Automating Software Testing with High Quality Input Generation and AI-Powered Synthesis

Vasudev Vikram

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Dr. Rohan Padhye (Chair), Carnegie Mellon University
Dr. Claire Le Goues, Carnegie Mellon University
Dr. Joshua Sunshine, Carnegie Mellon University
Dr. Caroline Lemieux, University of British Columbia
Dr. Michael Hicks, Amazon

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Keywords: Software Testing, Program Analysis

Abstract

Software testing is crucial for ensuring the quality, reliability, and security of applications. It helps identify bugs, validate functionality, and improve overall system performance. Automated software testing techniques such as property-based testing and fuzz testing are a powerful techniques for discovering corner cases and unexpected behaviors, exploring a vast range of scenarios that hand-written unit tests might overlook.

Coverage-guided fuzzing and generator-based fuzzing perform a guided exploration of the test-input space and evolving a corpus of "interesting" inputs that achieve new code coverage. However, code coverage alone is not the best metric to assess test-input quality; an input may cover certain lines of code but fail to test specific behaviors of the program. To test programs with specific input formats and semantic constraints, researchers have developed structure-aware fuzzing techniques that combine coverage-guided fuzzing with hand-written generators capable of producing structured inputs that satisfy specific pre-conditions.

This thesis presents a set of techniques to improve the quality and scale automated test input generation for software. By expanding beyond traditional metrics of code coverage, we can (1) incorporate finer-grained execution feedback and (2) automatically create stronger input generators, resulting in higher quality test inputs.

First, we propose to investigate the synthesis of a high quality test-input suite by bounding the size of the input generators and incorporating finer-grained execution feedback in mutation score. To this end, we develop and evaluate Bonsai Fuzzing and Mu2, which are able to synthesize test inputs that are more *concise* and have higher fault-finding capability.

Second, we study the automation of writing property-based tests, which requires writing generators and property assertions. We create Proptest-AI, which leverages large language models (LLMs) and treats property-based test synthesis as a code generation task. To evaluate test quality, we introduce the notion of *property coverage*, which checks whether a given property-based test is able to detect artificially injected property violations from the program.

Third, we look to explore the impact of coverage-guided fuzzing on the space of structured inputs. We propose to study this effect through analysis on parametric input generators and a case study into property-based testing of Cedar, a domain specific language used for writing authorization policies. In our analysis, we find that random generation of Cedar inputs with a complex type-directed generator is able to outperform coverage-guided fuzzing in terms of input efficiency and diversity.

My proposed work aims to further explore the relationship between generator complexity and coverage guidance, and additionally explore AI-based methods to automatically synthesize strong generators for structured inputs.

Contents

Chapter 1

Introduction

In software development, ensuring the reliability and robustness of applications is of immense importance; software bugs can lead to significant financial and reputational damage, with costs running into trillions of dollars [1]. For instance, the Log4j vulnerability, discovered in 2021, demonstrated the vast scale and impact that a single software bug can have, exposing the personal information of 147 million of consumers and costing Equifax $700 million in mitigation efforts.

The modern era has also begun to feature the widespread adoption of generative AI to produce code; as of October 2024, it's estimated that *over 25% of code* produced at Google is generated by AI [2]. Given the sheer volume of code that generative AI unlocks for software developers and companies and its potential to produce error-prone code [3], the need to validate the correctness of code is more crucial than ever.

To this end, software testing is and has been the most widely adopted process of ensuring that software behaves correctly, with the market size valued at $51.8 billion in 2023 [4]. In particular, unit testing is an integral component of the software development life cycle that aims to catch the detection of errors in individual components of a program. Despite its importance, though, unit testing often necessitates considerable manual effort and presents challenges for developers in identifying all possible edge cases and unexpected behaviors in code. Given the limitations of human reasoning in covering all possible software inputs and outcomes, there has been a growing demand and adoption of more efficient and comprehensive testing methodologies.

Over the years, automated testing techniques such as property-based testing [5] and fuzz testing[6] have gained attention for their effectiveness in uncovering bugs within complex software. Research building on these approaches have primarily concentrated on advancing the underlying algorithms to better explore the coverage of software. However, there is still a significant gap in enhancing our techniques to improve the *quality* of test inputs beyond code coverage and *automating* the process of writing strong test-input generators.

This thesis investigates three problems in the field of automated software testing. First, we aim to improve the quality of inputs produced in fuzz testing by optimizing conciseness and fault-detection capability. Second, we investigate the potential for leveraging generative AI to automatically synthesize high quality property-based tests, which could significantly enhance the adoption and applicability of automated testing techniques across a wider range

of software. Finally, we focus on exploring the impact of various input generation strategies in generator-based fuzzing for software with domain-specific input constraints, aiming to identify the most effective methods for diverse input generation. We finally propose new methods of automatically synthesizing strong input generators by leveraging advances in generative AI and language models.

1.1 Thesis Statement

> Contemporary randomized test generation efforts have focused on algorithmic improvements for maximizing code coverage with simple input generation strategies and implicit oracles. We show that the testing efficacy can be enhanced by (1) carefully shaping the input space via specially crafted generators and (2) incorporating fault-finding capability into the feedback loop.

1.2 Outline

The remainder of the proposal is outlined as follows. Chapter 2 provides important background and terminology about property-based testing and fuzz testing used throughout the document. Chapter 3 investigates techniques for synthesizing test-input suites that are concise and robust to artifically-injected faults. Chapter 4 investigates the potential of large language models to automatically synthesize and evaluate property-based tests. Chapter 5 investigates the effect of various parametric input generation strategies, studying input mutation strategies in generator-based fuzzing and hand-written generators on domains with complex input constraints. Finally, chapter 6 outlines proposed work of studying the impact of coverage guidance on generator-based fuzzing and synthesizing strong input generators by designing an LLM-based agent.

Chapter 2

Background

This chapter provides the background and terminology needed to appreciate the contributions in the later chapters of the thesis.

2.1 Property-Based Testing

Property-based testing (PBT) is a powerful testing technique for testing properties of a program through generation of inputs. Unlike traditional testing methods that rely on manually written test cases and examples, PBT uses automatic generation of a wide range of inputs to invoke a diverse set of program behaviors. PBT was first popularized by the Quickcheck [5] library in Haskell, and has been used to find a plethora of bugs in a variety of real-world software [7, 8, 9, 10].

Formally, property-based testing can be defined as follows: let $f : \mathrm{X} \to \Omega$ be a function under test, and $x \in \mathrm{X}$ be a test input. We want to verify that $\forall x \in \mathrm{X} : p(x) \implies q(f(x))$, where p is a precondition that valid inputs must satisfy, and q is a postcondition on the output of f. q can be a conjunction of component properties, that is, $q = q_1 \wedge q_2 \wedge \ldots q_k$.

In practice, for a property (specified in natural language) ϕ, we can implement a set of corresponding test T. consists of:

1. A *generator* function $gen : R \to \mathrm{X}$ that samples from X to produce a random input in X, i.e. $x = gen(r)$ given a source of randomness r.
2. A collection of property assertions $\{\alpha_1, \ldots, \alpha_n\}$ that check if ϕ holds.

If T returns false, e.g. $\exists x' \in \mathrm{X} : p(x') \wedge \neg q(f(x'))$, then this is considered a bug in f and we have witness input x' as a counterexample to property ϕ. Although PBT cannot prove the absence of property violations, it is nevertheless an improvement over testing specific hard-coded inputs and outputs as is commonly done in unit testing.

We next describe how our formal definition translates to code in Hypothesis [11], a popular PBT library for Python. Suppose our function under test f is the Python `sorted` function, which takes in a `list` as input and returns a sorted version. We want to test the property that the elements of the sorted list are monotonically increasing.

First, we must write our generator *gen* that samples an input from the input space of lists. An example of such a generator is the `generate_lists` function in lines 4–7 of

```
1  from hypothesis import given, strategies as st
2
3  # Random list generator
4  @st.composite
5  def generate_lists(draw):
6    return draw(st.lists(elements=st.integers(),
7                         min_size=1))
8
9  # Property-based test for checking monotonicity
10 @given(lst=generate_lists())
11 def test_sorted(lst):
12   sorted_lst = sorted(lst)
13   assert all(sorted_lst[i] <= sorted_lst[i + 1]
14               for i in range(len(sorted_lst) - 1))
```

Figure 2.1: Example property-based tests in Hypothesis for the Python `sorted` function to sort lists. The `test_sorted` function uses the generator function `generate_lists` and checks the property that the output list elements are monotonically increasing.

Figure 2.1. Hypothesis has a built-in set of sampling *strategies* for various data structures. The `lists` and `integers` strategies in line 6 are used to randomly generate and return a Python integer list of size ≥ 1.

Next, we must write the test T that takes in an input x and returns $q(f(x))$. Our property ϕ is that the elements of the sorted list are monotonically increasing. An example of such a test is `test_sorted` in Figure 2.1. In line 12, the `sorted` function is invoked on the input `lst`. Then, lines 13–14 contain an assertion α_1 that checks the elements of the sorted listed are increasing. T returns *true* if $q(f(x))$ is true, i.e. there is no assertion failure.

Finally, to complete the property-based test, we must invoke our generator to sample random inputs and call the parametrized test on the input. This is done using the Hypothesis `@given` decorator, as seen in line 10. The decorator specifies that the input `lst` of our test `test_sorted` should use `generate_lists` as the generator.

While the property-based tests shown in Figures 2.1 are valid and will properly run, they are not necessarily the *best* property-based tests for the `sorted` function. Perhaps the user would like validate the behavior of `sorted` on the empty list, which is not an input produced by our generator due to the `min_size=1` constraint in line 7 We expand more on evaluating the quality of a property-based test in Chapter 4.

2.2 Coverage-Guided Fuzz Testing

Coverage-guided greybox fuzzing (CGF) is a technique for automatic test-input generation using lightweight program instrumentation. It was first popularized by open-source tools such as AFL [12] and libFuzzer [13], but has since been heavily studied and variously extended in academic research [14, 15, 16, 17, 18, 19, 20, 21, 22, 23].

Algorithm 1 describes the basic greybox fuzzing algorithm, with many details elided. First, a corpus of test inputs is initialized with a set of one or more *seed* inputs (Line 2),

Algorithm 1 Coverage-guided greybox fuzzing

```
1: procedure CGF(Program P, Set of inputs seeds, Budget T)
2:     corpus ← seeds                                          ▷ Initialize saved inputs
3:     repeat                                                  ▷ Fuzzing loop
4:         x ← PickInput(corpus)                               ▷ Sample using heuristics
5:         x' ← MutateInput(x)                                 ▷ Synthesize new input
6:         if running P(x') leads to a crash then
7:             raise x'                                        ▷ Bug found!
8:         if coverage(P, x') ⊈ ⋃_{x∈corpus} coverage(P, x) then
9:             corpus ← corpus ∪ x'
10:    until budget T
11:    return corpus                                           ▷ Final corpus
```

which could be user-provided or randomly generated. Then, in each iteration of the fuzzing loop (Line 3), a new input is synthesized by first picking an existing input x from the corpus (Line 4) and then performing random mutations to produce x' (Line 5). The heuristics to sample an input (PickInput) vary, and often use some sort of energy schedule [14]. Some inputs may also be marked as *favored*, and receive higher energy than other inputs. The random mutations performed on x to get x' (MutateInput) also vary depending on the known format of inputs (e.g., bitflips for binary data or random keyword insertion for text files).

The program under test P is then executed with the new input x', using lightweight instrumentation to collect code coverage during execution. The function coverage referenced in Algorithm 1 returns a set of program locations executed when processing an input. If the run of x' causes new code to be covered (Line 8), then x' is saved to the corpus (Line 9); thus, x' may be used as the basis for further input mutation in subsequent iterations of the fuzzing loop. If the execution of any synthesized input x' causes the program to crash, then a bug is reported (Line 7). The fuzzing loop continues until a user-provided resource budget T runs out (Line 10), where this budget may be in terms of the number of fuzzing trials (i.e., iterations of the fuzzing loop) or in terms of wall-clock time. The corpus of fuzzer-synthesized test inputs is finally returned (Line 11) and may be used either as a regression test suite, for seeding future fuzzing campaigns, or for other applications [24, 25, 26, 27, 28]. The quality of the final test-input corpus is often evaluated using code coverage [29, 30], though mutation scores have also been used [27].

2.3 Generator-based Fuzz Testing

Generator-based fuzzing [31, 32] is a technique for testing programs with randomly generated input data produced via a domain-specific generation function, which samples inputs conforming to some data type or input-format structure. Parametric generators [18, 19, 33, 34, 35, 36, 37] enable mutations to be performed on inputs produced by such generators. This unlocks the benefits of coverage-guided greybox fuzzing [12, 13, 38, 39], which incorporate a feedback loop to guide input generation.

```
class Node {
  Node left;
  Node
  ↪ right;
  byte data;
}
```

(a) Binary tree node type.

```
Node generateNode(FuzzedDataProvider
↪ *provider) {
  Node node = new Node();
  if (provider->ConsumeBool()) {
    node.left = generateNode(provider);
  }
  if (provider->ConsumeBool()) {
    node.right = generateNode(provider);
  }
  node.data =
  ↪ provider->ConsumeIntegral<uint8>(10);
  return node;
}
```

(b) OSS-fuzz-style C++ generator.

```
Node generateNode(Random
↪ random) {
  Node node = new Node();
  if (random.nextBoolean()) {
    node.left =
    ↪ generateNode(random);
  }
  if (random.nextBoolean()) {
    node.right =
    ↪ generateNode(random);
  }
  node.data =
  ↪ random.nextByte(10);
  return node;
}
```

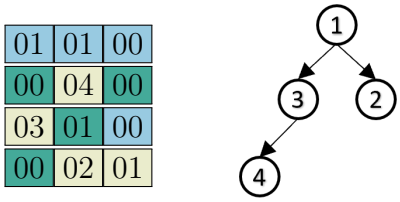
(c) Quickcheck-style Java generator.

Figure 2.2: A simplified generator for binary tree nodes in C++ (libFuzzer-style) and Java (JQF-style). While designed for random sampling, grey-box fuzzers such as Zest [19], libFuzzer [13], and AFL [12] supply a deterministic sequence of choices backed by a fixed byte stream that can be mutated.

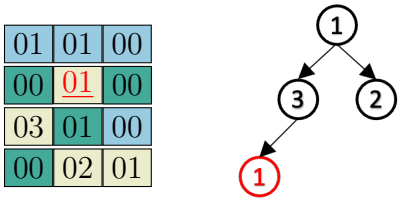
The key idea behind parametric generators is to treat generator functions as *decoders* of an arbitrary sequence of bytes, producing structurally valid inputs given any pseudo-random input sequence. Figure 2.2 depicts examples of such generators in C++ (via libFuzzer's `FuzzedDataProvider` [40]) and in Java (via JQF [18]) for sampling binary trees; in the latter case, the `Random` parameter is a facade for an object that extracts values from a regular `InputStream`. Fig. 2.3a depicts an example of the decoding process, with bytes color-mapped to corresponding decisions in the generator functions from Fig. 2.2.

By providing the byte-sequence decoded by the generator to a conventional mutation-based fuzzing algorithm, parametric generators get structured mutations "for free". Fig. 2.3b shows how a small bit-flip in the byte sequence leads to a small change in the data contained in the corresponding binary tree.

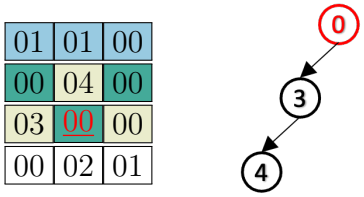
This combination of (a) a method to produce structurally valid inputs, and (b) a method to make small changes to structurally valid inputs, together enables *structure-aware grey-box fuzzing*, resulting in an ability to test deep program states beyond syntax parsing and validation [19].



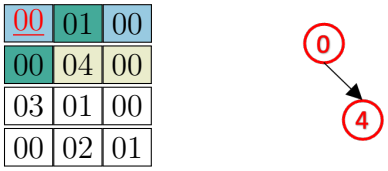
(a) Seed input



(b) Mutation I.



(c) Mutation II.



(d) Mutation III.

Figure 2.3: Four inputs as well as their corresponding binary tree object via the `generateNode` method (ref. Fig. 2.2). The changed bytes are highlighted in red.

Chapter 3

The Inputs We Saved Along the Way

Fuzzing is traditionally used to discover inputs that crash programs and reveal security vulnerabilities [6, 15, 16, 17, 19, 22, 23, 29, 41, 42]. In the absence of new bugs, fuzzers are evaluated based on code coverage achieved during the fuzzing campaign [30, 43]. However, in the vast majority of fuzzing research, the end goal is to find bugs in the moment [29]; not much attention is paid to the inputs saved along the way.

In this chapter, we explicitly focus on the quality of the test-input corpus produced at the end of a fuzzing campaign. Such a corpus can be used for continuous regression testing during subsequent program development. This practice is recommended by Google's OSS-Fuzz [24], and is already adopted by some mature projects [44]. For example, in SQLite, *"Historical test cases from AFL, OSS Fuzz, and dbsqlfuzz are collected [...] and then rerun by the* `fuzzcheck` *utility program whenever one runs* `make test`*"* [25]. Similarly, OpenSSL uses several distinct fuzzer-generated corpora and their corresponding fuzz drivers for continuous testing [26]. Even though these test corpora are used for regression testing, the only metric being targeted by conventional greybox fuzzers is code coverage. However, coverage alone is not the necessarily the only important quality in regression tests nor the strongest predictor of fault detection ability [45, 46]. Thus, in the remaining sections of this chapter, we investigate the following research question:

Can we use fuzz testing to synthesize a test suite of *high-quality* inputs?

What are some qualities we would like out of our resulting test-input corpus? We now detail two techniques used to synthesize a test-input corpus resulting in inputs that are (1) concise and (2) have high fault-finding capability. To accomplish this, we improve upon the coverage-guided fuzzing algorithm by (1) bounding and iteratively growing the size of the generators using Bonsai Fuzzing, and (2) incorporating mutation-analysis into the execution feedback using Mu2.

3.1 Growing a Test Corpus with Bonsai Fuzzing

3.1.1 Introduction

We describe a new technique for automatically generating a concise corpus of test inputs having a well-defined syntax and non-trivial semantics (e.g. for a compiler).

This project originated when the authors were faced with the task of generating a test corpus for use in an undergraduate compilers course. The course project targets the ChocoPy programming language [47]. ChocoPy is a statically typed subset of Python, designed specifically for education. In a ChocoPy-based course, students are expected to build a compiler in Java that statically checks and then translates ChocoPy programs to RISC-V assembly. Student projects can be autograded by comparing their compilers' output at various stages—parser, type checker, and code generator—with the corresponding output produced by a reference implementation. The ChocoPy project ships with a full formal specification of the language, the reference compiler, and a auto-grading infrastructure for use in such a course. However, a test suite—on which the auto-grader should run—is not provided to instructors. When starting their project, students are provided with a suite of ChocoPy test programs and the autograder, which together serve as a partial executable specification. This workflow simulates *test-driven development*, while also enabling students to continuously get feedback about their progress. For instructors, writing test cases to validate every language feature is a tedious task; we wanted to *automatically* synthesize such a test corpus. This chapter describes the technique we developed for this purpose. In particular, we focus on the problem of automatically generating test cases that exercise the typechecker, since generating well-typed programs is known to be a difficult problem [48, 49, 50, 51].

This task presents two conflicting challenges: (1) the generated test suite must be *comprehensive* in covering various semantics of the language, including corner cases; (2) the test suite must be *concise* and readable; in particular, each test case should be small in size so that test failures can guide students towards identifying which feature was incorrectly implemented. The conflict is apparent from previous work [52] which indicates that automated test generation for covering difficult program branches works better with larger test cases.

