2018. Accessed August 24, 2018.

[11] Mozilla Rhino. `https://github.com/mozilla/rhino`, 2018. Accessed August 24, 2018.

[12] React.JS. `https://reactjs.org`, 2018. Accessed August 24, 2018.

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

[14] FsCheck: Random testing for .NET. `https://hypothesis.works/`, 2019. Accessed January 28, 2019.

[15] Hypothesis for Python. `https://hypothesis.works/`, 2019. Accessed January 28, 2019.

[16] JSVerify: Property-based testing for JavaScript. `https://github.com/jsverify/jsverify`, 2019. Accessed January 28, 2019.

[17] PeachFuzzer. `https://www.peach.tech/`, 2019. Accessed January 28, 2019.

[18] ScalaCheck: Property-based testing for Scala. `https://www.scalacheck.org/`, 2019. Accessed January 28, 2019.

[19] test.check: QuickCheck for Clojure. `https://github.com/clojure/test.check`, 2019. Accessed January 28, 2019.

[20] CVE-2020-7212. Available from MITRE, 2020.

[21] Address Sanitizer. `https://clang.llvm.org/docs/AddressSanitizer.html`, 2021. Accessed Apr 30, 2021.

[22] Leak Sanitizer. `https://clang.llvm.org/docs/LeakSanitizer.html`, 2021. Accessed Apr 30, 2021.

[23] Memory Sanitizer. `https://clang.llvm.org/docs/MemorySanitizer.html`, 2021. Accessed Apr 30, 2021.

[24] Undefined Behavior Sanitizer. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`, 2021. Accessed Apr 30, 2021.

[25] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[26] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. `https://testing.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html`, 2016.

[27] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations*, 2018.

[28] P. Amini and A. Portnoy. Sulley. `https://github.com/OpenRCE/sulley`, 2012. Accessed August 22nd, 2017.

[29] B. Archer and Drakkey. Radamsa: a general-purpose fuzzer. `https://gitlab.com/akihe/radamsa`, 2019. Accessed August 21, 2019.

[30] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. Nautilus: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium*, NDSS '19, 2019.

[31] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz. Ijon: Exploring Deep State Spaces via Fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612, 2020.

[32] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security*, NDSS '19, 2019.

[33] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 975–985, New York, NY, USA, 2019. Association for Computing Machinery.

[34] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[35] T. Ball, P. Mataga, and M. Sagiv. Edge Profiling Versus Path Profiling: The Showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 134–148, New York, NY, USA, 1998. ACM.

[36] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, page 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[37] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing Program Input Grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, 2017.

[38] R. Bavishi, C. Lemieux, R. Fox, K. Sen, and I. Stoica. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[39] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[40] M. Beyene and J. 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*, pages 270–279, 2012.

[41] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002, Santa Clara, CA, Aug. 2019. USENIX Association.

[42] M. Böhme. AFLFast.new. `https://groups.google.com/d/msg/afl-users/1PmKJC-EKZ0/lbzRb8AuAAAJ`, 2016. Accessed August 23rd, 2017.

[43] M. Böhme. STADS: Software Testing as Species Discovery. *ACM Trans. Softw. Eng. Methodol.*, 27(2), June 2018.

[44] M. Böhme, V. J. M. Manès, and S. K. Cha. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 678–689, New York, NY, USA, 2020. Association for Computing Machinery.

[45] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, 2017.

[46] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.

[47] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[48] S. Bratus, A. Hansen, and A. Shubina. LZfuzz: a fast compression-based fuzzer for poorly documented protocols. Technical report, Department of Computer Science, Darmouth College, 2008.

[49] A. Bugariu, V. Wüstholz, M. Christakis, and P. Müller. Automatically Testing Implementations of Numerical Abstract Domains. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 768–778, New York, NY, USA, 2018. Association for Computing Machinery.

[50] M. Bynens. In search of the perfect URL validation regex. `https://mathiasbynens.be/demo/url-regex`, 2014.

[51] C. Cadar, D. Dunbar, and D. 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.

[52] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[53] S. K. Cha, M. Woo, and D. Brumley. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, 2015.

[54] P. Chen and H. Chen. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, 2018.

[55] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, Aug. 2019. USENIX Association.

[56] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. *SIGPLAN Not.*, 46(3):265–278, Mar. 2011.

[57] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734. Association for Computational Linguistics, 2014.

[58] K. Claessen and J. 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.

[59] L. A. Clarke. A program testing system. In *Proc. of the 1976 annual conference*, pages 488–491, 1976.

[60] E. Coppa, C. Demetrescu, and I. Finocchi. Input-Sensitive Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 89–98, New York, NY, USA, 2012. ACM.

[61] N. Coppik, O. Schwahn, and N. Suri. MemFuzz: Using Memory Accesses to Guide Fuzzing. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 48–58. IEEE, 2019.

[62] D. Coppit and J. 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, pages 356–359, New York, NY, USA, 2005. ACM.

[63] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2123–2138, New York, NY, USA, 2017. ACM.

[64] S. A. Crosby and D. S. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[65] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[66] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54:69–77, Sept. 2011.

[67] R. Feldt and S. Poulding. Finding test data with specific properties via metaheuristic search. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 350–359. IEEE, 2013.

[68] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program Synthesis Using Conflict-driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 420–435, New York, NY, USA, 2018. ACM.

[69] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. *SIGPLAN Not.*, 52(6):422–436, June 2017.

[70] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing Data Structure Transformations from Input-output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 229–239, New York, NY, USA, 2015. ACM.

[71] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[72] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, page 6, USA, 2000. USENIX Association.

[73] O. S. Foundation. OpenSSL: Cryptography and SSL/TLS Toolkit. `https://www.openssl.org/`, 1999. Accessed Apr 30, 2021.

[74] G. Fraser and A. 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.

[75] G. Fraser and A. Arcuri. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):8:1–8:42, Dec. 2014.

[76] S. Frizell. Report: Devastating Heartbleed Flaw Was Used in Hospital Hack. *Time*, 2014. Accessed Apr 30, 2021.

[77] V. Ganesh, T. Leek, and M. Rinard. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, 2009.

[78] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. 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*, pages 225–234, 2010.

[79] P. Godefroid. Fuzzing: Hack, Art, and Science. *Commun. ACM*, 63(2):70–76, Jan. 2020.

[80] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, 2008.

[81] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, 2005.

[82] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Symposium on Network and Distributed System Security*, NDSS '08, 2008.

[83] P. Godefroid, H. Peleg, and R. Singh. Learn & Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 50–59, Piscataway, NJ, USA, 2017. IEEE Press.

[84] S. F. Goldsmith, A. S. Aiken, and D. 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, pages 395–404, New York, NY, USA, 2007. ACM.

[85] Google. Continuous fuzzing of open source software. `https://opensource.google.com/projects/oss-fuzz`, 2019. Accessed March 26, 2019.

[86] Google. Set of tests for fuzzing engines. `https://github.com/google/fuzzer-test-suite`, 2019. Accessed March 20, 2019.

[87] R. Gopinath, B. Mathis, and A. Zeller. Mining Input Grammars from Dynamic Control Flow. In *Proceedings of the 2019 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 1–12, New York, NY, USA, 2020. Association for Computing Machinery.

[88] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[89] J. Graham-Cumming. Incident report on memory leak caused by Cloudflare parser bug. `https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/`, 2017. Accessed Apr 30, 2021.

[90] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 156–166. IEEE, 2012.

[91] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*, pages 408–418. IEEE, 2009.

[92] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, 2013.

[93] M. Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, pages 342–357. IEEE Computer Society, 2007.

[94] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[95] S. Hocevar. zzuf. `http://caca.zoy.org/wiki/zzuf/`, 2007. Accessed August 22nd, 2017.

[96] M. R. Hoffmann, B. Janiczak, and E. Mandrikov. Eclemma-jacoco java code coverage library, 2011.

[97] C. Holler, K. Herzig, and A. Zeller. Fuzzing with Code Fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012.

[98] P. Holser. junit-quickcheck: Property-based testing, JUnit-style. `https://pholser.github.io/junit-quickcheck`, 2014. Accessed January 11, 2019.

[99] M. Höschele and A. Zeller. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, 2016.

[100] A. D. Householder and J. M. Foote. Probability-Based Parameter Selection for Black-Box Fuzz Testing. Technical report, Carnegie Mellon University Software Engineering Institute, 2012.

[101] S. Inc. Heartbleed. `https://heartbleed.com/`, 2014. Accessed Apr 30, 2021.

[102] K. Ispoglou, D. Austin, V. Mohan, and M. Payer. FuzzGen: Automatic Fuzzer Generation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2271–2287. USENIX Association, Aug. 2020.

[103] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[104] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[105] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *ArXiv e-prints*, Dec. 2014.

[106] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2123–2138, New York, NY, USA, 2018. ACM.

[107] B. Korel. Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879, 1990.

[108] H. Krasner. The Cost of Poor Quality Software in the US: A 2018 Report. Technical report, Consortium for Information & Software Quality, 2018.

[109] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen. RFUZZ: Coverage-directed Fuzz Testing of RTL on FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, pages 28:1–28:8, New York, NY, USA, 2018. ACM.

[110] LafIntel. Circumventing Fuzzing Roadblocks with Compiler Transformations. `https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/`, 2016. Accessed March 20, 2019.