Much work has been done on automatically generating concise and comprehensive unit test suites [53, 54, 55]. However, this work mainly focuses on generating test code as sequences of method calls while minimizing the number of test cases or size of the entire test suite. Our goal is to generate non-trivial test *inputs* (e.g. strings) while minimizing the *individual size of each test case*, on average. This is because our conciseness goals are related to readability and debuggability [56, 57] rather than reducing the cost of test execution [58].

The state-of-the-art in concise automatic test case generation for structured input domains such as compilers is as follows: first, perform some form of random fuzzing [6, 59] to automatically discover unexpected or coverage-increasing inputs. Then, perform test-case reduction [60, 61] on every fuzzer-saved input in order to find a corresponding (locally) minimal input that causes the test program to exhibit the same behavior. For example,

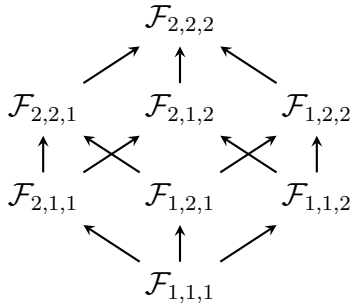


Figure 3.1: An example bonsai fuzzing architecture—A lattice of coverage-guided size-bounded grammar-based fuzzers $\mathcal{F}_{m,n,d}$, ordered by three size bounds on the syntax of the test cases they produce: number of unique identifiers m, maximum sequence length n, and maximum nesting depth d. Test cases flow along directed edges: the inputs generated by each fuzzer are used as the seed inputs to its successors. The result of bonsai fuzzing is the corpus produced by the top-most element.

CSmith [48] and C-Reduce [32] complement each other by respectively generating and minimizing C programs for automated testing of C compilers.

By their very nature, fuzzer-generated inputs exercise program features chaotically. This can make isolating the most significant features of a fuzzer-saved test input challenging both for humans and for minimization algorithms. Further, to make the test-case minimization problem tractable, algorithms such as delta-debugging [60] and its variants perform only local optimization.

In this section, we present *Bonsai Fuzzing*, a technique for automatically generating a concise and comprehensive test corpus of structurally complex inputs. Our key insight is that instead of reducing large convoluted inputs that exercise many program features at once, we can grow a concise test corpus *bottom-up*. Bonsai Fuzzing generates small inputs *by construction* in an iterative evolutionary algorithm: the first round generates tiny trivial inputs and then each successive round generates inputs of slightly larger size by mutating inputs saved in a previous round. In particular, we first define a procedure to sample syntactically valid inputs from a grammar specification using bounds on the number of identifiers, linear repetitions, and nested expansions in the resulting derivation trees. We then define a partial order over coverage-guided bounded grammar fuzzers (CBGFs). For any given desired size bound, this partial order results in a lattice of CBGFs. A corpus produced by each CBGF in this lattice is used as a set of seed inputs for all successive CBGFs. The bottom element of the lattice has minimum size bound—a fuzzer with no good seed inputs—and the top element has the maximum desirable size bound—a fuzzer that produces the final test corpus. Fig. 3.1 visualizes bonsai fuzzing for a given size bound.

3.1.2 Proposed Technique

Our proposed technique leverages the scalability advantages of grammar-based coverage-guided fuzzing while avoiding the constraints of the fuzz-then-reduce approach. The *key idea*

```
# (Ex. A) Single pass statement
pass

# (Ex. B) Simple assignment statement
a:object = 1

# (Ex. C) Function definition with return
def a():
    return
```

Figure 3.2: Three examples of ChocoPy programs saved during bonsai fuzzing, in a corpus produced by $\mathcal{F}_{1,1,1}$.

```
# (Ex. D) Class definition with attribute declaration
class a(object):
    a:int = 1
pass

# (Ex. E) Indexing into a string
("a")[0]

# (Ex. F) Less-than comparison on two integers
0 < 0

# (Ex. G) Equality comparison on two strings
"" == "a"

# (Ex. H) Function definition with two arguments
def a(b:str, a:int):
    pass
```

Figure 3.3: Four examples of ChocoPy programs saved during bonsai fuzzing by $\mathcal{F}_{2,1,1}$, $\mathcal{F}_{1,2,1}$, and $\mathcal{F}_{1,1,2}$.

in our approach is to grow a test corpus *bottom-up* by (1) using coverage-guided bounded grammar fuzzing (CBGF) to generate small inputs *by construction* and (2) iteratively increasing the input size, inspired by iterative-deepening-based search algorithms [62]. We call our approach *bonsai fuzzing*.

Figs. 3.2, 3.3, and 3.4 show a total of eleven ChocoPy programs saved during various rounds of bonsai fuzzing (comments added manually). These programs are concise and the language features they exercise can be easily discerned. In our opinion, they look almost like hand-written test cases that are precisely designed for testing specific features of the ChocoPy language semantics. However, they were generated completely automatically and without knowledge of any typing rules. We next build a series of concepts leading up to a description of the bonsai fuzzing algorithm.

Bounded Grammar Fuzzers

We start by considering an input generator that can randomly sample inputs of a bounded size, where the bounds are based on the definition of an input language's grammar. We

```
# (Ex. I) Nested list expression
[[1], [None]]

# (Ex. J) Object construction and attribute assignment
class a(object):
    a:int = 1
(a()).a = 1

# (Ex. K) Nested functions
def a():
    def b():
        pass
    return
```

Figure 3.4: Example ChocoPy programs saved in the bonsai fuzzing corpus of $\mathcal{F}_{3,3,3}$.

can observe three properties of a ChocoPy program to get an idea of how we might bound the input space.

1. *idents*: the number of new unique identifiers (variable names, function names, class names) excluding predefined identifiers (e.g. `int`).
2. *items*: the maximum number of elements in a linear group. This can correspond to the maximum number of statements in a block, arguments in a function definition, arguments in a list expression, etc.
3. *depth*: the maximum number of times an expression, statement, or function definition is nested.

We can bound the input space if we restrict the maximum value of *idents*, *items*, and *depth* for any generated ChocoPy program. We now generalize this to any language.

Consider a specification for the syntax of an input language in the form of a context-free grammar $\mathcal{G}$. We consider definitions in an extended Backus–Naur form [63], where $\mathcal{G}$ consists of a set of terminals $\mathcal{T}$, a set of non-terminals $\mathcal{N}$, a start symbol $S \in \mathcal{N}$, and a set of production rules of the form:

$$A \longrightarrow \alpha, \quad \text{where } A \in \mathcal{N} \text{ and } \alpha = a_1 a_2 \ldots$$

The right-hand side of production rules α are a sequence of zero or more *symbols* which are defined recursively as follows: a *symbol* is either a terminal in $\mathcal{T}$, a non-terminal in $\mathcal{N}$ or of the form $[b]^*$, where b is a symbol. The Kleene-star in the final form has the usual meaning and enables non-recursive definitions of linear repeating sequences, e.g. list of statements or arguments to a function call. We also consider a special class of terminals $\tau \subseteq \mathcal{T}$ whose concrete values are user-defined (e.g. identifiers) instead of predefined (e.g. '+' or '`while`'). In the ChocoPy grammar, we have $\tau = \{ID, IDSTRING\}$.

Now consider the set of programs $\mathcal{P} = \{p : p \sim \mathcal{G}\}$. Each program p has a corresponding derivation tree t from $\mathcal{G}$. We are interested in bounding the following properties:

1. *idents*(p): The maximum number of distinct values for any terminal in τ (e.g. number of distinct identifiers) observed across the entire tree t.
2. *items*(p): The maximum number of repetitions in any expansion of a Kleene-star (e.g. number of statements in a block) when generating t.

3. *depth*(p): The maximum number of expansions of the same non-terminal (e.g. `expr`) in any path from the root to any leaf node in t.

We can then define a smaller input space $\mathcal{P}_{m,n,d}$, where

$$\mathcal{P}_{m,n,d} = \left\{ p \quad : \quad \begin{array}{l} p \in \mathcal{P} \\ \text{idents}(p) \leq m, \\ \text{items}(p) \leq n, \\ \text{depth}(p) \leq d \end{array} \right\}$$

The $\mathcal{F}$ notation

We now define some short-hand notation that will be useful when describing our proposed bonsai fuzzing technique. Let $\mathcal{F}^{\mathcal{G},p}_{m,n,d}$ denote a coverage-guided bounded grammar fuzzer (CBGF) parameterized by grammar $\mathcal{G}$, test program p, size bounds m, n, and d. $\mathcal{F}^{\mathcal{G},p}_{m,n,d}$ is a function that accepts an ordered set of inputs and returns a corpus of the same type. Since the grammar and target program are usually fixed in a given application, we will omit the superscripts hereon; therefore, $\mathcal{F}_{m,n,d}$ is a CBGF of size bounds (m, n, d).

Bonsai Fuzzing

Our novel solution is to build a concise test corpus from the bottom up by using a set of CBGFs with gradually increasing size bounds. The intuition is that the smaller CBGFs would initially build a corpus of tiny test corpus covering simple features, and larger CBGFs can build on the smaller programs to generate more complex test cases that achieve better coverage. By inceasing the size bounds gradually at each step, we expect the complex test cases in later stages to be structural mutations of test cases discovered in earlier stages; thus, we hope to simultaneously achieve validity, conciseness, and comprehensiveness. We now define a way to iteratively increment the size of a CBGF, which allows us to create a formal procedure for this approach.

Given upper bounds M, N, and D, we can consider the set of CBGFs

$$\mathcal{C}_{M,N,D} = \left\{ \mathcal{F}_{m,n,d} \quad : \quad \begin{array}{l} 1 \leq m \leq M \\ 1 \leq n \leq N \\ 1 \leq d \leq D \end{array} \right\}$$

With upper bounds $(3, 3, 3)$, we would have 27 different CBGFs in set $\mathcal{C}_{3,3,3}$.

We define a *partial order* $\leq$ over $\mathcal{C}_{M,N,D}$ as follows:

$$\mathcal{F}_{m,n,d} \leq \mathcal{F}_{m',n',d'} \quad \iff \quad m \leq m', n \leq n', d \leq d'$$

Consequently, $\mathcal{F}_{m,n,d} < \mathcal{F}_{m',n',d'}$ iff $\mathcal{F}_{m,n,d} \leq \mathcal{F}_{m',n',d'}$ and $\mathcal{F}_{m,n,d} \neq \mathcal{F}_{m',n',d'}$.

This ordering suggests that $\mathcal{C}_{M,N,D}$ is a lattice with $\mathcal{F}_{1,1,1}$ being the bottom element (denoted $\mathcal{F}_\perp$) and $\mathcal{F}_{M,N,D}$ being the top element (denoted $\mathcal{F}^\top$). Fig. 3.1 visualizes the lattice for $\mathcal{C}_{2,2,2}$, where the partial order corresponds to graph reachability. In this example, $\mathcal{F}^\top = \mathcal{F}_{2,2,2}$.

Additionally, we define the terms *successor* and *predecessor* with their usual meaning:

Algorithm 2 Bonsai fuzzing algorithm

```
1: procedure BonsaiFuzzing
2:     F ← F_⊥
3:     seeds ← [random()]                                  ▷ Single random seed
4:     corpus(F_⊥) = F(seeds)                              ▷ Run CBGF to generate corpus
5:     worklist ← successors(F)
6:     while worklist is not empty do
7:         for each F in worklist do                       ▷ Parallelizable
8:             P ← predecessors(F)
9:             seeds ← SortBySize ( ⋃_{F_p ∈ P} corpus(F_p) )
10:            corpus(F) ← F(seeds)                        ▷ Run CBGF
11:        worklist ← ⋃_{F_s ∈ worklist} successors(F_s)
12:    return corpus(F^⊤)
```

1. $\mathcal{F}_s$ is a *successor* of $\mathcal{F}$ if $\mathcal{F} < \mathcal{F}_s$ and there exists no CBGF $\mathcal{F}'_s$ such that $\mathcal{F} < \mathcal{F}'_s < \mathcal{F}_s$.
2. $\mathcal{F}_p$ is a *predecessor* of $\mathcal{F}$ if $\mathcal{F}_p < \mathcal{F}$ and there exists no CBGF $\mathcal{F}'_p$ such that $\mathcal{F}_p < \mathcal{F}'_p < \mathcal{F}$.

For example, $\mathcal{F}_{2,1,1}$ is a successor of $\mathcal{F}_{1,1,1}$, whereas $\mathcal{F}_{2,2,1}$ is not. Conversely, $\mathcal{F}_{1,1,1}$ is a predecessor of $\mathcal{F}_{2,1,1}$ but not a predecessor of $\mathcal{F}_{2,2,1}$. In Fig. 3.1, every node has incoming edges from its predecessors and outgoing edges to its successors. Naturally, $\mathit{predecessors}(\mathcal{F}_\perp) = \mathit{successors}(\mathcal{F}^\top) = \{\}$.

We now formally define *bonsai fuzzing* as a procedure that begins with the smallest configuration $\mathcal{F}_\perp$ and iteratively increases the size until a given upper bound is reached.

Algorithm 2 describes the procedure for bonsai fuzzing. Variable $\mathcal{F}$ is initialized to the smallest CBGF $\mathcal{F}_\perp$. Initially, we have no seeds. We thus start by running the CBGF $\mathcal{F}_\perp$ with one random seed input (similar to SLF [64]) to produce $\mathit{corpus}(\mathcal{F}_\perp)$. Then, a *worklist* is populated with the $\mathit{successors}(\mathcal{F})$. For each unprocessed element in the worklist—that is, each unexecuted fuzzer—we prepare its *seeds* by taking a union of all test cases in the *corpus* generated by each of its predecessors. The seeds are also sorted by size in ascending order (so that Algorithm 1 encounters smaller inputs to mutate first). We then run $\mathcal{F}$, save its resulting *corpus*, and repeat this process. Eventually, we reach the point where $\mathcal{F} = \mathcal{F}^\top$ and there are no more successors. The final corpus is the result of $\mathcal{F}^\top$.

Consider a sample run of bonsai fuzzing over the set $\mathcal{C}_{3,3,3}$. We start by running the CBGF $\mathcal{F}_\perp = \mathcal{F}_{1,1,1}$ with one randomly generated seed input. Fig. 3.2 shows three sample test cases saved in the resulting $\mathit{corpus}(\mathcal{F}_{1,1,1})$. We can see that the generated programs are small in size and test simple language features. These inputs will then be used as seeds in successor CBGFs: $\mathcal{F}_{2,1,1}, \mathcal{F}_{1,2,1}$, and $\mathcal{F}_{1,1,2}$. Fig. 3.3 lists some programs saved in the corresponding corpora of these fuzzers. We can now start to see programs with slightly complex features, such as class attributes, binary expressions, and functions with multiple parameters. We repeat the process until we reach $\mathcal{F}_{3,3,3}$, the top element of the lattice $\mathcal{C}_{3,3,3}$. Fig. 3.4 shows some example programs saved in the its corpus. More complex features such as nested list expressions and nested function definitions are demonstrated in these

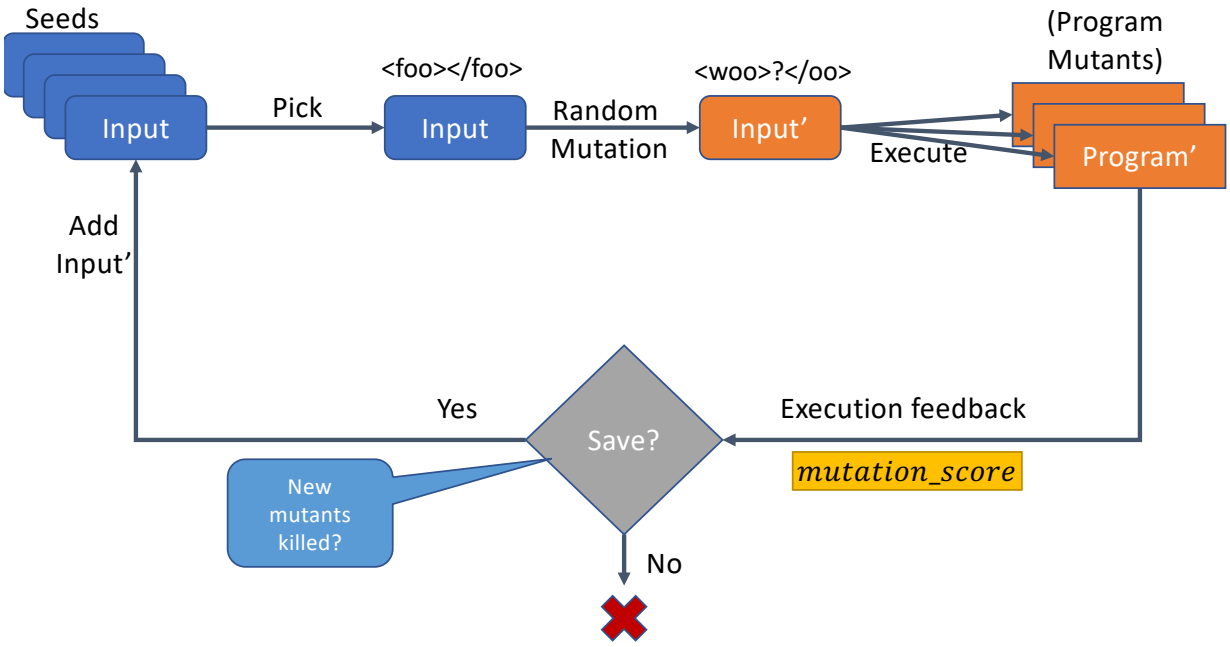


Figure 3.5: A mutation-analysis-guided fuzzing loop. Each fuzzer-generated input is run through a set of program mutants to compute a mutation score. Inputs are saved to the corpus if they improve mutation score.

generated programs. Note that some of these inputs may have been copied verbatim from its seeds, having been discovered by predecessors. The final corpus necessarily incorporates the corpora generated by all CBGFs in the lattice.

3.1.3 Evaluation Summary

We evaluate bonsai fuzzing by measuring its ability to generate a test corpus containing test cases that are *concise*, *comprehensive*, *semantically valid*, and (where applicable) able to detect faults. We compare bonsai fuzzing to a baseline of CBGF (that is; Zest [19] with a grammar-based input generator) post-processed with minimization techniques. The baseline is thus the conventional "fuzz-then-reduce" approach. We run our evaluation on two test targets: our primary application ChocoPy and a secondary target Google Closure Compiler to ensure that our solution is not biased towards a particular implementation or input language.

Experimental results show that Bonsai Fuzzing can generate test corpora having inputs that are 16–45% smaller in size on average as compared to a fuzz-then-reduce approach, while achieving approximately the same code coverage and fault-detection capability. For the full evaluation details, refer to Appendix A and the paper [27].

3.2 Guiding Greybox Fuzzing with Mutation Testing

Another valuable quality of synthesized test inputs is their ability to detect faults in the program under test. However, as seen in Algorithm 1, test inputs are only saved if they increase the code coverage achieved during the fuzzing campaign. Now, the technique of *mutation testing* [65], which evaluates the ability of tests to catch artificially injected bugs (a.k.a. *mutation analysis*), has shown promise as an adequacy criteria for improving test-suite effectiveness [46, 66, 67]. A test is said to *kill* a program mutant if it fails when executed on the mutant, whereas mutants that fail no tests are said to *survive*. A goal of mutation testing is to produce a test corpus that has a high *mutation score*, defined as the fraction of all mutants that are killed by the test suite. A natural question thus arises: *can we use mutation scores to guide the fuzzer*?

We investigate incorporating mutation analysis in the fuzzing loop. The idea is as follows (see Fig. 3.5): after a new input is synthesized by a fuzzer via random mutation of a previously saved input, it is evaluated by executing a *set of mutants of* the program under test. If the new input *kills* any previously surviving program mutant, then it is added to the corpus. In this process, we distinguish between *input mutations* (e.g., randomly setting input bits or fields to zero) and *program mutations* (e.g., replacing the expression `a+b` with `a-b` in the target's source code). Our Java-based implementation, called *Mu2*—for *Mutation-Based Greybox Fuzzing + Mutation Testing*—incorporates program mutations from the popular PIT toolkit [68] into a custom guidance in the JQF [18] greybox fuzzing framework. Mu2 is open source and available at: `https://github.com/cmu-pasta/mu2`.

The rest of this chapter details two main aspects of Mu2's design. First, with a conventional fuzzing oracle that only identifies program crashes or aborts, many inputs will be discarded for not killing any mutant even though they exercise interesting program functionality. For mutation testing to be useful, we need a stronger test oracle. Mu2 incorporates the idea of *differential mutation testing*, which validates the *output* of program execution. Second, evaluating each fuzzer-generated input on the set of all program mutants is prohibitively expensive, thereby reducing fuzzing throughput. Mu2 prunes the set of mutants to run at each fuzzing iteration using dynamic analysis of the original program's execution in two ways: (a) *sound* optimizations that prune mutants which cannot be killed by a given input, and (b) *aggressive* optimizations that select only a bounded subset of candidate mutants to run in each iteration.

3.2.1 Background

Mutation Testing

Mutation testing (also known as a *mutation analysis*) is a methodology for assessing the adequacy of a set of tests using artificially injected "bugs", or *program mutants* [65, 69]. In assessing test adequacy [70], we are given a program P and a suite of passing tests X. The goal is to evaluate the quality of X by computing a score that grows monotonically [71] with additions to the set X. *Code coverage* is an example of a test adequacy criteria.

In mutation testing, a set of program mutants, say $\textsc{Mutants}(P)$, is first generated.

Each mutant $P' \in \text{MUTANTS}(P)$ is a program that differs from P in a very small way. Most commonly, mutations are replacements of program expressions. For example, an expression `a+b` at line 42 in P may be replaced with the expression `a-b`. We can use the notation $\langle P, \texttt{a+b}, \texttt{a-b}, 42\rangle$ to refer to this mutation. For purposes of this chapter, we use the notation:

$$P' = \langle P, e, e', n\rangle$$

to refer to a program mutant P' as a modification of program P where expression e is replaced with e' at program location n. The main idea is that a program mutation simulates a simple programmer error or an artificially injected "bug".

The test suite X is then run on each mutant P'. If some test $x \in X$ fails when run on mutant P', then the mutant P' is said to be *killed*, which we denote as $\text{KILLS}(P', x)$. If the test suite X still passes, then the mutant P' is said to *survive*.

Ideally, we want our tests to be able to identify "bugs" and so we hope to have tests that fail on each mutant P'. So, the adequacy of test suite X is defined by the *mutation score*, which is computed as the fraction of mutants killed: $\frac{|\{P' \in \text{MUTANTS}(P) | \exists x \in X : \text{KILLS}(P', x)\}|}{|\text{MUTANTS}(P)|}$.

In general, a mutation score of 100% is rarely achievable because some mutants P' may actually be *equivalent* to P—that is, $\forall x : P(x) = P'(x)$. Similar to code coverage—where 100% may not be achievable due to unreachable code—the best use of the adequacy score is as a relative measurement rather than an absolute one.

One of the most mature and actively developed mutation testing frameworks, PIT [68], targets Java programs by mutating JVM bytecode. PIT's default mutation operators include:

- Conditional boundary mutator (e.g., `a<b` to `a<=b`))
- Increments mutator (e.g., `a++` to `a--`)
- Invert negatives (e.g., `-a` to `a`)
- Math mutators (e.g., `a+b` to `a*b`)
- Negate conditionals (e.g., `a==b` to `a!=b`)
- Return values mutator (e.g., replacing operands in `return` statements with a constant such as `null`, `0`, `1`, `false`, etc. depending on type).

3.2.2 Problem Statement and Scope

For the rest of this chapter, we focus on the following problem:

> *Can we use mutation analysis to guide greybox fuzzing in order to synthesize a test-input corpus with high mutation score?*

Gopinath et al. [72] have identified and discussed several challenges of combining mutation analysis with fuzzing, including (1) the strength of oracles used by the fuzzer, (2) the computational expense of performing mutation analysis, (3) dealing with equivalent mutants, and (4) the lack of mutation testing frameworks that focus on fuzzers. We directly address such challenges in this chapter. Oracles are discussed in Section 3.2.4 and performance concerns in Section 3.2.5. Our evaluation is not dependent on identifying

Algorithm 3 Mutation-analysis-guided fuzzing. Changes to Alg. 1 are highlighted.

```
1: procedure MU2 (Program P, Set of inputs seeds, Budget T)
2:     corpus ← seeds
3:     repeat
4:         x ← PICKINPUT(corpus)
5:         x′ ← MUTATEINPUT(x)
6:         if COVERAGE(P, x′) ⊈ ⋃_{x∈corpus} COVERAGE(P, x) then
7:             corpus ← corpus ∪ x′
8:         for all P′ ∈ PROGMUTS2RUN(P, corpus, x′) do
9:             if KILLS(P′, x′) ∧ P′ ∉ KILLED(P, corpus) then
10:                corpus ← corpus ∪ x′
11:    until budget T
12:    return corpus
13: function KILLED(Program P, Set of inputs X)
14:    return {P′ | P′ ∈ MUTANTS(P) ∧ ∃x ∈ X : KILLS(P′, x)}
```

equivalent mutants, since we only care about *relative* mutation scores (higher=better) rather than the exact number of mutants killed by a test-input corpus. Section 3.2.5 deals with reducing the performance impact of equivalent mutations.

Scope Since there is a vast amount of literature on the many variables involved in mutation analysis, as surveyed by Papadakis et al. [73], we restrict ourselves in this chapter to investigating only the aspects of *combining* mutation analysis with greybox fuzzing. In particular, we (1) work with the assumption that a high mutation score is a desirable property of a test-input corpus used for regression testing, referring the reader to several empirical studies examining the relationship between mutation scores and real faults [46, 66, 67, 74, 75, 76, 77], and (2) directly use the default set of mutation operators provided by PIT (ref. Section 3.2.1), which have been chosen based on several empirical studies of effectiveness, sufficiency, and to align with developer expectations [68, 78, 79, 80].

3.2.3 The Mu2 Framework

To address our problem statement, we present the mutation-analysis-guided greybox fuzzing technique in Algorithm 3. This is an extension of Alg. 1, with changes highlighted in grey. The key additions of this algorithm are in evaluating whether a fuzzer-generated input x' should be saved to the corpus. The function PROGMUTS2RUN (Line 8) returns a set of program mutants to evaluate with input x'. For now, assume it to return MUTANTS(P) as defined in Section 3.2.1, though we will refine this in Section 3.2.5. We then determine whether the input x' is the first input to kill some mutant P'. If P' is killed by x' and P' has not previously been killed by any input in the *corpus* (Lines 9 and 14), then we add x' to the corpus (Line 10). Broadly, this algorithm saves fuzzer-generated inputs if they increase either code coverage or mutation score. Additionally, inputs that increase mutation score are marked as *favored*, giving them more energy to be picked for fuzzing (Line 4). As

```
1  class Sort {
2    static int[] insertionSort(int[] arr) {
3      for (int j = 1; j < arr.length; j++) {
4        int key = arr[j], i = j-1;
5        while (i >= 0 &&       // P'2 changes `>=` to `>`
6               key < arr[i]) {
7          arr[i+1] = arr[i];  // P'3 sets RHS to `1`
8          i = i-1;            // P'4 removes `-1`
9        }
10       arr[i+1] = key;       // P'1 removes `+1`
11     } return arr;
12   }}
```

Figure 3.6: Java program that implements insertion sort, annotated with four sample program mutants.

before, the final corpus of fuzzer-generated inputs is returned as the result (Line 12).

We have implemented Algorithm 3 for fuzzing Java programs by integrating PIT [68] into JQF [18]. We call this system Mu2, since it combines *Mutation-based Greybox Fuzzing* with *Mutation Testing*.

We chose PIT and JQF because of their maturity, extensibility, and their common target platform. As described in Section 3.2.1, PIT is an actively developed mutation testing framework that operates on JVM bytecode. The JQF framework [18] was originally designed for *coverage-guided property-based testing*, which is a structure-aware variant of greybox fuzzing (ref. Chapter 2.2) and instruments JVM bytecode for collecting code coverage. JQF also has a highly extensible design for creating pluggable *guidances*, which supports rapid prototyping of new fuzzing algorithms [19, 27, 28, 81, 82, 83, 84].

In Mu2, MUTANTS(P) includes all of PIT's default expression mutation operators (ref. Sections 3.2.1 and 3.2.2). For heuristics such as PICKINPUT and MUTATEINPUT, Mu2 reuses the logic and code from Zest [19], which we also use as a baseline for evaluation.

3.2.4 Oracle: Differential Mutation Testing

One challenge of mutation-analysis-guided fuzzing is determining whether a program mutant is killed by a particular input. This corresponds to the KILLS function invoked in line 9 of Algorithm 3.

In mutation testing, a program mutant P' is considered killed if any test in the test suite fails. The logic that determines whether a test passes or fails is known as the *test oracle*.

Greybox fuzzing generally relies on *implicit oracles*, which aim to detect anomalous behavior such as crashes or uncaught exceptions, or *property tests*, which assert a predicate over the output of some computation. For example, consider the insertion sort method defined in Figure 3.6 and the following test method, which is written in the property-testing style using JQF's `@Fuzz` annotation:

```
1  @Fuzz // Inputs generated using greybox fuzzing
```

```
1 @Diff // inputs generated by Mu2
2 int[] runInsertionSort(int[] input) {
3     return Sort.insertionSort(input);
4 }
5 @Compare // outputs compared with mutant
6 boolean checkEq(int[] outOrig, int[] outMut) {
7     return Arrays.equals(outOrig, outMut);
8 }
```

Figure 3.7: A Mu2 differential mutation test driver and comparison method for the `insertionSort` method (Fig. 3.6).

```
2 void fuzzInsertionSort(int[] input) {
3   assert(isSorted(Sort.insertionSort(input)));
4 }
```

For Mu2, we *could* use this property test as an oracle. Consider the following examples, using the notation introduced in Section 3.2.1: executing mutant $P_1' = \langle \texttt{Sort}, \texttt{i+1}, \texttt{i}, 10 \rangle$ with input array $x =$ `[3, 2, 1]` would result in an uncaught `IndexOutOfBoundsException` (-1) on line 10, triggering a failure via the *implicit oracle*. Additionally, executing $P_2' = \langle \texttt{Sort}, \texttt{i>=0}, \texttt{i>0}, 5 \rangle$ with x would result in an assertion failure in the property test because the result of $P_2'(x)$ would be the array `[3, 1, 2]`, which is not sorted. So, both mutants P_1' and P_2' would get killed by the fuzzer if it discovers such an input.

Unfortunately, the property test is not a *complete* oracle in that it does not fully specify the expected behavior of the sort function. Consider a third mutant $P_3' = \langle \texttt{Sort}, \texttt{arr[i]}, 1, 7 \rangle$, which assigns a constant to every array element at line 7. This is clearly a bug in insertion sort, yet the output is always sorted. For example, when $x =$ `[3, 2, 1]`, the result of $P_3'(x)$ is `[1, 1, 1]`. Such a mutant would incorrectly survive on any input the fuzzer generates.

Writing a complete oracle for testing insertion sort is possible, but quite cumbersome. In general, this is a hard problem [85]. For many applications, a complete oracle would need to be as complex (or in some cases exactly the same) as the original program itself.

In Mu2, we use the well-known concept of *differential testing* to define our oracle. In differential testing [86, 87], different implementations of a program that are expected to satisfy the same specification are executed on a single input, and their results are compared to identify discrepancies. In Mu2, our different "implementations" are the original program and program mutants; any discrepancy between the original program output and a mutant's output leads to that mutant being *killed*.

To support the comparison of outputs, we create a *differential mutation testing* framework. This allows for (1) output values to be returned from a fuzzing driver (as opposed to the `void` returns used by conventional property testing methods) and (2) a user-defined comparison function for specifying how outputs from the original program and a program mutant should be compared. An example of differential mutation testing methods in our framework is shown in Figure 3.7. The `@Diff` method `runInsertionSort` returns an output value of type `int[]`. The user-defined comparison method `checkEq` simply

Table 3.1: Geometric mean of speedups achieved by the execution and infection based optimizations (Alg. 4, Line 4) from the PIE model [89] across 10 repetitions of 3 hours each[1].

Mean Speedup From:	Execution Opt.	Infection Opt.
ChocoPy	3.6×	7.4×
Gson	18.2×	23.2×
Jackson	60.5×	77.4×
Tomcat	13.6×	23.8×

determines if the output arrays are equal. If unspecified, the `@Compare` function defaults to the `java.lang.Objects.equals()` method. Our interface is general enough to support complex differential testing oracles such as the ones used in CSmith [88].

With differential mutation testing, we are able to kill mutants such as P'_3 described above with an input like `[3, 2, 1]`, where the output of `insertionSort` on the original program—`[1, 2, 3]`—is not equal to the output of the mutant—`[1, 1, 1]`.

We can now precisely define KILLS(P', x) which was referenced in Algorithm 3. Given a mutant $P' = \langle P, e, e', n\rangle$ and an input x, KILLS(P', x) returns true iff:

1. $P(x) = y \wedge P'(x) = y' \wedge \neg$COMPARE$(y, y')$, where COMPARE is the user-defined `@Compare` method (e.g., `checkEq` in Figure 3.7) or `Object.equals()` if one is not defined; or
2. $P(x) = y$ but executing $P'(x)$ results in an uncaught run-time exception being thrown; or
3. Executing $P'(x)$ takes longer than a predefined `TIMEOUT`.

The timeout is required for killing mutants such as $P'_4 = \langle$`Sort, i-1, i,` $8\rangle$, which effectively removes the decrement of `i`, leading to an infinite loop on the input `[3, 1, 2]`.

3.2.5 Performance

The biggest challenge with incorporating mutation testing inside a fuzzing loop is performance. Given its need to execute many mutants on each iteration, mutation testing is in general a very expensive technique [73], so scaling Mu2 to real-world software is a non-trivial task. Two aspects of improving scalability are: (1) reducing the average time required to execute each program mutant, and (2) reducing the number of program mutants that must be evaluated at each iteration of the fuzzing loop.

Improving Performance of Mutant Execution

When running a mutation testing tool such as PIT [68], each mutant and test is run in a different JVM. For general mutation testing, this is ideal because it simplifies managing multiple copies of the same program (sans mutations), and prevents global state changes

[1]The Closure Compiler benchmark was too large to run without the execution and infection optimizations, so we did not include the speedups in this table.

Algorithm 4 Logic for determining which mutants to run in a given iteration of the fuzzing loop (Alg. 3)

1: **function** ProgMuts2Run(Program P, Old inputs *corpus*, New input x)
2: $\quad$ *surviving* $\leftarrow$ mutants(P) $\setminus$ killed(P, *corpus*)
3: $\quad$ *killable* $\leftarrow \{P' = \langle P, e, e', n\rangle \mid (P' \in \textit{surviving}) \wedge$
4: $\quad\quad\quad (n \in \text{coverage}(P, x)) \wedge (\text{infect}(P, e, e', x))\}$
5: $\quad$ **if** *AGGRESSIVE_OPT* is configured **then**
6: $\quad\quad$ **return** filter(*killable*, *AGGRESSIVE_OPT*)
7: $\quad$ **return** *killable*

from one program mutant affecting the state of another program mutant. However, this is not necessary for Mu2. For in-process fuzzing, test driver methods are expected to be self-contained and not depend on global state. Like JQF and Zest, Mu2 is designed to work in a single JVM.

Mu2 thus adopts a different strategy than PIT and takes advantage of the Java class-loader mechanism to load and run program mutants within the same JVM, essentially by having copies of the entire class hierarchy (one per mutant) in memory at the same time. First, a `CoverageClassLoader` (CCL) is responsible for loading the original target program P and collecting code coverage using on-the-fly instrumentation. For differential testing, the CCL-loaded classes compute the ground-truth outcome $P(x)$. Second, a family of `MutationClassLoaders` (MCL) are used to load program mutants; one MCL per mutant $P' = \langle P, e, e', n\rangle$. When a mutant test program is loaded by the MCL, it performs on-the-fly bytecode instrumentation exactly at location n, replacing expression e with e' and loading the rest of the program without changing semantics. The MCL adds instrumentation at backward jumps (i.e., loops) in order to detect timeouts and exit test execution cleanly if necessary.

Further, assuming that fuzz tests do not affect global state, Mu2 loads only one copy of each library class (defined as classes outside a specified package identifying the target application as long as they and their transitive dependencies do not reference any application class) using a common `SharedClassLoader`—this dramatically reduces memory pressure when mutating large programs.

To validate our design, we ran an informal preliminary experiment of performing mutation analysis with PIT and Mu2's in-memory set-up on a fixed corpus of seed inputs for the Google Closure Compiler [90]. In the steady state (after the first 8 inputs), Mu2's in-memory analysis runs with a 9.6× speed-up over PIT.

Reducing the Number of Mutants to Run in the Fuzzing Loop

For each *trial*—i.e., iteration of the fuzzing loop—(1) the input must be executed once by the original program and (2) the input must be executed by each mutant. Thus, we can model the time required to execute each trial as the following:

$$\textit{trialTime} = \text{time}_{\text{orig}} + M * \text{avgTime}_{\text{mut}} \tag{3.1}$$

where $M = |\text{ProgMuts2Run}(P, \textit{corpus}, x)|$ as per Algorithm 3.

Observe that the time per trial scales linearly with M. We can improve the fuzzing throughput (i.e., the number of trials executed per unit time) directly by reducing M. From Algorithm 3 (Lines 9–10), we can see that we only care about executing a program mutant if it will help us determine if a given input is the first input to kill it. We can therefore reduce M by dynamically pruning mutants whose execution will necessarily lead to Line 9 evaluating to *false*.

So, we begin by applying the following conditions for a given $P' = \langle P, e, e', n \rangle$, which are shown in Algorithm 4, lines 2–4:

1. If $P' \in \text{KILLED}(P, \textit{corpus})$, then P' does not need to be executed for any future inputs.
2. If the program mutant P' applies a mutation to a program location n, but n is *not covered* when executing the original program on x, then P' cannot be killed by x. This corresponds to *execution*-based pruning in the PIE model [89].
3. If we can guarantee that all dynamic evaluations of e during the execution of P on x are equivalent to the corresponding evaluations of mutated expression e', then P' cannot be killed by x. This corresponds to *infection*-based pruning in the PIE model [89], which we implemented as a dynamic analysis of the execution of the original program $P(x)$.

The last two strategies from the PIE model require additional overhead when executing x: (1) the execution-based pruning depends on coverage instrumentation, and (2) infection-based pruning requires evaluating and comparing the mutation expression e each time that it is executed by x. Referring to Equation 3.1, the optimization results in a trade-off for *trialTime* due to the increase in $\text{time}_{\text{orig}}$ and decrease in the number of mutants to run M. However, we find this is quite beneficial overall. Table 3.1 shows the results of preliminary experiments on 4 benchmarks included in our evaluations to validate these optimizations; clearly, they improve performance significantly.

We note that all the pruning methods mentioned above are *sound* optimizations: a mutant is pruned only if it is *guaranteed* to survive when executed. Effectively, we are pruning mutants that are *equivalent modulo inputs* [91].

Aggressive Mutant Selection Optimizations

While the execution and infection optimizations significantly improve the overall throughput of Mu2, the M factor in Equation 3.1 still grows linearly with the size of the program (more code = more mutants). We can be aggressive about reducing M by attempting to bound it by a constant k, at the risk of potentially missing out on analyzing some mutants that could have been killed by a given input. We call these *aggressive optimizations*. We use the function FILTER in Algorithm 4 (Line 6) to optionally apply a selection strategy [92, 93] that returns a bounded subset of the *killable* mutants. We have implemented two types of filters in Mu2:

1. k-Random Mutant Filter: For each generated input, k mutants are randomly sampled from the *killable* set in Alg. 4.
2. k-Least-Executed Mutant Filter: For each generated input, the *killable* mutants are sorted by the number of times they have been executed on previous inputs. The first

k mutants are then selected. The goal is to prioritize executing mutants that have not been tested as frequently during the fuzzing fuzzing campaign. This is a novel reduction strategy designed specifically for the fuzzing loop.