[111] L. Lampropoulos, D. Gallois-Wong, C. Hriţcu, J. Hughes, B. C. Pierce, and L.-y. Xia. Beginner's Luck: A Language for Property-based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 114–129, New York, NY, USA, 2017. ACM.

[112] L. Lampropoulos, M. Hicks, and B. C. Pierce. Coverage Guided, Property Based Testing. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[113] C. Lattner and V. 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, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[114] C. Lemieux, R. Padhye, K. Sen, and D. Song. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 254–265, New York, NY, USA, 2018. ACM.

[115] C. Lemieux and K. 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.

[116] M. Leung and C. Commisso. Canadians filing taxes late due to 'Heartbleed' bug won't face penalties: CRA. `https://www.ctvnews.ca/canada/canadians-filing-taxes-late-due-to-heartbleed-bug-won-t-face-penalties-cra-1.1767727`, 2014. Retrieved October 23, 2020.

[117] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, 2017.

[118] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated Graph Sequence Neural Networks. *CoRR*, abs/1511.05493, 2015.

[119] Z. Lin and X. Zhang. Deriving Input Syntactic Structure from Execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 83–93, New York, NY, USA, 2008. Association for Computing Machinery.

[120] Z. Lin, X. Zhang, and D. Xu. Reverse Engineering Input Syntactic Structure from Program Execution and Its Applications. *IEEE Transactions on Software Engineering*, 36(5):688–703, 2010.

[121] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition.* Addison-Wesley Professional, 1st edition, 2014.

[122] V. Livinskii, D. Babokin, and J. Regehr. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020.

[123] A. Löscher and K. Sagonas. Targeted Property-based Testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 46–56, New York, NY, USA, 2017. ACM.

[124] A. Loscher and K. Sagonas. Automating Targeted Property-Based Testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, volume 00, pages 70–80, Apr 2018.

[125] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. 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, pages 190–200, New York, NY, USA, 2005. ACM.

[126] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, Aug. 2019. USENIX Association.

[127] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. Fuzzing: Art, Science, and Engineering. *CoRR*, abs/1812.00140, 2018.

[128] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 94–105, New York, NY, USA, 2016. Association for Computing Machinery.

[129] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller. Parser-Directed Fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 548–560, New York, NY, USA, 2019. Association for Computing Machinery.

[130] P. M. Maurer. Generating test data with enhanced context-free grammars. *Ieee Software*, 7(4):50–55, 1990.

[131] P. 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, pages 153–163, Washington, DC, USA, 2011. IEEE Computer Society.

[132] Microsoft. Gated Graph Neural Network Samples. https://github.com/Microsoft/gated-graph-neural-network-samples, 2017. Accessed October 17th, 2018.

[133] B. Miller, M. Zhang, and E. Heymann. The Relevance of Classic Fuzz Testing: Have We Solved This One? *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[134] B. P. Miller, G. Cooksey, and F. Moore. An Empirical Study of the Robustness of MacOS Applications Using Random Testing. In *Proceedings of the 1st International Workshop on Random Testing*, RT '06, page 46–54, New York, NY, USA, 2006. Association for Computing Machinery.

[135] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[136] B. P. Miller, D. Koski, C. Pheow, L. V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin-Madison, 1995.

[137] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223, 1976.

[138] G. J. Myers. *Art of Software Testing.* John Wiley & Sons, Inc., New York, NY, USA, 1979.

[139] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

[140] S. Nilizadeh, Y. Noller, and C. S. Păsăreanu. DiffFuzz: Differential Fuzzing for Side-channel Analysis. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 176–187, Piscataway, NJ, USA, 2019. IEEE Press.

[141] A. Nistor, L. Song, D. Marinov, and S. Lu. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press.

[142] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner. Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, pages 1475–1482, New York, NY, USA, 2018. ACM.

[143] I. Ogrodnki. 900 SINs stolen due to Heartbleed bug: Canada Revenue Agency. `https://globalnews.ca/news/1269168/900-sin-numbers-stolen-due-to-heartbleed-bug-canada-revenue-agency/`, 2014. Retrieved October 23, 2020.

[144] OpenSSL. OpenSSL Security Advisory [07 Apr 2014]. `https://www.openssl.org/news/secadv/20140407.txt`, 2014. Retrieved October 23, 2020.

[145] OpenSSL. OpenSSL Security Advisory [26 Sep 2016]. `https://www.openssl.org/news/secadv/20160926.txt`, 2016. Retrieved October 23, 2020.

[146] OW2 Consortium. ObjectWeb ASM. `https://asm.ow2.io`, 2018. Accessed August 21, 2018.

[147] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, 2007.

[148] R. Padhye, C. Lemieux, and K. Sen. JQF: Coverage-guided Property-based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '19, 2019.

[149] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. L. Traon. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '19, 2019.

[150] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), Oct. 2019.