Section 6.4 evaluates the impact of these aggressive optimizations.

3.2.6 Evaluation Summary

We evaluate Mu2 on five real-world Java targets using state-of-the-art greybox fuzzer Zest [19], which is also built on top of the JQF framework, as a baseline. We also empirically evaluate 7 variants of Mu2 employing different strategies for improving performance. Our combined evaluation represents 21,600 CPU-hours (2.5 CPU-years) of fuzzing campaigns.

In summary, our results indicate: (1) an optimized version of Mu2 has an overall improvement of up to 20% in mutation scores across five benchmarks (5% increase on average); (2) mutation-analysis feedback generates test-input corpora with higher reliability of killing *nontrivial* mutants compared to coverage-only feedback; (3) the differential testing oracle is significantly valuable to Mu2, detecting 30% more mutants on average than a conventional fuzzing oracle. For the full evaluation details, refer to Appendix B and the paper [94].

3.3 Status

Growing a Test Corpus with Bonsai Fuzzing [27] has been published at ICSE 2021 and Guiding Greybox Fuzzing with Mutation Testing [94] has been published at ISSTA 2023.

3.4 Summary

In this chapter, we focused on two different methods of improving the quality of test inputs produced by coverage-guided fuzzing: (1) improving conciseness by iteratively growing parameters to bound the size of our generators, and (2) improving artificial fault-finding capability by incorporating mutation analysis into the greybox fuzzing loop. Our findings show that we are able to optimizing aspects of input quality beyond code coverage as a part of the test-input generation process and successfully improve the overall quality of inputs.

Chapter 4

Towards Synthesis of Property-Based Tests with Generative AI

4.1 Introduction

Despite its proven results and impact in the research community, PBT is not as widely adopted by open source and industry software developers. Using the Open Source Insights [95] dependency dataset, we find that only 222 out of 180,000 PyPI packages list the Python PBT library *Hypothesis* as a dependency, despite it being a very popular project (6.7k+ stars on GitHub). Goldstein et al. [96] conducted a series of interviews and detail a set of challenges faced by professional developers when attempting to use PBT in their software. Developers reported difficulties in (1) writing random data generators for inputs and (2) articulating and implementing properties that would meaningfully test their code. Furthermore, Goldstein et al. describe the "critical mass problem" that PBT is still relatively unknown and unpopular among the software industry.

While developers have been reticent to adopt PBT, the practice of *documenting code* is widespread. Documentation for library API methods is fairly common, especially for languages like Python, and contains valuable information about input parameters and properties of the output. An truncated version of the documentation for the `numpy.cumsum` API method can be seen in Figure 4.2.

The use of pre-trained large language models (LLMs) for code generation has become increasingly popular [97, 98, 99]. LLMs have been effective at translating natural language specifications and instructions to concrete code [100, 101]. Additionally, LLMs have shown potential to improve existing automated unit test generation techniques [102], and even generate unit tests from scratch [103, 104]. In this chapter, we seek to answer the following research question:

RQ: *Can we synthesize high-quality property-based tests using large language models?*

We believe that the *documentation* of an API method can assist the LLM in producing logic to generate random inputs for that method and deriving meaningful properties of the result to check.

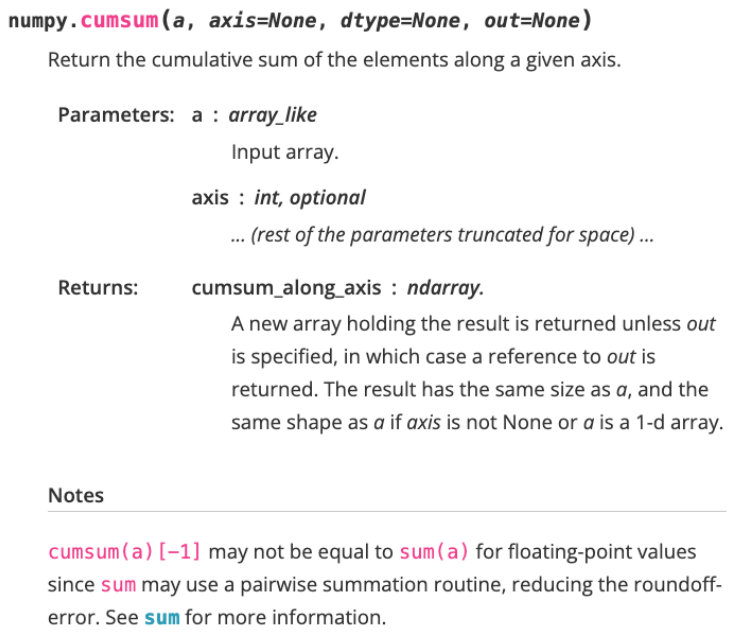
While LLMs have shown effectiveness in synthesizing unit tests [105] and fuzz har-

```python
from hypothesis import given, strategies as st
import numpy as np

# Summary: Generate random input parameters for numpy.cumsum and test properties
@given(st.data())
def test_numpy_cumsum(data):
    # Generating a list with varyin length and integer elements
    a = data.draw(st.lists(st.integers(min_value=-10, max_value=10), min_size=0, max_size=10))

    # Generate random axis
    axis = data.draw(st.one_of(st.none(), st.integers(min_value=0, max_value=a.ndim-1)))

    # Call numpy.cumsum with generated input
    cumsum_result = np.cumsum(a, axis=axis)

    # Test property: output shape should be the same as input shape if axis is not None or
    # input is 1-d array
    if axis is not None or a.ndim == 1:
        assert cumsum_result.shape == a.shape

    # Test property: output size should be the same as input size
    assert cumsum_result.size == a.size

    # Test property: cumsum(a)[-1] should be approximately equal to sum(a) for
    # non floating-point values
    if not np.issubdtype(a.dtype, np.floating):
        np.testing.assert_almost_equal(cumsum_result.flatten()[-1], np.sum(a))
# End program
```

Figure 4.1: A GPT-4 generated property based test for `numpy.cumsum`. The test first generates random integer arrays between size 1 and 20 and a random axis. Then, the API method under test `np.cumsum` is invoked on the randomly generated inputs. Finally, three properties are checked on the output array, all derived from information in the API documentation. All comments are also generated by GPT-4.

nesses [106, 107, 108], synthesizing property-based tests has unique challenges. In unit testing, the test oracles are usually simple equality assertions between expected and actual outputs; in PBT, a property assertion that holds *generally* for inputs must be synthesized in the test. Similarly, fuzz harnesses usually only contain *implicit* oracles that capture unexpected behaviors such as uncaught exceptions or crashes. Further, synthesizing property-based tests requires generating custom logic for random input generation over various types of inputs; unit tests do not require this logic since there is only one input, and fuzz harnesses primarily rely on inputs represented as raw byte streams.

An example of using LLMs for PBT can be seen in Figure 4.1, which displays an LLM-synthesized property-based test for the `numpy.cumsum` method when provided the documentation in Figure 4.2. First, the logic for generating random values for the input parameters *a* and *axis* is in lines 8–11. Then, the `cumsum` method is invoked on these arguments on line 14. Finally, lines 18–27 contain property assertions for the output `cumsum_result`. We specifically note that these properties assertions match natural language descriptions in the API documentation in Figure 4.2. The documentation specifies



numpy.cumsum(*a, axis=None, dtype=None, out=None*)

Return the cumulative sum of the elements along a given axis.

Parameters: **a** : *array_like*
Input array.

axis : *int, optional*
... (rest of the parameters truncated for space) ...

Returns: **cumsum_along_axis** : *ndarray.*
A new array holding the result is returned unless *out* is specified, in which case a reference to *out* is returned. The result has the same size as *a*, and the same shape as *a* if *axis* is not None or *a* is a 1-d array.

Notes

`cumsum(a)[-1]` may not be equal to `sum(a)` for floating-point values since `sum` may use a pairwise summation routine, reducing the roundoff-error. See **sum** for more information.

Figure 4.2: Truncated Numpy documentation for the `numpy.cumsum` API method. The documentation includes descriptions of properties about the result shape/size and additional information about the last element of the result.

that "result has the same size as a", which has a direct translation to the assertion in line 22. Similarly, the specification that result has "the same shape as a if *axis* is not None or a is a 1-d array" is checked conditionally as an assertion in lines 18–19. Finally, the property assertion shown in lines 26–27 checks that the last element of the result is equal to `np.sum(a)` if the array is not of `float` type. This assertion translates information from the notes section in the documentation into a useful property to check. While not a perfect property-based test, this example demonstrates the ability of LLMs to write logic for generating random inputs and derive meaningful property assertions from API documentation.

In this chapter, we study the use of state-of-the-art LLMs—Open AI's GPT-4, Anthropic's Claude-3-Opus, and Google's Gemini-1.5.-Pro—to automatically synthesize property-based tests. We propose single stage and two stage approaches for using API documentation to prompt the LLM to synthesize property-based tests. But how do we know whether an LLM-synthesized property-based test is good enough? We characterize several desirable properties of these tests and propose a methodology to evaluate the LLM's output: property-based tests must be *valid* (free of compile-time or run-time errors), *sound* (only assert properties that must be true), and ideally *complete* (i.e., actually check for properties that are mentioned in the API documentation). To measure completeness of property assertions, we introduce the notion of *property coverage*, which uses *mutation testing* to measure the ability of a property-based test to fail when the target API method behaves in a way that violates its documented property. We find LLM-based PBT generation a suitable approach for automating some of the tedious process of writing property-based

tests.

4.2 Background

4.2.1 Large Language Models

Pre-trained large language models (LLMs) [101, 109, 110, 111, 112, 113, 114] are a class of neural networks with a huge number of parameters, trained on large corpora of text data. These models are trained in an *autoregressive* manner—i.e., trained to predict the next token in a sequence—which allows them to be trained on a large volumes of unlabelled text. This extensive pre-training allows them to function as *one-shot* or *zero-shot* learners [110]. That is, these models can perform a variety of tasks when given only one example of the task, or a textual instruction of the tasks. The natural-language instructions, along with any additional input data, that are passed to the LLM are called the *prompt* [115]. The practice of creating prompts that allow the LLMs to effectively solve a target task is called *prompt engineering*.

Further, a number of LLMs have been trained extensively on code [97, 116, 117, 118, 119]. These models, as well as more general-purpose LLMs, have been used for numerous software engineering tasks, including program synthesis [98, 120], program repair [121, 122, 123], code explanation [124], and test generation [102, 103, 104]. These techniques use the LLMs out-of-the-box, getting them to accomplish the tasks via prompt engineering alone.

Like prior work, we use pre-trained language models and adapt them to our tasks only via prompt engineering. We discuss our methods of constructing these prompts in the following section.

4.3 The Proptest-AI Approach

4.3.1 Prompt Design

To synthesize a property-based test from the LLM, we first design prompts that include the API documentation and instructions to write a property-based test for the input method. We explored two prompting strategies: one to generate a single property-based test and one to generate a property-based testing suite.

Our high-level prompt templates contains:

1. System-level instructions stating that the LLM is an expert Python programmer.
2. The target API documentation, taken from the API website.
3. User-level task instructions to review the API documentation and perform a particular task.
4. The desired output format.

Based on this template, our first prompting strategy generates a single property-based test. The user-level task instructions begin with Chain of Thought [125] instructions to outline a generation strategy and a list of properties to test before synthesizing the

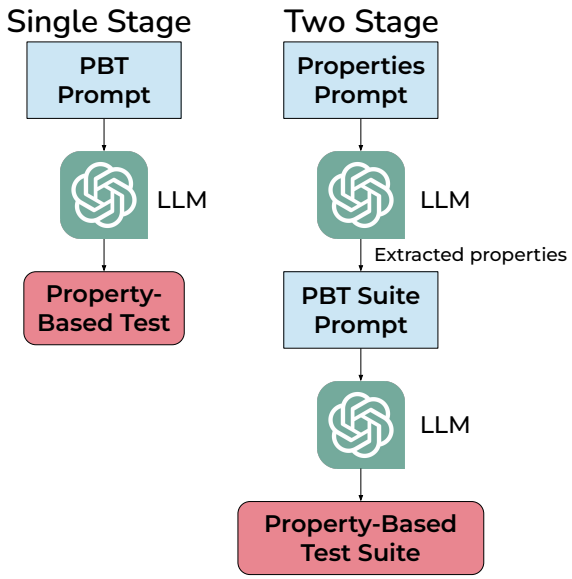


Figure 4.3: Two methods of generating property-based test using an LLM. The first is a single stage prompt of the LLM with zero-shot CoT instructions to (1) explain a generation strategy, (2) properties to test, and (3) generate a single property-based test. The second method instructs the LLM to extract a list of properties from the API docs and *continues the conversation*, instructing the LLM to write a test for each property.

property-based test. The prompt ends with instructions to generate a Hypothesis PBT and adds an output format including Hypothesis import statements, relevant API import statements, a boilerplate function signature, and an example Hypothesis PBT from the Hypothesis documentation. We refer to this prompt to generate the entire property-based test as the "PBT Prompt", as seen in Figure 4.3.

The second prompting strategy we explore has a two stage prompting method of (1) instructing the LLM to extract properties from the API (maximum five) and (2) instructing the LLM to create individual property-based test functions for each property. The first prompt contains user-level task instructions to extract five properties that hold for all outputs of the API method. The second prompt contains instructions to synthesize a PBT using Hypothesis for each of the properties generated by the LLM. Our full prompt templates are available in our data artifact.

4.3.2 Evaluation Methodology

Using our Proptest-AI methodology to prompt the LLM to synthesize property-based tests, how do we evaluate the quality of these generated PBTs? While the effectiveness of unit tests has been a well studied topic for decades [126, 127, 128, 129], this is not the case for property-based tests. One difficulty in conducting these types of evaluations for PBT is the lack of readily available property-based tests for software. Thankfully, LLMs can provide us a method of automatically generating property-based tests for which we can design an

```
1  from hypothesis import strategies as st
2  from datetime import timedelta
3
4  @given(days=st.integers(min_value=0), seconds=st.integers(min_value=0),
5      microseconds=st.integers(min_value=0),   milliseconds=st.integers(min_value=0),
6      minutes=st.integers(min_value=0),   hours=st.integers(min_value=0),
7      weeks=st.integers(min_value=0))
8  def test_timedelta_total_seconds(days, seconds, microseconds, milliseconds, minutes,
9      hours, weeks):
10     td = timedelta(days=days, seconds=seconds, microseconds=microseconds,
11         milliseconds=milliseconds, minutes=minutes, hours=hours, weeks=weeks)
12     (...)
```

Figure 4.4: An example invalid `datetime.timedelta` PBT produced by GPT-4. The `datetime.timedelta` constructor on line 11 raises an `OverflowError` when the absolute magnitude of days exceeds 1,000,000.

evaluation methodology.

We propose a PBT evaluation methodology and metrics focusing on (1) the validity of the tests, (2) the soundness of the tests, and (3) the *property coverage* (detailed in Section 4.3.2) of the tests. To motivate each of these metrics, we include examples of inaccurate LLM-synthesized PBTs and discuss the issues that impact each of these qualities. We then propose mitigation strategies for each of the inaccuracies that improve the quality of the property-based tests. All of our examples use the Hypothesis PBT library in Python.

Validity

One type of incorrect behavior in a property-based test is a validity issue in which a run-time error is encountered when the test is executed. The problem of hallucination is a well-known problem with using LLMs to generate code [3]. LLMs may generate plausible-looking but incorrect code that throws unexpected errors at runtime. This issue arises in property-based testing as well, in which the LLM may synthesize test cases that are syntactically valid but result in a runtime error during execution. Since the test is executed on multiple random inputs, it is also possible that only a percentage of randomly generated inputs result in runtime errors. An example is the generator in Figure 4.4 for `timedelta` objects in the Python `datetime` module. The generator function can produce values for the `timedelta` object constructor that may result in an `OverflowError` raised by the `datetime.timedelta` constructor when the magnitude of days exceeds 1,000,000. Thus, we determine a property-based test as *valid* if 100% of test function invocations do not result in any run-time errors, excluding property assertion errors as these relate to soundness.

Soundness

LLMs may also synthesize a property-based test that is *unsound*, i.e., there exists an input/output pair that violates a property assertion but is valid given the specification. Figure 4.5 provides an example of an LLM-synthesized property-based test for the

```
1 @given(generate_array())
2 def test_cumsum(a):
3     # Test that the shape of the output is the same as the input
4     out_shape = np.cumsum(a).shape
5     assert out_shape == a.shape
```

Figure 4.5: A Gemini-1.5-Pro generated property-based test containing an unsound property for `numpy.cumsum` on line 5.

```
1 import numpy as np
2
3 # Violate Non-Decreasing Sequence
4 def buggy_cumsum_1(a,
5     axis=None, dtype=None, out=None):
6     result = np.cumsum(a, axis=axis, dtype=dtype, out=out)
7     # Reverse the result to create decreasing elements
8     return result[::-1]
```

Figure 4.6: Example property mutant of `numpy.cumsum` produced by GPT-4. The mutant violates the property that the output of `numpy.cumsum` must be non-decreasing by reversing the result.

`numpy.cumsum` method that contains an unsound property on line 5. The `numpy.cumsum` documentation specifies that for a given input array a, the output should have "the same shape as a if axis is not None or a is a 1-d array". The synthesized property is unsound because it unconditionally checks whether the output and input shapes match. A randomly generated input of `array([[0]]` produces an assertion failure when this test is run since the input shape is `(1, 1)` and the output shape is `(1,)`; this is not an actual bug and the behavior of the `cumsum` method conforms with the API documentation.

If we encounter an assertion failure from a property check during the test, how do we know whether it is due to an unsound property or due to a bug in the API implementation? Given an assertion failure, we assume that the likelihood of the LLM generating an unsound property is higher than the likelihood of a bug. This is a weak assumption. We can capture the soundness of a property by measuring the frequency at which the assertion fails across multiple generated inputs. If an assertion fails on a significant percentage of the inputs, it is most likely an unsound property.

A property-based test is determined as *sound* if 100% of test function invocations do not result in assertion errors from the property checks. We specifically filter out any executions that result in any other runtime errors, as these are related to the validity of the property-based test rather than the soundness.

Property Coverage

While validity and soundness indicate that a property-based test runs without errors, an additional measurement is needed to evaluate effectiveness at testing specific properties. A property-based test may execute correctly with 100% validity and 100% soundness, yet inadequately test key properties due to having weak assertions. Our goal is to establish

a metric that answers the question: how effective is the property-based test at detecting whether an API method violates a specific property?

One related idea is *mutation testing*, which measures the ability of a test to detect bugs artificially bugs in the program under test [65]. First, mutant versions of the program are created with an artificially injected bug. If the test fails when executing the mutated programs, then the mutant is *killed*; otherwise, the mutant *survives*. Mutation testing provides a measurement related to the bug-finding capability of a particular test.

Mutation testing injects bugs by applying syntactic operators on the *source code*; these types of syntactic operators may not necessarily correspond to property violations of the API method. We would like to construct a mutant that performs a property-level mutation on the API method. More formally, given a method under test f, we would like to generate a mutant f' such that $\forall x \in \mathrm{X} : \neg q(f'(x))$.

Thus, we define a *property mutant* as a buggy version of the API method that returns an output violating a specific property. A property mutant f' of a method under test f is defined as follows:

$$f'(x) = mut(f(x))$$

where *mut* is a mutation operation on $f(x)$. The property mutant contains the same signature as the API method, invokes the original API method on the input, and finally performs a mutation operation on the output to violate the property. Generating this mutation operation requires semantic reasoning about how an output would violate a property.

LLMs have been used to perform bug injection and create program mutants [130, 131]. We design a specific prompt for the LLM to generate property mutants of a method under test for a given property. An example of a property mutant for `numpy.cumsum` is shown in Figure 4.6. This mutant can return an output that violates the property that the output must be non-decreasing by reversing the order of the output. In order to kill this mutant, the PBT must contain an assertion checking this property and contain logic to generate inputs of length greater than 1.

To check whether a property mutant is killed, we create a modified version of the property-based test. This modified version substitutes the call to the original API method with the call to the mutant API method. Assuming the original property-based test is valid and sound, we check whether there is an assertion failure on the output of the mutant API method in the modified property-based test.

We define the *property mutation score* of a property-based test as the percentage of killed property mutants of a given API method. The process of generating property mutants and measuring property mutation score is described as follows:

1. Prompt the LLM to extract a list of properties from the API documentation.
2. Prompt the LLM to generate a set of property mutants for each property.
3. If the property-based test is sound, execute the property-based test with a substituted call to the property mutant API method instead of the original API method.
4. Check whether the property assertions fail in the modified property-based test. If so, then the mutant is killed.

Finally, a property is considered *covered* if the PBT is able to kill any of the corresponding property mutants. The overall *property coverage* of a PBT is the percent of properties for which the PBT is able to kill property mutants.

4.4 Evaluation Summary

In our evaluation on 40 Python API methods and three state of the art language models, we find that our two-stage prompting approach with GPT-4 is able to produce valid and sound PBTs in 2.4 samples on average. We additionally find that our automated soundness metric is aligned with human judgment and that LLMs can be used to evaluate PBTs through the generation of property mutants. This approach demonstrates the potential for generative AI to automatically synthesizing and evaluating PBTs. For the full evaluation results, refer to Appendix C and the public preprint [132].

4.5 Status

This work is currently ongoing and the evaluation is being reworked. I am planning on rerunning the evaluation with newer models, creating another labeling task for measuring alignment between the LLM-generated mutants and human judgement, and testing the evaluation methodology on Hypothesis tests in the wild.

4.6 Summary

In this chapter, we explored and identified the unique challenges in using LLMs to synthesizing property-based tests. We characterized important properties of good property-based tests to propose an evaluation methodology that allows us to rigorously evaluate the LLM outputs. These properties include validity, soundness, and coverage of properties in the documentation. Our results demonstrate the potential of LLMs to synthesize high-quality PBTs.

Chapter 5

The Impact of Parametric Generators in Generator-Based Fuzzing

5.1 Introduction

In chapters 3 and 4, we looked at methods of augmenting the core CGF algorithm and synthesizing PBTs. In these approaches, the underlying strategy for random input generation uses parametric input generators as a method of producing structured inputs. In this chapter, we look to investigate the following research question:

RQ: *What is the impact of parametric input generation on the test-input space?*

As described in Chapter 2.3, parametric generators treat generator functions as decoders of an arbitrary sequence of bytes, producing structurally valid inputs given any pseudo-random input sequence. This combination of (a) a method to produce structurally valid inputs, and (b) a method to make small changes to structurally valid inputs, together enables *structure-aware grey-box fuzzing*, resulting in an ability to test deep program states beyond syntax parsing and validation [19]. The high-level insight, popularized by Zest [19], is that *small changes* in the byte-sequence will map to *small changes* in the structured input produced by the generators (e.g., the binary trees)... at least, in theory.

This insight, while compelling, does not always hold. A common criticism of the parametric generator approach is that certain mutations on the byte stream—especially on those bits whose values influence conditional branches in the generator function—can lead to drastic changes in the corresponding structured input being produced. Fig. 2.3d depicts such a case, where a single bit-flip in the first byte leads to a completely different binary tree being produced; there is almost no similarity to the original tree shown in Fig. 2.3a. We call this phenomenon the *havoc effect*, inspired by the terminology used by AFL [12] and prior work [133, 134].

Intuitively, the havoc effect appears to be a severe limitation of the parametric generator-based approach to structure-aware grey-box fuzzing because the fuzzing process relies on subtle changes to explore program paths incrementally. This unpredictability can potentially degrade the effectiveness of grey-box fuzzing, transforming it into a black-box approach where the feedback mechanism is unable to guide the exploration meaningfully due to the

havoc effect. Researchers have developed techniques that increase structure-preservation when performing mutations on parametric generators: In JQF [18], the EI backend reduces the destructiveness of mutations by tracking in which generation context the bytes are used; BeDivFuzz [33] separates structure-preserving mutations from structure-changing ones; and Zeugma [35] traces generator execution to enable structure-preserving cross-over across distinct inputs.

In this chapter, we investigate the paradoxical nature of the havoc effect in generator-based fuzzing by studying the properties of byte-level mutations, their effect on the generated structured inputs, and the performance of generator-based greybox fuzzing under mutation strategies more and less prone to the havoc effect. We additionally present a case study of property-based testing applied to a project with mature generators, and find that *pure random generation* is able to outperform coverage-guided parametric input generation.

5.2 Consequences of the Havoc Effect

It is hard, based on intuition alone, to determine whether the havoc effect is inherently good or bad for fuzzing performance. As aforementioned, Gramatron [134] explicitly added aggressive mutation strategies to improve the performance of grammar-based greybox fuzzing. Similarly, the standard best practice when running AFL variants is to disable the deterministic mutation stage by default [135, 136]. So, in this section, we look at some instances where the havoc effect could have a negative impact on the coverage achieved.

To investigate this, we ran some preliminary experiments on the Google Closure Compiler benchmark. Closure is a Javascript compiler written in Java, used in the original benchmark suite for Zest [19]. In particular, we ran both Zest [19] and JQF's execution indexing (EI) extension on this benchmark. The JQF EI extension performs finer-grained mutations on the parametric input space with the goal of better preserving structure. Examining several 24-hour runs, we saw JQF's EI had the ability to cover one particular code fragment that Zest did not.

This code fragment is illustrated in Figure 5.1. In particular, Figure 5.1 shows the process by which EI identifies an input covering a branch in the closure compiler's cost estimator for all our initial experiments. This branch is not covered by Zest. A seed input (top left) covers four statements within the `costEstimator` method. In the seed input, the statement `call(foo)` is a method call with a function pointer as an argument. This call triggers the cost estimation path for the `foo` method. To estimate the execution cost for `foo`, the estimator proceeds by iterating and analyzing the cost of each statement enclosed in `foo`. Since the definition of `foo` in the seed input includes a for statement, the seed input covers the `analyzeFor` statement found within the `costEstimator` method.

We note that the generator used in these experiments generates the JavaScript program in sequential order. Initially, it constructs the `foo` method. Following this, it generates the `call(foo)` statement. However, the `costEstimator` method processes the input in reversed order. It interprets the statements contained within the `foo` method if and only if the `call(foo)` expression exists.

To cover the `analyzeIf` statement from here, a mutated input needs to add an if

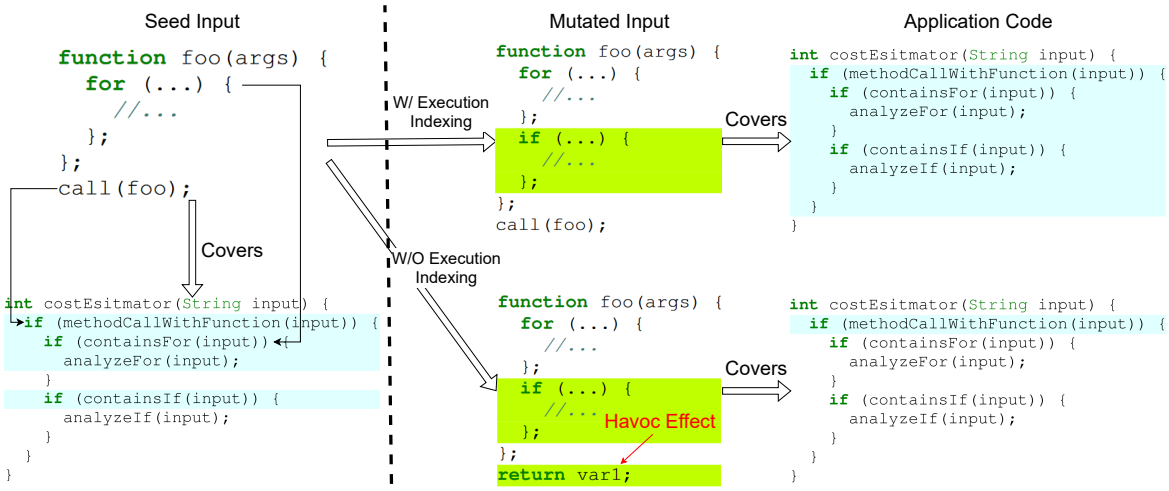


Figure 5.1: Case study of fuzzing the Closure Javascript compiler (written in Java). EI can find an input that covers the `analyzeIf` statement, which requires introducing an `if` statement before the statement calling `foo`; this is hard for Zest to cover due to the havoc effect which disrupts the suffix of the input containing the `foo` call whenever Zest manages to generate an `if` statement.

statement into the `foo` method while keeping the `call(foo)` statement. Let's consider a scenario where both Zest and EI successfully generate a mutation that introduces an if statement after the for statement. Thanks to the structure-preserving mutations provided by execution indexing, EI is able to insert this if statement without disturbing the subsequent statements (top middle of Figure 5.1). Conversely, due to the havoc effect, this mutation in Zest removes the `call(foo)` statement (bottom middle of Figure 5.1). When the closure compiler is executed with these mutated inputs, the one generated by EI successfully extends the coverage to include the `analyzeIf` statement. In contrast, the Zest-generated input fails to augment the coverage: the `call(foo)` statement has been turned into a return statement, so the code guarded by `methodCallWithFunction(input)` is not executed.

This example motivates the value of reducing the havoc effect in order to cover certain program behaviors. Thus, we propose to examine whether different generator-based coverage-guided fuzzing systems demonstrate the havoc effect, as well as whether this results in an overall positive or negative effect on fuzzing performance on a broader benchmark set.

5.2.1 Evaluation Summary

Our preliminary results studying the havoc paradox indicate that there are significant variations in the *mutation distances*, i.e. the Levenshtein distances between parent inputs and its child inputs, for all inputs generated by Zest. We also observe that there is a substantial percentage of *zero mutations*, i.e. Levenshtein distances of 0 between parent inputs and child inputs, observed across all benchmarks. Our evaluation results and proposed extended evaluation can be found in Appendix D.

5.3 Cedar: Random is All You Need

While our analysis on the havoc paradox motivates the need for smarter parametric input mutation strategies, we note that the results are impacted by the input generator. In this section, we report our experience applying coverage-guided fuzzing with parametric input generation on the implementation of Cedar [44].

Cedar is an authorization policy language developed at Amazon Web Services. It was built using an engineering process called verification-guided development (VGD) [137]. VGD first involves creating formal models of Cedar's components in Lean, and then uses differential testing and PBT to show that the models match the production implementation.

A critical aspect of validating Cedar's implementation through DRT and PBT is the generation of random inputs for each Cedar component. For instance, testing the Cedar authorizer requires generating valid combinations of Cedar policies, user data, and authorization requests. This generation process presents unique challenges due to the semantic constraints inherent in Cedar's design. Cedar components impose strict semantic requirements on their inputs, such as type correctness in expressions and the existence of referenced attributes in user data. These constraints make the development of input generators challenging, as they must consistently produce inputs that satisfy all validity conditions while maintaining sufficient diversity for effective testing.

Cedar is implemented in Rust and randomized testing is performed using `cargo-fuzz` [37]. To generate structured inputs for fuzzing, the input struct must derive the `Arbitrary` trait [138]. `cargo-fuzz` performs parametric input generation (ref. Section 2.3) and applies coverage guidance with libFuzzer [13] as a backend. The `Arbitrary` trait provides a default implementation for generating random values of a given type, which can be customized to control the distribution and constraints of generated values.

5.3.1 Approach

To evaluate generator effectiveness in terms of validity, efficiency, and diversity, we developed and analyzed three distinct approaches to recursive input generation: First, we leveraged Rust's built-in derivation capabilities using `#derive(Arbitrary)` to automatically generate implementations of the Arbitrary trait for each input type. Second, we created *fail-fast* generators that terminate immediately upon detecting potential validity violations, such as when attempting to generate an expression referencing a nonexistent attribute. Third, we implemented *fail-fix* generators that, instead of terminating on potential validity errors, substitute problematic expressions with valid constants. We additionally evaluated purely random generation against coverage guidance.

We applied these three generator strategies to test some of Cedar's core components: the validator, which checks a set of policies against a schema; the authorizer, which checks whether authorization requests are allowed or denied given a set of policies; and the evaluator, which computes expression results. This comprehensive testing approach allowed us to assess the relative strengths and limitations of each generation strategy.

Finally, to compare purely random generation against coverage guidance, we used the `cargo-bolero` crate [139], which supports different engines be used as a backend for

fuzzing. We evaluated the generators with a random engine and the libFuzzer engine.

5.3.2 Evaluation Summary

Overall, we observed that coverage guidance was able to improve validity for the derived generators and the fail-fast generators, which follows from prior work [19]. Since the fail-fix generators were designed to always return valid inputs, coverage guidance made no significant change.

However, we noticed that coverage guidance significantly decreased the diversity of generated inputs, with about 40% of the inputs being duplicates when testing the Cedar authorizer. Comparatively, only 1% of generated inputs were duplicates when using purely random generation. We find that this comes from a limitation of mutating parametric inputs–many of the child parametric inputs were decoded into the same Cedar inputs as the parent. This is more evidence of the *zero mutation* observed when studying the havoc paradox. We additionally observed purely random generation significantly improved the diversity of various input qualities over coverage guidance, such as the size and the distribution of node types in the AST. More evaluation results can be found in Appendix E.

5.4 Status

The investigation into the Havoc Paradox and the Cedar generators are both ongoing projects. The Havoc Paradox project is currently under submission as a TOSEM stage 1 registered report and the proposed evaluation of the 4 different parametric mutation strategies needs be completed. The experiments from the Cedar project are completed and my goal is to submit the paper to the FSE 2025 Industry Track.

5.5 Summary

In this chapter, we gain insights and preliminary evidence in some of the drawbacks of using parametric generators for fuzz testing. From the Havoc Paradox, we observe that mutations applied on the byte-level stream can result in inputs too dissimilar to the seed inputs. This, in certain cases, limits the coverage achieved from mutations performed on the byte-level parametric input. The generators used for the JQF benchmarks are all written to satisfy syntactic constraints; thus, coverage guidance is able to help generate a higher percentage of *semantically* valid inputs and improve overall performance. However, we observe that in the case of the type-directed Cedar generators, the coverage guidance applied to parametric input generation actually *decreases* diversity of the input space. This motivates further investigation of how coverage guidance interacts with the generators used in parametric generation.

Chapter 6

Proposed Work and Timeline

In the previous chapters, we (1) developed methods for generating high-quality test inputs by bounding the input generator and adding mutation score as execution feedback, (2) developed methods for automating the synthesis and evaluation of property-based tests, and (3) conducted an analysis of how parametric generators and byte-level mutations influence various characteristics of the generated inputs. Our findings provide evidence that LLMs possess the capability to synthesize random input generators in our study of Hypothesis property-based tests. Our evaluation on the havoc paradox and our case study in Cedar demonstrate that applying byte-level mutations to parametric generators can significantly impact the diversity of the resulting test inputs.

Building upon these insights, we ask a central research question that addresses crucial gaps in our understanding of automated test-input generation:

RQ1: *Can a strong input generator obviate the need for coverage guidance in generator-based fuzz testing?*

This question challenges the conventional wisdom that coverage guidance is always beneficial, particularly in contexts where sophisticated input constraints must be satisfied. In prior work in Zest [19], Bonsai Fuzzing [27], and the havoc paradox, we note that all generators are written to satisfy *syntactic* constraints of the input domain, but not to satisfy semantic constraints. Thus, coverage guidance is able to improve the *semantic validity* of the generated inputs by mutating saved valid inputs and thus improve overall performance. However, we observed that the Cedar generators, written in way to satisfy semantic constraints, resulted in higher diversity and efficiency of generated inputs when using purely random generation instead of coverage guidance. The challenge comes from the overhead of writing these complex generators, which is much higher than writing the syntactically-driven generators seen in the JQF benchmarks. We thus divide our main research question into two parts:

RQ1.1: *Can we automatically synthesize strong input generators that satisfy semantic constraints?*

RQ1.2: *What is the impact of coverage guidance on synthesized generators?*

6.1 Approaches

LLM-based iterative generator pipeline The process of writing generators is very iterative [140]. Developers often use execution feedback to refine their generator implementation. Tools such as Tyche [140] can provide statistical and visual feedback to measure input validity and diversity of a generator, which can then be used to improve upon generator implementations.

LLMs have also shown the ability to refine and repair code generations [141] using execution feedback. With the goal of automatically synthesizing generator implementation, I first propose an LLM-based iterative pipeline. The system will operate as an iterative dialogue between an LLM and the testing infrastructure: the LLM generates an initial implementation of *gen* based on the specification of $p(x)$, and the testing system executes the generator and collects multiple types of feedback. This feedback is incorporated into the next prompt to the LLM in order to revise the generator implementation and address the identified gaps. This will provide a structured dialogue between the LLM and the testing framework for synthesizing and refining a generator implementation.

The execution feedback that can be captured by JQF includes input validity (and examples of invalid inputs), lines of code not covered from generator execution, and program mutants that are executed but still survive. One of the main insights gained from this approach is learning which types of feedback are valuable for refining an initial LLM-synthesized generator.

Feedback-driven generator agent Recently, LLM *agents* have emerged as powerful tools in software engineering. An agent, in the context of LLMs, is a system that can autonomously take actions to achieve specific goals by combining:

1. Planning Capability: The ability to break down complex tasks into smaller steps and determine a sequence of actions to achieve a goal.
2. Tool Use: The ability to interact with external tools and APIs to accomplish tasks. This could include running code, accessing documentation, reading files, etc.
3. Memory and Context Management: The ability to maintain context across multiple steps of a task and remember previous actions and their results.

These agentic approaches have proven to be effective in end-to-end software development [142, 143], fault localization and repair [144], and even discovering new security vulnerabilities [145].

I also propose an LLM-based *agentic* approach to synthesize test-input generators. This approach leverages the LLM's ability to understand both natural language specifications and concrete examples, while using systematic feedback to guide improvements. We will incorporate this generator agent into the JQF framework [18], which extends the `junit-quickcheck-generator` library for writing structured input generators. The difference between this approach and the structured pipeline is providing more autonomy for the LLM to decide which actions to take and how to refine the generator implementations.

While there are many different ways to write random generators, I will primarily focus on designing the agent to synthesize recursive generators written similarly to existing

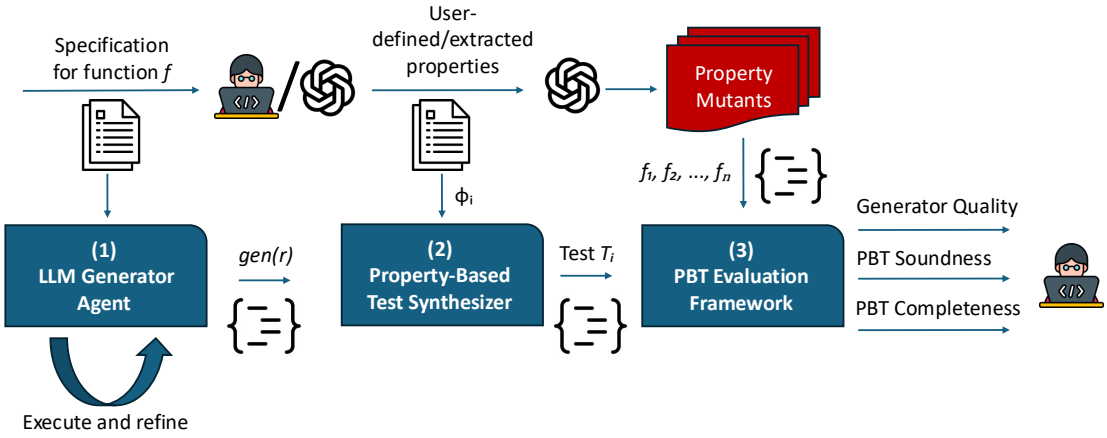


Figure 6.1: End-to-end automated workflow for property-based test generation in JQF

generators in the JQF benchmarks, as this is a popular and structured pattern.