[151] M. Papadakis and K. 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, pages 39–50, New York, NY, USA, 2011. ACM.

[152] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *ICML*, 2017.

[153] H. Peng, Y. Shoshitaishvili, and M. Payer. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[154] J. Pereyda. BooFuzz. `[https://github.com/jtpereyda/boofuzz`, 2016. Accessed May 4th, 2020.

[155] N. Perlroth. Security Experts Expect 'Shellshock' Software Bug in Bash to Be Significant. *The New York Times*, 2014. Accessed Apr 30, 2021.

[156] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632. IEEE, 2017.

[157] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2017.

[158] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart Greybox Fuzzing. *CoRR*, abs/1811.09447, 2018.

[159] S. Poeplau and A. Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198. USENIX Association, Aug. 2020.

[160] S. Poeplau and A. Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *28th Annual Network and Distributed System Security Symposium*, NDSS '21, 2021.

[161] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Network and Distributed System Security Symposium*, NDSS '17, 2017.

[162] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 861–875, Berkeley, CA, USA, 2014. USENIX Association.

[163] S. Reddy, C. Lemieux, R. Padhye, and K. Sen. Quickly Generating Diverse Valid Test Inputs with Reinforcement Learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1410–1421, New York, NY, USA, 2020. Association for Computing Machinery.

[164] T. Ringer, D. Grossman, D. Schwartz-Narbonne, and S. Tasiran. A Solver-aided Language for Test Input Generation. *Proc. ACM Program. Lang.*, 1(OOPSLA):91:1–91:24, Oct. 2017.

[165] K. Sen, D. Marinov, and G. 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.

[166] K. Sen, G. Necula, L. Gong, and W. Choi. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 842–853. ACM, 2015.

[167] K. Serebryany. libFuzzer. `http://llvm.org/docs/LibFuzzer.html`, 2016. Accessed August 25th, 2017.

[168] K. Serebryany, V. Buka, and M. Morehouse. Structure-aware fuzzing for Clang and LLVM with libprotobuf-mutator, 2017.

[169] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[170] E. G. Sirer and B. N. Bershad. Using Production Grammars in Software Testing. In *Proceedings of the 2Nd Conference on Domain-specific Languages*, DSL '99, pages 1–13, New York, NY, USA, 1999. ACM.

[171] L. Song and S. 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, pages 561–578, New York, NY, USA, 2014. ACM.

[172] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, 2016.

[173] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*, chapter 5, pages 120–124. MIT Press, 2018.

[174] R. Swiecki. honggfuzz. `http://honggfuzz.com/`, 2016. Accessed August 7, 2018.

[175] L. Szekeres. FuzzBench: 2020-09-07 Sample Report . `https://web.archive.org/web/20210503193514/https://www.fuzzbench.com/reports/sample/index.html`. Accessed May 3rd, 2021.

[176] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-Aware Greybox Fuzzing. In *41st International Conference on Software Engineering*, ICSE '19, 2019.

[177] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, 2010.

[178] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu. MemLock: Memory Usage Guided Fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 765–777, New York, NY, USA, 2020. Association for Computing Machinery.

[179] V. Wüstholz and M. Christakis. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1398–1409, New York, NY, USA, 2020. Association for Computing Machinery.

[180] V. Wüstholz and M. Christakis. Targeted Greybox Fuzzing with Static Lookahead Analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 789–800, New York, NY, USA, 2020. Association for Computing Machinery.

[181] X. Yang, Y. Chen, E. Eide, and J. 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.

[182] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 140–150. ACM, 2007.

[183] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 769–786, 2019.

[184] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 745–761, Berkeley, CA, USA, 2018. USENIX Association.

[185] M. Zalewski. American Fuzzy Lop. `http://lcamtuf.coredump.cx/afl`, 2014. Accessed August 18th, 2017.

[186] M. Zalewski. Bash bug: the other two RCEs, or how we chipped away at the original fix (CVE-2014-6277 and '78). `https://web.archive.org/web/20210428205228/http://lcamtuf.blogspot.com/2014/10/bash-bug-how-we-finally-cracked.html`, 2014. Accessed May 2, 2020.

[187] M. Zalewski. Pulling JPEGs out of thin air. `https://web.archive.org/web/20210123014427/https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html`, 2014. Accessed May 2, 2020.

[188] M. Zalewski. FidgetyAFL. `https://groups.google.com/d/msg/afl-users/fOPeb62FZUg/CES5lhznDgAJ`, 2016. Accessed August 23rd, 2017.

[189] M. Zalewski. American Fuzzy Lop Technical Details. `http://lcamtuf.coredump.cx/afl/technical_details.txt`, 2017. Accessed August 18th, 2017.

[190] D. Zaparanuks and M. Hauswirth. Algorithmic Profiling. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 67–76, New York, NY, USA, 2012. ACM.

[191] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.