As a final goal, I would also like to integrate the workflow from Proptest-AI into JQF. Figure 6.1 shows an ideal workflow in which a user can provide a specification and/or desired properties to the system, and the system will synthesize a generator, property-based test, and an evaluation summary of the PBT.

6.2 Experimental Design

To investigate these research questions systematically, we propose a evaluation on the JQF benchmarks. Our experimental design will test three competing hypotheses:

1. **Null Hypothesis (H0):** Coverage guidance with syntax-aware generators will produce the most effective results. This hypothesis reflects the current conventional wisdom that a syntax-aware generator and coverage guidance is effective enough for fuzz testing. This is currently used for all the JQF benchmarks and libraries such as `libfuzzer` and Rust `arbitrary`. The benefit of this approach is that the overhead of writing syntax-aware generators is low.
2. **Alternative Hypothesis 1 (H1):** Coverage guidance with stronger semantics-aware generators will produce the most effective results. While intuitively semantics-aware generators should outperform syntax-aware generators since they will always produce valid inputs, the challenge comes from the overhead of writing these generators that must encode domain-specific input constraints. This hypothesis still reflects the conventional wisdom that coverage can help guide input generation to deeper code paths.
3. **Alternative Hypothesis 2 (H2):** Random generation with stronger semantics-aware generators will produce the most effective results. This hypothesis proposes that well-designed semantics-aware generators might be sufficient without the overhead of additional coverage guidance.

6.2.1 Baselines

Syntax-Aware Generators The existing generators in the JQF benchmarks are written to satisfy syntactic constraints, effectively sampling from the grammar (with some bound on recursive depth). These will be used for testing **H0**.

Derived Type-based Generators Another method of synthesizing generators for structured inputs is to derive them automatically based on the type of the input. In `junit-quickcheck-generators`, we can use the *implicit generators* [146] by using the `@From(Ctor.class)` and `@From(Fields.class)` decorators.

LLMs as input generators There has been a recent trend in fuzzing research to use LLMs themselves as test input generators. For example, Fuzz4All [147] uses an LLM to generate and mutate test inputs for C compilers, given language standard as documentation. For any benchmarks with string inputs or that can be serialized, I would like to evaluate whether LLMs themselves are able to outperform the alternative approaches. To implement this baseline, we can simply ignore the parametric input as a source of randomness in JQF and invoke an LLM API in the generator function. Since the cost of this may be too high, I may need to look into using open source models or compare results on a smaller number of generated inputs.

6.2.2 Evaluation Metrics

To compare different generators and guidances, I will report and statistically evaluate the following metrics:

- Branch coverage
- Efficiency of input generation (inputs/sec)
- Validity of generated test inputs
- Diversity of generated test inputs (uniqueness, size distribution)
- Mutation score of generated inputs
- Mean Time to find historical bugs and newly discovered bugs

All of these metrics are currently supported in JQF and its extension Mu2.

6.3 Related Work

There has recently been a large body of work to use large language models for software testing tasks [105, 142, 148]. There are two research ideas closely related to my proposed work that I will describe.

Automatic Fuzz Driver Synthesis The problem of *fuzz harness* or *fuzz driver* generation bears similarity to our input generator synthesis task [149, 150]. In fuzz driver generation, the goal is to take unstructured byte data provided by the fuzz tester, and

use it to exercise the program under test in a meaningful manner. UTopia [151] aims to extracts fuzz drivers from unit tests, note that some unit tests assertions (e.g., checking null pointers) must be preserved to maintain property validity. LLMs also have been applied for the task of fuzz driver generation [106], in a zero-shot prompting approach. OSS-Fuzz-Gen [108] is a workflow to prompt LLMs for automated fuzz driver generation for targets in OSS Fuzz. For this task, Zhang et al. [106] note a limitation that "the performance of LLM-based generation declines significantly when the complexity of API specific usage increases". PromptFuzz [152] guides fuzz driver generation using code coverage. However, the majority of the target programs do not have semantic pre-conditions on the inputs and simply require sequences of bytes in a specific format. A significant portion of the prior work focuses on satisfying correct types for invoking the API method under test, rather than the satisfying pre-conditions for the structured input. Further, the LLM-generated fuzz drivers are primarily evaluated using code coverage and bugs; we look to evaluate input generators along more dimensions, as listed in the previous section.

Large Language Models for Input Generation Recently, researchers have developed methods of providing specific documentation as context to the LLM for the purpose of generating test inputs that exercise specific behavior. Fuzz4All [147] provides language standards to develop and *autoprompting* workflow to prompt LLMs to act as input generation and mutation engines and fuzz various compilers. DiffSpec [153] compiles historical bug data and typing rules from documentation to provide to LLMs in order to differentially test the `wasm` validator. Ackermann et al. [154] utilize LLMs to examine natural language format specifications to generate strong seeds for a mutation-based fuzzing. Meng et al. [155] develop ChatAFL to enhance protocol fuzzing by enriching initial seeds with specific LLM message outputs. FuzzGPT [156] and WhiteFox [157] prompt LLMs to synthesize Python programs for to test functionality deep-learning libraries and compilers. EvalPlus augments the HumanEval unit testing dataset by prompting the LLM to produce seed inputs and performing further type-based mutation [158]. While this type of approach have been successful at improving upon existing state-of-the-art fuzzing baselines, it is still unclear how it would compare to an LLM-synthesized generator. Jiang et al. [159] detail some of the limitations with these approaches, including insufficient input diversity and limited validity. My proposed work aims to address these challenges by incorporating diversity, validity, and code coverage as part of the LLM agent design and leveraging JQF as a testing framework to provide these metrics.

6.4 Timeline

Table 6.1 shows an estimated timeline from January 2025 - June 2026 for the completion of existing projects, the proposed project, and the dissertation.

Dataset Collection While the JQF benchmarks provide a good set of target programs to implement the LLM-based generator agent, I would like to expand the evaluation to include more types of inputs with varying constraints. It is possible that the JQF

Table 6.1: Timeline of proposed work. Boxes in green are completed work, red are ongoing, and blue are planned.

	Tasks	Prior	2025		2026
			Spring	Fall	Spring
Chapter 3					
Chapter 4	Paper Revision and Resubmission				
Chapter 5	Paper Submissions				
Chapter 6	Dataset Creation				
	Implementation of Technique				
	Evaluation				
	Paper Submission				
Dissertation	Dissertation Writing				
	Thesis Defense				

generator source code exist in the training data of large language models and may thus result in unrepresentative . I plan to expand the targets by finding Java projects that use the `junit-quickcheck` library. This can be done using Github search, the Open Source Insights [95], or the Boa infrastructure for mining software repositories [160]. To find PBTs with input constraints, we can target tests that use the `Assume` statement in `junit-quickcheck`.

For each project, we would like a list of existing property-based tests, documentation for any input constraints and existing generators. Each PBT will need to be converted from `junit-quickcheck` into JQF in order to run coverage-guided fuzzing. I estimate this expansion of the JQF benchmarks will take 1-2 months.

Implementation of LLM Generator Agent The LLM-based generator agent will be the bulk of the project since there are many ways to design an AI agent for a programming task.I will be able to use insights from the Proptest-AI work for developing prompting strategies and providing contexts for the agent. There will also need to be some additional work on JQF to provide as a tool for the agent and to give feedback such as invalid inputs and uncovered lines of code. I estimate this will take 3-4 months to prototype and iterate on.

Evaluation The evaluation will be computationally expensive to synthesize generators and run them for long experiments. All of the evaluation metrics are currently supported in JQF, so the majority of time will be spent running the evaluation and analyzing the results. I estimate this will take 2-3 months.

Bibliography

[1] H. Krasner, "The cost of poor software quality in the us: A 2022 report," *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, vol. 2, 2022.

[2] M. Nazzaro, "Google ceo says more than 25 percent of company's new code written by ai," https://thehill.com/policy/technology/4962336-google-ceo-says-more-than-25-percent-of-companys-new-code-written-by-ai/, October 2024, accessed October 30, 2024.

[3] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," *arXiv preprint arXiv:2310.03533*, 2023.

[4] P. Wadhwani and A. Ambekar, "Software testing market size - by component (application, services), by type (system integrator, pureplay software testing), by industry (mobile, web-based), by business type (b2b, b2c), by application & forecast, 2024 - 2032," https://www.gminsights.com/industry-analysis/software-testing-market, June 2024, Accessed October 30, 2024.

[5] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, 2000, pp. 268–279.

[6] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, dec 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[7] T. Arts, J. Hughes, J. Johansson, and U. Wiger, "Testing telecoms software with quviq quickcheck," in *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, 2006, pp. 2–10.

[8] T. Arts, J. Hughes, U. Norell, and H. Svensson, "Testing autosar software with quickcheck," in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2015, pp. 1–4.

[9] J. Hughes, "Experiences with quickcheck: testing the hard stuff and staying sane," in *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer, 2016, pp. 169–186.

[10] J. Hughes, B. C. Pierce, T. Arts, and U. Norell, "Mysteries of dropbox: property-based testing of a distributed synchronization service," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016,

pp. 135–145.

[11] D. R. MacIver, Z. Hatfield-Dodds *et al.*, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019.

[12] M. Zalewski, "American fuzzy lop," https://lcamtuf.coredump.cx/afl/, 2014, accessed February 11, 2022.

[13] "libfuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, [n. d.], accessed: 2021-08-31.

[14] 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*, ser. CCS, 2016, pp. 1032–1043.

[15] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[16] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[17] 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)*, 2019, pp. 1949–1966.

[18] 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*, ser. ISSTA'19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 398–401. [Online]. Available: http://doi.acm.org/10.1145/3293882.3339002

[19] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with Zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. ACM, 2019, pp. 329–340. [Online]. Available: http://doi.acm.org/10.1145/3293882.3330576

[20] L. Lampropoulos, M. Hicks, and B. C. Pierce, "Coverage guided, property based testing," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360607

[21] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[22] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data flow sensitive fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2577–2594.

[23] Z. Y. Ding and C. Le Goues, "An empirical study of oss-fuzz bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, IEEE. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 131–142. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MSR52588.2021.00026

[24] Google, "Ideal integration with OSS-Fuzz," https://google.github.io/oss-fuzz/advanced-topics/ideal-integration/#regression-testing, 2019, retrieved August 31, 2022. [Online]. Available: https://web.archive.org/web/20200301084941/https://google.github.io/oss-fuzz/advanced-topics/ideal-integration/#regression-testing

[25] SQLite Authors, "How SQLite is Tested," https://www.sqlite.org/testing.html#the_fuzzcheck_test_harness, 2019, retrieved August 31, 2022. [Online]. Available: https://web.archive.org/web/20200427011538/https://www.sqlite.org/testing.html#the_fuzzcheck_test_harness

[26] The OpenSSL Project, "Run the fuzzing corpora as tests." https://github.com/openssl/openssl/commit/90d28f05, 2016, retrieved August 31, 2022. [Online]. Available: https://github.com/openssl/openssl/tree/openssl-3.0.0/fuzz/corpora

[27] V. Vikram, R. Padhye, and K. Sen, "Growing a test corpus with bonsai fuzzing," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 723–735. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00072

[28] H. L. Nguyen and L. Grunske, "Bedivfuzz: Integrating behavioral diversity into generator-based fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 249–261. [Online]. Available: https://doi.org/10.1145/3510003.3510182

[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[30] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *44th IEEE/ACM International Conference on Software Engineering*, ser. ICSE'22, 2022, to appear.

[31] C. Miller, Z. N. Peterson *et al.*, "Analysis of mutation and generation-based fuzzing," *Independent Security Evaluators, Tech. Rep*, vol. 4, 2007.

[32] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[33] H. L. Nguyen and L. Grunske, "Bedivfuzz: Integrating behavioral diversity into generator-based fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 249–261.

[34] J. Kukucka, L. Pina, P. Ammann, and J. Bell, "Confetti: Amplifying concolic guidance for fuzzers," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 438–450.

[35] K. Hough and J. Bell, "Crossover in parametric fuzzing," in *Proceedings of the*

IEEE/ACM 46th International Conference on Software Engineering, 2024, pp. 1–12.

[36] "Structure-aware fuzzing with libFuzzer," https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md, [n. d.], accessed: 2022-06-02.

[37] "cargo-fuzz," https://github.com/rust-fuzz/cargo-fuzz, [n. d.], accessed: 2023-05-01.

[38] 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.

[39] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1051–1065. [Online]. Available: https://doi.org/10.1145/3548606.3560602

[40] "libFuzzer – how to split a fuzzer-generated input into several parts," https://github.com/google/fuzzing/blob/41d7725/docs/split-inputs.md, [n. d.], accessed: 2021-08-31.

[41] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 861–875.

[42] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 725–741.

[43] 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*, 2018, pp. 475–485.

[44] J. W. Cutler, C. Disselkoen, A. Eline, S. He, K. Headley, M. Hicks, K. Hietala, E. Ioannidis, J. Kastner, A. Mamat *et al.*, "Cedar: A new language for expressive, fast, safe, and analyzable authorization," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 670–697, 2024.

[45] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th international conference on software engineering*, ser. ICSE'14, 2014, pp. 435–445.

[46] Y. T. Chen, R. Gopinath, A. Tadakamalla, M. D. Ernst, R. Holmes, G. Fraser, P. Ammann, and R. Just, "Revisiting the relationship between fault detection, test adequacy criteria, and test set size," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. Association for Computing Machinery, 2020, p. 237–249. [Online]. Available: https://doi.org/10.1145/3324884.3416667

[47] R. Padhye, K. Sen, and P. N. Hilfinger, "Chocopy: A programming language for compilers courses," in *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, ser. SPLASH-E 2019. Association for Computing Machinery, 2019, p. 41–45. [Online]. Available: https://doi.org/10.1145/3358711.3361627

[48] 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*, ser. PLDI '11, 2011.

[49] M. H. Pałka, K. Claessen, A. Russo, and J. Hughes, "Testing an optimising compiler by generating random lambda terms," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 91–97. [Online]. Available: https://doi.org/10.1145/1982595.1982615

[50] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the rust typechecker using clp," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, p. 482–493. [Online]. Available: https://doi.org/10.1109/ASE.2015.65

[51] B. Fetscher, K. Claessen, M. Pałka, J. Hughes, and R. B. Findler, "Making random judgments: Automatically generating well-typed terms from the definition of a type-system," in *European Symposium on Programming Languages and Systems*. Springer, 2015, pp. 383–405.

[52] A. Arcuri, "A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage," *IEEE Transactions on Software Engineering*, vol. 38, no. 3, pp. 497–519, 2012.

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

[54] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 147–158. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831728

[55] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2012.

[56] Yong Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, 2005, pp. 10 pp.–276.

[57] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 417–420. [Online]. Available: https://doi.org/10.1145/1321631.1321698

[58] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ser. ICSTW '10. USA: IEEE Computer Society, 2010, p. 182–191. [Online]. Available: https://doi.org/10.1109/ICSTW.2010.31

[59] P. Godefroid, "Fuzzing: Hack, art, and science," *Communications of the ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[60] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[61] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 142–151. [Online]. Available: https://doi.org/10.1145/1134285.1134307

[62] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[63] R. Scowen, "International Standard ISO/IEC 14977:1996—Extended BNF," 1996.

[64] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "SLF: Fuzzing without valid seed inputs," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 712–723. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00080

[65] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[66] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE'14, 2014, pp. 654–665.

[67] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 537–548.

[68] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for Java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA'16, 2016, pp. 449–452.

[69] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[70] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *SIGPLAN Not.*, vol. 10, no. 6, p. 493–510, apr 1975. [Online]. Available: https://doi.org/10.1145/390016.808473

[71] E. J. Weyuker, "Axiomatizing software test data adequacy," *IEEE transactions on software engineering*, no. 12, pp. 1128–1138, 1986.

[72] R. Gopinath, P. Görz, and A. Groce, "Mutation analysis: Answering the fuzzing challenge," *CoRR*, vol. abs/2201.11303, 2022. [Online]. Available: https://arxiv.org/abs/2201.11303

[73] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019,

vol. 112, pp. 275–378.

[74] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05. Association for Computing Machinery, 2005, p. 402–411. [Online]. Available: https://doi.org/10.1145/1062455.1062530

[75] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?" in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 189–200.

[76] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 597–608.

[77] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. Le Traon, "How effective are mutation testing tools? an empirical analysis of Java mutation testing tools with manual analysis and real faults," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2426–2463, 2018.

[78] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, p. 99–118, apr 1996. [Online]. Available: https://doi.org/10.1145/227607.227610

[79] P. Ammann, "Transforming mutation testing from the technology of the future into the technology of the present," in *International conference on software testing, verification and validation workshops (ICST): Mutation workshop*. IEEE, 2015. [Online]. Available: https://mutation-workshop.github.io/2015/program/MutationKeynote.pdf

[80] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. Le Traon, and A. Ventresque, "Assessing and improving the mutation testing practice of PIT," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 430–435.

[81] H. L. Nguyen, N. Nassar, T. Kehrer, and L. Grunske, "MoFuzz: A fuzzer suite for testing model-driven software engineering tools," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1103–1115.

[82] S. Reddy, C. Lemieux, R. Padhye, and K. Sen, "Quickly generating diverse valid test inputs with reinforcement learning," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1410–1421.

[83] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim, "Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 722–733.

[84] J. Kukucka, L. Pina, P. Ammann, and J. Bell, "Confetti: Amplifying concolic guidance

for fuzzers," in *44th IEEE/ACM International Conference on Software Engineering*, ser. ICSE'22, 2022, pp. 438–450.

[85] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[86] W. M. McKeeman, "Differential testing for software," *DIGITAL TECHNICAL JOURNAL*, vol. 10, no. 1, pp. 100–107, 1998.

[87] R. B. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, 2007, pp. 549–552.

[88] 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*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[89] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA'14, 2014, pp. 315–326.

[90] Google, "Google Closure Compiler," https://github.com/google/closure-compiler, 2022, retrieved August 31, 2022.

[91] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, 2014, p. 216–226. [Online]. Available: https://doi.org/10.1145/2594291.2594334

[92] A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, "A systematic literature review of techniques and metrics to reduce the cost of mutation testing," *Journal of Systems and Software*, vol. 157, p. 110388, 2019.

[93] M. P. Usaola and P. R. Mateo, "Mutation testing cost reduction techniques: A survey," *IEEE Software*, vol. 27, no. 3, pp. 80–86, 2010.

[94] V. Vikram, I. Laybourn, A. Li, N. Nair, K. OBrien, R. Sanna, and R. Padhye, "Guiding greybox fuzzing with mutation testing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 929–941. [Online]. Available: https://doi.org/10.1145/3597926.3598107

[95] Google, "Open Source Insights," https://deps.dev/, 2023, retrieved June 1, 2023.

[96] H. Goldstein, J. W. Cutler, D. Dickstein, B. C. Pierce, and A. Head, "Property-based testing in practice," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 971–971.

[97] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards,

Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021.

[98] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[99] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg, H. Nori, H. Palangi, M. T. Ribeiro, and Y. Zhang, "Sparks of artificial general intelligence: Early experiments with gpt-4," 2023.

[100] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[101] OpenAI, "Gpt-4 technical report," 2023.

[102] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *45th International Conference on Software Engineering, ser. ICSE*, 2023.

[103] S. Lahiri, A. Naik, G. Sakkas, P. Choudhury, C. von Veh, M. Musuvathi, J. P. Inala, C. Wang, and J. Gao, "Interactive code generation via test-driven user-intent formalization," arXiv, August 2022. [Online]. Available: https://www.microsoft.com/en-us/research/publication/interactive-code-generation-via-test-driven-user-intent-formalization/

[104] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "Adaptive test generation using a large language model," 2023.

[105] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2023.

[106] C. Zhang, M. Bai, Y. Zheng, Y. Li, X. Xie, Y. Li, W. Ma, L. Sun, and Y. Liu, "Understanding large language model based fuzz driver generation," *arXiv preprint arXiv:2307.12469*, 2023.

[107] L. Huang, P. Zhao, H. Chen, and L. Ma, "Large language models based fuzzing techniques: A survey," *arXiv preprint arXiv:2402.00350*, 2024.

[108] D. Liu, J. Metzman, and O. Chang, "Open Source Insights," https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html, 2023, retrieved February 27, 2024.

[109] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[110] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[111] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[112] R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du *et al.*, "Lamda: Language models for dialog applications," *arXiv preprint arXiv:2201.08239*, 2022.

[113] M. Reid, N. Savinov, D. Teplyashin, D. Lepikhin, T. Lillicrap, J.-b. Alayrac, R. Soricut, A. Lazaridou, O. Firat, J. Schrittwieser *et al.*, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," *arXiv preprint arXiv:2403.05530*, 2024.

[114] Anthropic, "Introducing the next generation of Claude," https://www.anthropic.com/news/claude-3-family, retrieved April 1, 2024.

[115] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, 2023.

[116] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[117] R. Li, L. Ben Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Randy, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, S. Gunasekar, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "STARCODER: May the Source be With You!" 2023. [Online]. Available: https://drive.google.com/file/d/1cN-b9GnWtHzQRoE7M7gAEyivY0kl4BYs/view

[118] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[119] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[120] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[121] J. A. Prenner and R. Robbes, "Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs," *arXiv preprint arXiv:2111.03922*, 2021.

[122] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs?" *arXiv preprint arXiv:2112.02125*, 2021.

[123] ——, "Examining zero-shot vulnerability repair with large language models," in *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 1–18. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00001

[124] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models," in *Proceedings of the 2022 ACM Conference on International Computing Education Research V. 1*, 2022, pp. 27–43.

[125] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.

[126] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," in *Proceedings of the international conference on Reliable software*, 1975, pp. 493–510.

[127] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," in *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, 1998, pp. 153–162.

[128] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, pp. 1–42, 2014.

[129] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 201–211.

[130] A. R. Ibrahimzada, Y. Chen, R. Rong, and R. Jabbarvand, "Automated bug generation in the era of large language models," *arXiv preprint arXiv:2310.02407*, 2023.

[131] A. Garg, R. Degiovanni, M. Papadakis, and Y. Le Traon, "On the coupling between vulnerabilities and llm-generated mutants: A study on vul4j dataset."

[132] V. Vikram, C. Lemieux, J. Sunshine, and R. Padhye, "Can large language models write good property-based tests?" *arXiv preprint arXiv:2307.04346*, 2023.

[133] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1634–1645. [Online]. Available: https://doi.org/10.1145/3510003.3510174

[134] P. Srivastava and M. Payer, "Gramatron: Effective grammar-aware fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA:

Association for Computing Machinery, 2021, p. 244–256. [Online]. Available: https://doi.org/10.1145/3460319.3464814

[135] 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*, 2018, pp. 475–485.

[136] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: An open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1393–1403. [Online]. Available: https://doi.org/10.1145/3468264.3473932

[137] C. Disselkoen, A. Eline, S. He, K. Headley, M. Hicks, K. Hietala, J. Kastner, A. Mamat, M. McCutchen, N. Rungta *et al.*, "How we built cedar: A verification-guided approach," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 351–357.

[138] "The `Arbitrary` trait crate: Generating structured data from arbitrary, unstructured input." https://github.com/rust-fuzz/arbitrary, accessed: 2023-05-01.

[139] "cargo-fuzz," https://github.com/camshaft/bolero, [n. d.], accessed: 2023-05-01.

[140] H. Goldstein, J. Tao, Z. Hatfield-Dodds, B. C. Pierce, and A. Head, "Tyche: Making sense of pbt effectiveness," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–16.

[141] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[142] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh *et al.*, "Openhands: An open platform for ai software developers as generalist agents," *arXiv preprint arXiv:2407.16741*, 2024.

[143] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *arXiv preprint arXiv:2405.15793*, 2024.

[144] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large language model-based agents for software engineering: A survey," *arXiv preprint arXiv:2409.02977*, 2024.

[145] B. S. team, "From naptime to big sleep: Using large language models to catch vulnerabilities in real-world code," November 2024.

[146] "junit-quickcheck-generator," https://pholser.github.io/junit-quickcheck/site/1.0/usage/other-types.html, [n. d.], accessed: 2024-10-31.

[147] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," *Proc. IEEE/ACM ICSE*, 2024.

[148] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature

review," *ACM Transactions on Software Engineering and Methodology*, 2023.

[149] 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*, 2019, pp. 975–985.

[150] K. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 2271–2287.

[151] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. Hwang, "Utopia: Automatic generation of fuzz driver using unit tests," in *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 746–762. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00043

[152] Y. Lyu, Y. Xie, P. Chen, and H. Chen, "Prompt fuzzing for fuzz driver generation," *arXiv preprint arXiv:2312.17677*, 2023.

[153] N. Rao, E. Gilbert, T. Ramananandro, N. Swamy, C. L. Goues, and S. Fakhoury, "Diffspec: Differential testing with llms using natural language specifications and code artifacts," *arXiv preprint arXiv:2410.04249*, 2024.

[154] J. Ackerman and G. Cybenko, "Large language models for fuzzing parsers (registered report)," in *Proceedings of the 2nd International Fuzzing Workshop*, 2023, pp. 31–38.

[155] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, "Large language model guided protocol fuzzing," in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[156] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt," *arXiv preprint arXiv:2304.02014*, 2023.

[157] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "Whitefox: White-box compiler fuzzing empowered by large language models," *arXiv preprint arXiv:2310.15991*, 2023.

[158] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[159] Y. Jiang, J. Liang, F. Ma, Y. Chen, C. Zhou, Y. Shen, Z. Wu, J. Fu, M. Wang, S. Li, and Q. Zhang, "When fuzzing meets llms: Challenges and opportunities," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 492–496. [Online]. Available: https://doi.org/10.1145/3663529.3663784

[160] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of*

the 35th International Conference on Software Engineering, ser. ICSE'13, 2013, pp. 422–431.

[161] U. C. Berkeley, "Chocopy," https://chocopy.org/, 2019, reference compiler JAR retrieved on January 12, 2022.

[162] R. Hodovan, "Picire," https://github.com/renatahodovan/picire.

[163] ——, "Picireny," https://github.com/renatahodovan/picireny.

[164] R. Hodován and Á. Kiss, "Practical improvements to the minimizing delta debugging algorithm." in *ICSOFT-EA*, 2016, pp. 241–248.

[165] ——, "Modernizing hierarchical delta debugging," in *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, 2016, pp. 31–37.

[166] R. Hodován, Á. Kiss, and T. Gyimóthy, "Coarse hierarchical delta debugging," in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 194–203.

[167] ——, "Tree preprocessing and test outcome caching for efficient hierarchical delta debugging," in *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*. IEEE, 2017, pp. 23–29.

[168] M. R. Hoffmann, B. Janiczak, and E. Mandrikov, "Eclemma-JaCoCo Java code coverage library," 2011.

[169] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[170] H. Coles, "PITest," https://pitest.org/, 2016.

[171] Google, "Gson: A Java serialization/deserialization library to convert Java objects into JSON and back," https://github.com/google/gson, 2021, retrieved August 31, 2022.

[172] FasterXML, "Jackson: JSON for Java," https://github.com/FasterXML/jackson, retrieved August 31, 2022.

[173] Apache Foundation, "Tomcat," https://github.com/apache/tomcat, 2022, retrieved August 31, 2022.

[174] V. Vikram, I. Laybourn, A. Li, N. Nair, K. OBrien, R. Sanna, and R. Padhye, "Mu2: Guiding Greybox Fuzzing with Mutation Testing (Artifact)," May 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8006662

[175] S. J. Kaufman, R. Featherman, J. Alvin, B. Kurtz, P. Ammann, and R. Just, "Prioritizing mutants to guide mutation testing," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. Association for Computing Machinery, 2022, p. 1743–1754. [Online]. Available: https://doi.org/10.1145/3510003.3510187

[176] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[177] "Apache ant is a java-based build tool," https://github.com/apache/ant, [n. d.].

[178] F. P. Miller, A. F. Vandome, and J. McBrewster, *Apache Maven*. Alpha Press, 2010.

[179] "Rhino: Javascript in java," https://github.com/mozilla/rhino, [n. d.].

[180] P. Holser, "junit-quickcheck: Property-based testing, junit-style," https://github.com/pholser/junit-quickcheck, accessed: 2021-08-31.

Appendix

Appendix A: Evaluation of Bonsai Fuzzing

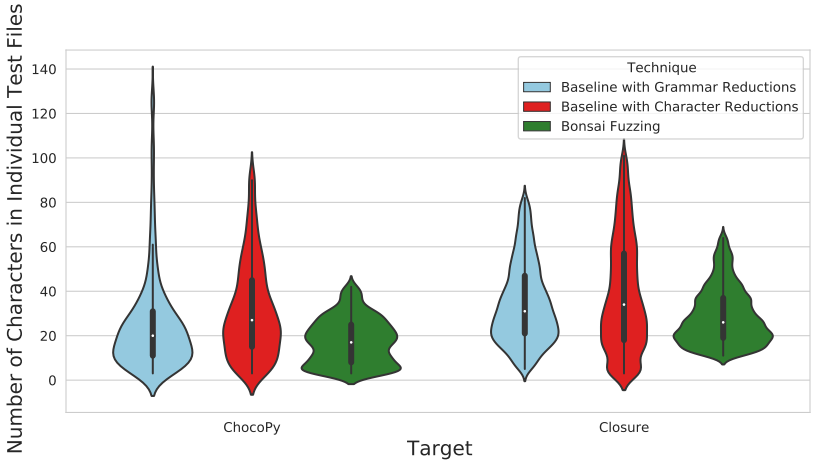


Figure 6.2: Distribution of size of individual test files, excluding whitespace characters, in saved test corpora. Lower is better.

We evaluate bonsai fuzzing by measuring its ability to generate a test corpus containing test cases that are *concise*, *comprehensive*, *semantically valid*, and (where applicable) able to detect faults. We compare bonsai fuzzing to a baseline of CBGF (that is; Zest [19] with a grammar-based input generator) post-processed with minimization techniques. The baseline is thus the conventional "fuzz-then-reduce" approach. We run our evaluation on two test targets: our primary application and a secondary target to ensure that our solution is not biased towards a particular implementation or input language.

1. ChocoPy [161] reference compiler (∼6K LoC): The test driver reads in a ChocoPy program and runs the semantic analysis / type-checking stage of the ChocoPy reference compiler. For the fault-detection evaluation, we additionally run a differential test on the typed ASTs returned by a reference and buggy compiler (see Section 6.4).
2. Google Closure Compiler[90] (∼250K LoC): The test driver (borrowed from prior work [19]) expects a JavaScript program as input and performs source-to-source optimizations.

Experimental Setup

1. **Bound**: Overall, we found the bounds of ($M = N = D = 3$) to be a good trade-off between conciseness and comprehensiveness. We use these bounds for bonsai fuzzing as well as for the baseline CBGF.
2. **Duration**: We run each CBGF node in the bonsai fuzzing extended lattice for one hour, which totals 54 hours of CPU time. We allocate the same 54 hours of CPU time for the baseline CBGF to run[1]
3. **Repetition**: We run each experiment 10 times and report metrics across all repetitions due to the nature of randomness in fuzzing and its effect on results.

Minimization Techniques

For the fuzz-then-reduce baseline, we use Picire [162] and Picireny [163], which are state-of-the-art [164, 165, 166, 167] implementations of character-level [60] and grammar-based hierarchical [61] delta debugging respectively. An "interestingness" predicate script was required for each of these tools. We provided a predicate that checked whether a candidate minimized input program met the same criterion as was used to save the original input during CBGF. Table 6.2 lists the average CPU-time for each of these reduction tools to minimize an entire corpus.

Conciseness: Test Corpus Size

We evaluate *conciseness* by measuring the size of each test file—excluding whitespace characters—in the generated corpus. Fig. 6.2 displays the distribution of test input sizes for the baseline and bonsai fuzzing.

On both targets, we observed that bonsai fuzzing produces test files that are statistically significantly lower in size than those of the baseline. The ChocoPy files are on average 42.22% smaller than the results of grammar-based reduction and 44.51% smaller than the results of character based reduction. The Closure files are on average 16.49% smaller than the results of grammar-based reduction and 25.56% smaller than the results of character-based reduction. We also see that the variance of the size of files in the violin plot of bonsai fuzzing is much lower than that of the baseline. One clear advantage is that bonsai fuzzing is able to produce these smaller inputs *without requiring any additional post-processing time*. In contrast, the fuzz-then-reduce approach of the baseline can take up to 6 hours for minimization to run.

As a sanity check, we also report the number of files in the test corpora as shown in Table 6.3. The resulting corpus from bonsai fuzzing contains about 18% fewer files in both targets. This shows that bonsai fuzzing does not compensate for its smaller test inputs by having a large number of tests.

[1]We chose these durations because one hour is sufficient time for coverage to stagnate for each CBGF node, and because it helps us make a fair comparison with the baseline by fixing total fuzzing duration to a constant. Bonsai Fuzzing can be optimized by stopping each CBGF early by detecting saturation dynamically, but this would make the total fuzzing duration variable. Our evaluation is conservative.

Table 6.2: Time to minimize Zest-saved test inputs (minutes, avg ± stdev).

	ChocoPy	Closure
Picireny Grammar Reductions	56.510 ± 1.887	356.863 ± 37.560
Picire Character Reductions	20.491 ± 3.209	392.777 ± 48.456

Table 6.3: Number of files in test corpus (avg ± stdev). Lower is better.

	ChocoPy	Closure
Baseline	185.9 ± 7.666	1507.7 ± 28.351
Bonsai fuzzing	152.9 ± 1.912	1231.2 ± 36.705

Semantic Validity

One of our goals was to generate a high fraction of semantically valid inputs. For each input program in the saved test corpora, we re-run the ChocoPy compiler to test whether the input is semantically valid or whether the compiler reports any errors.

The average percent of semantically valid programs in the generated corpora is shown in Fig. 6.3. Bonsai fuzzing has a statistically significant increase in both targets. On average, it is able to achieve a 21% improvement in validity in ChocoPy and a 7% improvement in Closure. Why is this so? In the initial round of bonsai fuzzing, sampling smaller programs leads to a higher likelihood of semantically valid inputs as compared to sampling a larger program from scratch. In subsequent rounds, it is easier to mutate a small valid program into a slightly larger valid program, as there are less opportunities to introduce errors. We observed that the baseline's seed pool quickly fills up with invalid or large programs early-on in the fuzzing campaign, making it harder to recover in producing diverse valid inputs via random mutations.

We value this improvement in validity resulting from bonsai fuzzing, since it means that more language features are being covered by test cases that are semantically valid, which in our opinion results in more meaningful and readable test cases.

Comprehensiveness: Coverage

A key concern when generating small inputs by construction is whether they *comprehensively* exercise various program behaviors as conventional coverage-guided fuzzing.

We measure coverage using a third-party tool: the widely used JaCoCO library[168]. We report the branch coverage on the semantic analysis classes within each of the benchmarks, similar to approach in [19]. Since many of the branches are unreachable from our test drivers, it is important to focus on the relative difference between the baseline and bonsai fuzzing rather than the raw coverage values.

Fig. 6.4 shows the branch coverage achieved by the baseline and bonsai fuzzing on each of the targets. We can see that both techniques achieve approximately the same branch coverage. On Closure, the difference is statistically insignificant. On ChocoPy, the difference is significant but its effect is small: bonsai fuzzing loses 1.175% of branch coverage on average. We are not dismayed with this small reduction. In our application,

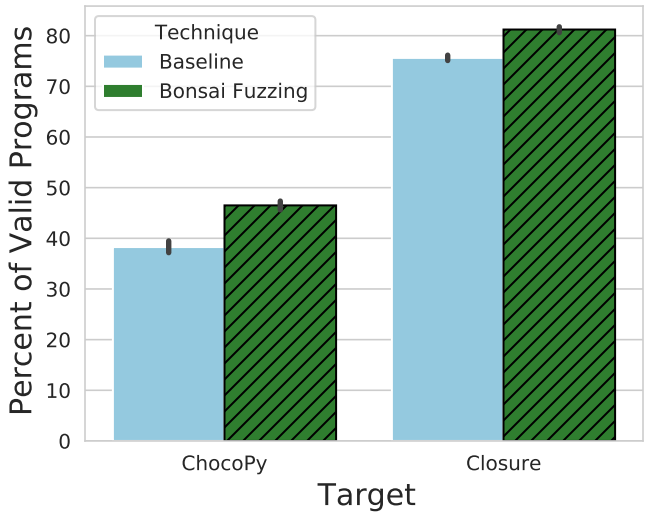


Figure 6.3: Fraction of semantically valid programs in test corpora (averages with standard deviation). Higher is better.

Table 6.4: Mutation scores for ChocoPy typechecker (avg ± stdev)

Baseline	$91.486 \pm 1.012\%$
Bonsai Fuzzing	$90.428 \pm 0.714\%$

we can easily incorporate the few test cases from conventional fuzzing that cover logic that is not exercised by bonsai fuzzing—in ChocoPy, this is usually just one test case.

Fault Detection: Mutation Scores

Finally, we want to ensure that the concise inputs generated by bonsai fuzzing for the ChocoPy target are still useful for catching faults; that is, they can be used for automated grading or providing student feedback. This is essentially a validation of the small scope hypothesis [169]. In a classroom setting, we would compare a candidate buggy student implementation with the reference implementation. For our experimental evaluation, we simulate such a buggy candidate by using a mutation testing tool [170] on a copy of the reference compiler. We run the ChocoPy autograder on the reference compiler and its mutation; if the auto-grader detects a failure, then the mutation is *killed*.

The test corpus saved by bonsai fuzzing by itself achieves a mutation-killing score of 81% on average. This is despite the fact that the fuzzing technique and input-saving criteria is related to coverage improvements within the reference compiler only, and is unaware of program mutations or bugs in student implementations. As observed by Chen et al. [46], a better technique for increasing fault detection while minimizing test sizes is to first optimize for coverage and then optimize for mutation scores when coverage saturates. We thus use the corpus produced by bonsai fuzzing (and the baseline, for comparison) as seed inputs for a simple grammar-based blackbox fuzzer with the maximum bounds $(3, 3, 3)$ for 30 minutes. We do this for *each* of the 444 mutated compilers–that is, simulated buggy candidates. If

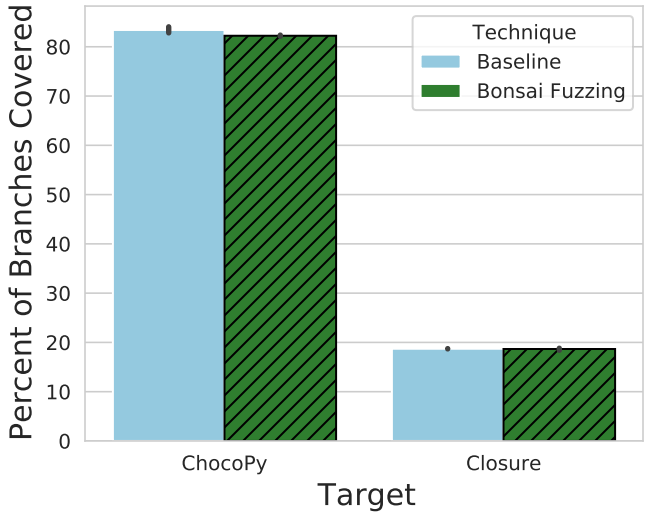


Figure 6.4: Branch coverage in semantic analysis stages achieved by saved test corpora (averages with standard deviation). Higher is better.

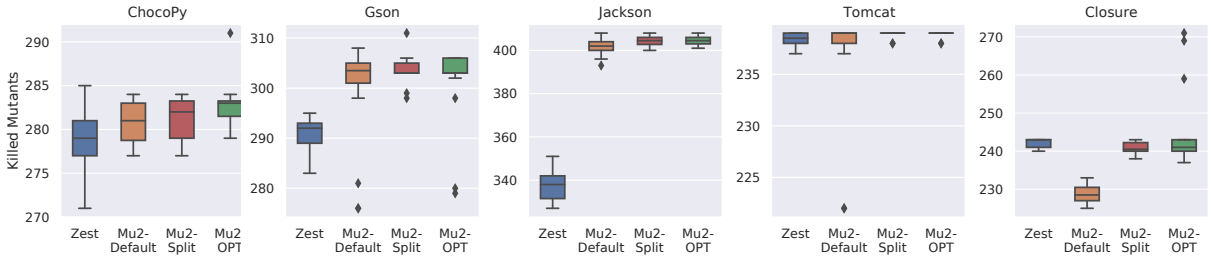


Figure 6.5: Box plots showing the number of killed mutants by Zest and Mu2-generated test corpora across 20 repetitions of 24-hour fuzzing campaigns (higher is better). Mu2-Split and Mu2-OPT are two variants of Mu2 detailed in Appendix B.

any blackbox -fuzzer-generated input kills the mutation, we say that the corresponding technique kills that mutation.

Table 6.4 summarizes these results. Both the baseline and bonsai fuzzing achieve more than 90% mutation-killing score, which we find to be acceptable. We therefore conclude that size-bounded fuzzing does not significantly sacrifice fault detection capability on ChocoPy. Unfortunately, we cannot report meaningful mutation scores on Closure, since the project does not have a proper differential testing oracle.

Appendix B: Evaluation of Mu2

We evaluate Mu2 on 5 different Java program benchmarks, using state-of-the-art coverage-guided fuzzer Zest [19] as the baseline. We structure our evaluation around four research questions:

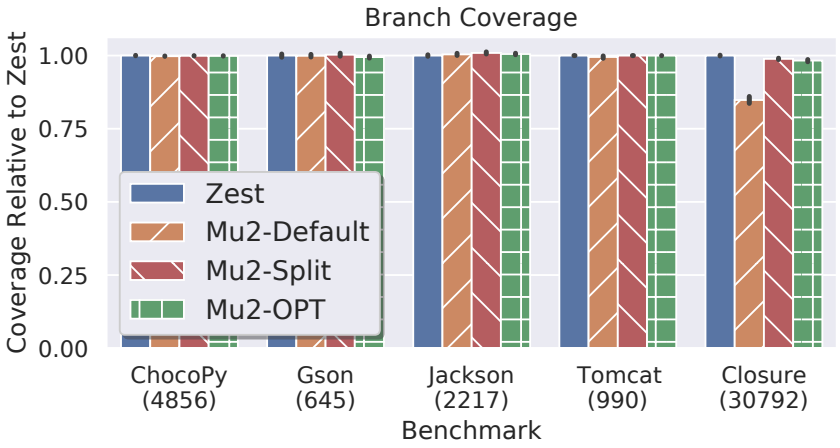


Figure 6.6: Branch coverage across all benchmarks normalized to the mean coverage achieved by Zest. The number of branches covered by Zest (used to normalize) is listed below each target. Error bars represent 95% confidence intervals.

RQ1: *Does mutation-analysis guidance produce a higher quality test-input corpus than coverage-only feedback in greybox fuzzing?*
RQ2: *How do the performance optimizations impact the quality of the test-input corpus produced by mutation-analysis guidance?*
RQ3: *How does the reliability of killing nontrivial mutants differ between mutation-analysis guidance and coverage guidance?*
RQ4: *How much stronger is the differential mutation testing oracle than the implicit oracle?*

Benchmarks We consider five real-world Java programs:[2]

1. ChocoPy [47, 161] reference compiler (˜6K LoC): The test driver (reused from [27]) reads in a program in ChocoPy (a statically typed dialect of Python) and runs the semantic analysis stage of the ChocoPy reference compiler to return a type-checked AST object.
2. Gson [171] JSON Parser (˜26K LoC): The test driver parses a input JSON string and returns a Java object output.
3. Jackson [172] JSON Parser (˜49K LoC): The test driver acts similar to that of Gson.
4. Apache Tomcat [173] WebXML Parser (˜10K LoC): The test driver parses a string input and returns the WebXML representation of the parsed output.
5. Google Closure Compiler [90] (˜250K LoC): The test driver (reused from [19] and [27]) takes in a JavaScript program and performs source-to-source optimizations. It then returns the optimized JavaScript code.

[2]While we note lines of code (LoC) for completeness, only a fraction of this code is reachable from fuzz drivers. Fig. 6.6 indicates actual code coverage.

Mutation selection Following previous work on semantic fuzzing [19, 27], we filter on package names to identify classes relating to the core logic of the program under test. The mutation operators are then applied on these classes. We use the same generators, oracles, and filters for both Zest and Mu2. All of the test drivers return objects that override `Object.equals`, and were thus properly compared by the differential oracle.

Duration Following best practices [29], we use a time bound of 24 hours for each experiment.

Repetitions To account for the randomness in fuzzing, we run each experiment 20 times and report statistics.

Metrics For our evaluations, we compute the branch coverage and mutation scores across each fuzzer-generated test-input corpus. We report mutation scores as the absolute *number of mutants killed* instead of as a fraction (ref. Section 3.2.1), since we only care about comparing these numbers across fuzzing variants, and since the denominator is meaningless when considering a single test entry point. We additionally compute the kill frequency of each of the nontrivial mutants across the repetitions. When reporting statistical significance, a Mann-Whitney-U test was performed with $\alpha = 0.05$.

Default variant Unless explicitly qualified with an aggressive optimization strategy, the default variant of Mu2 used in our evaluation only uses sound optimizations described in Section 3.2.5.

Reproducibility and Data Availability We have published a replication package and evaluation data at: `https://doi.org/10.5281/zenodo.7647828`. The evaluation data contains logs of fuzzing campaigns used to generate all evaluation figures and tables [174].

RQ1: Test-Input Corpus Quality

> *Does mutation-analysis guidance produce a higher quality test-input corpus than coverage-only feedback in greybox fuzzing?*

RQ1 focuses on evaluating mutation-analysis-guided fuzzing with a fixed time budget. Higher mutation score from the Mu2-produced corpus and comparable coverage results would demonstrate that mutation-analysis can be used as an off-the-shelf replacement for coverage-only guidance. We first discuss results for Mu2-Default, then evaluate two variants against the Zest baseline.

Figure 6.5 visualizes the mutation scores for each fuzzer-generated corpus. The default mutation-analysis guidance (Mu2-Default) is able to produce a corpus with higher mutation scores than coverage-only feedback in the first three benchmarks, achieving statistically significant increases in all three. Additionally, Figure 6.6 shows equivalent branch coverage

between Zest and Mu2 for these benchmarks. For the Tomcat WebXML parser, the number of killed mutants saturated at 239 in almost all of the repetitions of the fuzzing campaigns. For the Closure Compiler, our largest benchmark, the Mu2-Default corpora achieve, on average, approximately 17% less branch coverage than Zest (shown in Figure 6.6). This is likely due to the performance overhead of running mutation analysis for a large benchmark, and also likely accounts for the Zest corpora on average killing 12 more mutants than Mu2-Default, as covering code is a necessary condition for killing mutants in that part of the code. This suggests Mu2-Default may not scale well to very large programs.

One way to mitigate this slowdown is to add mutation-analysis feedback to coverage-guided fuzzing *later* in the campaign. The Mu2-Split variant utilizes coverage-only feedback for the first half of the campaign (which is very efficient) and then introduces expensive mutation-analysis feedback for the second half. This is based on an idea by Gopinath et al. [72], who suggested saturating coverage before adding mutation analysis to the fuzzing loop. The Mu2-Split-generated corpora show statistically significant increases in mutation score over Zest for the first 4 benchmarks (Fig. 6.5), although the effect for Tomcat is very small. There is also a major improvement over Mu2-Default in the Closure benchmark; Mu2-Split is able to bridge the gap in coverage (Fig. 6.6) and mutation scores (Fig. 6.5) that Mu2-Default had with the Zest baseline.

Another method of scaling Mu2 is to apply the aggressive optimizations detailed in Section 3.2.5. Mu2-OPT is a particular variant we chose that applies the *k-Least-Executed* filter with $k = 10$ mutants. The Mu2-OPT generated corpus similarly achieves statistically significant increases in mutation scores across the first four benchmarks over Zest, with up to 20% increase in the Jackson JSON parser (Fig. 6.5). There is no significant difference between the mutation scores of Mu2-OPT and Zest on the Closure Compiler. Mu2-OPT achieves slightly less coverage than Zest on two benchmarks (ChocoPy and Closure) and more on one (Jackson)—however, the differences are fairly small (below 2%).

We are also curious about whether the additional saving of mutant-killing inputs in Mu2 may bloat the size of the generated test-input corpus, impacting its use in regression testing. Table 6.5 displays the average sizes and runtimes for each fuzzer-generated corpus and show that no such bloat occurs in Mu2. While there are some differences in the number of test inputs, the runtime of the Mu2-produced corpora are not significantly higher than those produced by Zest. Thus, mutation-analysis-guided fuzzing is able to produce a higher quality test-input corpus and can be feasibly used for regression testing.

We believe that an aggressively optimized version of mutation-analysis-guided fuzzing can be used as a replacement for coverage-guided fuzzing if the goal is to produce a test input corpus with high mutation score. Mu2-OPT provides an improvement for 4 benchmarks and scales to the largest target without paying a performance penalty.

RQ2: Aggressive Optimizations

> *How do the performance optimizations impact the quality of the test-input corpus produced by mutation-analysis guidance?*

This RQ focuses on understanding the benefit of the *aggressive* optimizations in miti-

Table 6.5: Average number of test inputs (and average runtime, in parentheses below) of fuzzer-generated corpora. Corresponding standard deviations also listed.

	Zest	**Mu2-Default**	**Mu2-Split**	**Mu2-OPT**
ChocoPy	864 ± 34 $(18.6\,\text{s} \pm 2.1\,\text{s})$	711 ± 47 $(9.8\,\text{s} \pm 0.8\,\text{s})$	725 ± 36 $(11.7\,\text{s} \pm 1.3\,\text{s})$	746 ± 40 $(10.4\,\text{s} \pm 1.0\,\text{s})$
Gson	467 ± 18 $(1.7\,\text{s} \pm 0.1\,\text{s})$	461 ± 17 $(1.7\,\text{s} \pm 0.1\,\text{s})$	469 ± 15 $(1.7\,\text{s} \pm 0.0\,\text{s})$	489 ± 21 $(1.7\,\text{s} \pm 0.0\,\text{s})$
Jackson	598 ± 19 $(2.4\,\text{s} \pm 0.1\,\text{s})$	655 ± 16 $(2.4\,\text{s} \pm 0.1\,\text{s})$	641 ± 19 $(2.4\,\text{s} \pm 0.0\,\text{s})$	673 ± 15 $(2.4\,\text{s} \pm 0.1\,\text{s})$
Tomcat	138 ± 7 $(2.6\,\text{s} \pm 0.1\,\text{s})$	122 ± 6 $(2.5\,\text{s} \pm 0.1\,\text{s})$	136 ± 6 $(2.6\,\text{s} \pm 0.1\,\text{s})$	171 ± 5 $(2.7\,\text{s} \pm 0.1\,\text{s})$
Closure	4885 ± 205 $(554\,\text{s} \pm 82\,\text{s})$	1075 ± 219 $(58\,\text{s} \pm 13\,\text{s})$	4044 ± 146 $(360\,\text{s} \pm 27\,\text{s})$	4037 ± 192 $(353\,\text{s} \pm 30\,\text{s})$

Table 6.6: Geometric mean of speedups achieved by each aggressively filtered variant of Mu2 over Mu2-Default. 20/10/5 refer to the sizes of the filtered subset of mutants.

	Random (20/10/5)	LeastExecuted (20/10/5)
ChocoPy	1.1/1.7/2.8×	1.1/1.6/2.5×
Gson	0.9/1.1/1.3×	1.0/1.2/1.3×
Jackson	1.0/1.1/1.3×	1.1/1.0/1.2×
Tomcat	3.4/3.5/8.2×	2.3/6.0/7.9×
Closure	10.4/13.8/21.3×	13.8/19.2/24.9×

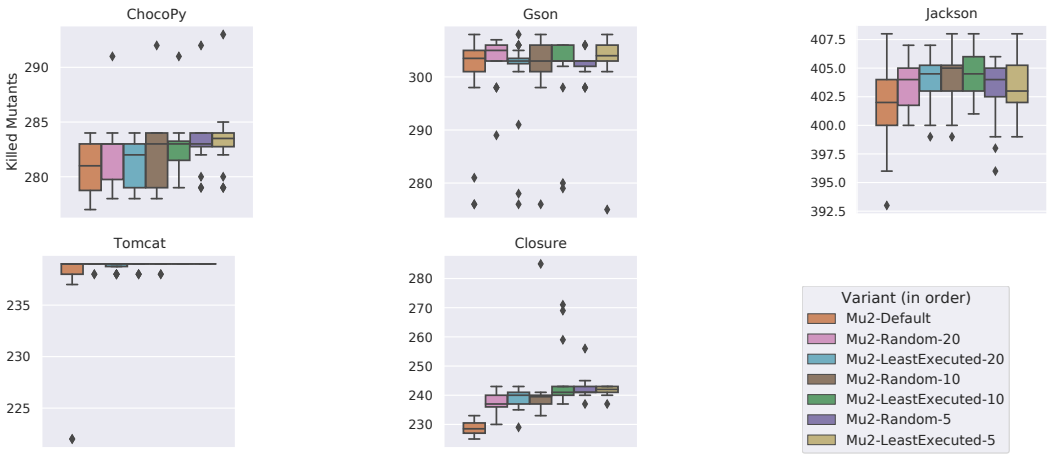


Figure 6.7: Box plots showing number of killed mutants by each aggressively optimized variant of Mu2 and the default, across 20 repetitions of 24 hour campaigns (higher is better).

gating the scalability concerns of Mu2-Default. We created variants Mu2-LeastExecuted-k and Mu2-Random-k, each applying the corresponding filter described in Section 3.2.5, and chose three different values of $k \in \{5, 10, 20\}$.

First, we measure just the performance benefit. Table 6.6 shows the speedups achieved—in terms of number of inputs evaluated over a 24-hour period—by each variant over Mu2-Default. The improvement for the benchmarks Gson and Jackson is relatively minor due to the already small number of mutants executed for each input after applying the execution and infection optimizations (ref. Section 3.2.5 and Table 3.1). However, the aggressive optimizations provide significant improvement for the larger benchmarks, with almost 25× speedup for the Mu2-LeastExecuted-5 variant on the Closure benchmark. This makes sense, as the main purpose of aggressive optimizations is to enable scaling to large programs.

Due to the aggressive nature of the mutant filtering, it is possible that input candidates that do kill mutants are not saved simply because those killable mutants were filtered. To determine whether the speedup actually results in a test-input corpus with higher mutation score, we must also measure the impact of these optimizations on the mutation score of the generated corpus.

Figure 6.7 displays the mutation scores of all of the variants for each of the 5 benchmarks. At least one optimized variant was better than the default in all benchmarks. Somewhat surprisingly, we observe similar mutation scores between the Mu2-LeastExecuted-k and Mu2-Random-k variants for the same value of k in the first four benchmarks. The one exception is Closure Compiler, where Mu2-LeastExecuted-10 achieves a statistically significantly higher mutation score than Mu2-Random-10. Again, the effect of aggressive optimizations is most pronounced in the largest target.

Another interesting observation is that we can visualize the trade-off between execution speed and mutation score in the Jackson benchmark: although the Mu2-Random-5 variant has a faster execution speed than Mu2-Random-10 (Tab. 6.6) due to the smaller number of mutants, the mutation score slightly decreases (Fig. 6.7) since the optimization might skip some mutants at the wrong time. Nonetheless, the speedup displayed by the variants for the Closure Compiler results in better test-input corpus quality. All of the Mu2 variants are able to achieve statistically significantly higher mutation scores than Mu2-Default. Specifically, Mu2-LeastExecuted-10, Mu2-Random-5, and Mu2-LeastExecuted-5 kill ∼15 more mutants on average than Mu2-Default.

We found that Mu2-LeastExecuted-10 and Mu2-LeastExecuted-5 were the strongest variants, as they had a statistically significant increase in mutation score over Mu2-Default in the most benchmarks (3 out of 5) out of all variants. There was no significant difference in mutation scores between these two variants in any benchmarks, so we arbitrarily picked Mu2-LeastExecuted-10 as the optimized version of mutation-analysis-guided fuzzing (Mu2-OPT) in our evaluation of RQ1 and RQ4. We do however note for future practitioners that the best aggressively optimized variant of Mu2 may change depending on the target program.

RQ3: Nontrivial Mutants

How does the reliability of killing nontrivial mutants differ between mutation-analysis guidance and coverage guidance?

Not all mutants are equal—some mutants are easier to kill than others. We define a mutant $P' = \langle P, e, e', n \rangle$ as *trivial* if it is killed by the first input that executes n in every experiment (this is the dynamic version of Kaufman et al.'s definition [175]). Since trivial mutants are killed as soon as the corresponding code is covered, conventional coverage-guided fuzzing like Zest suffices to capture them. On the other hand, since nontrivial mutants may or may not be killed even after the mutated expression is covered, we are interested to know whether these get killed based on pure luck or whether these get killed *reliably* across repetitions potentially due to the guidance in the fuzzing algorithm. We measure *reliability* by counting the number of repetitions in which each mutant is killed. In particular, we study the *difference in reliability* of killing nontrivial mutants between Zest and the best variant of Mu2.

Figure 6.8 is a histogram showing the difference in kill rate of nontrivial mutants between Mu2-OPT and Zest. The values on the right side (green) correspond to mutants killed more reliably by Mu2-OPT than Zest. For the sake of visualization, the mutants with no difference in kill rate (X-axis value 0) are excluded from the charts.

Mu2-OPT is able to achieve a significantly higher kill frequency of nontrivial mutants in ChocoPy and Jackson. In fact, there are 29 mutants in Jackson that are killed during *all* repetitions of Mu2-OPT and *zero* repetitions of Zest. This is a strong indication that mutation-analysis feedback can consistently discover mutant-killing inputs that coverage-only feedback is incapable of finding. For the Gson parser, there are 22 vs. 24 nontrivial mutants killed more reliably by Zest and Mu2-OPT respectively, though the X-axis values are generally higher for Mu2-OPT. For Closure, there are over 60 mutants killed by at least one more repetition of Mu2-OPT compared to the 4 by Zest. Overall, Mu2-OPT is able to kill nontrivial mutants more reliably than Zest.

We also note that Figure 6.8 provides some insight into the diversity of mutants, particularly *redundant* mutants. By definition, redundant mutants are grouped together in the same bars since they are always killed at the same frequency. Flattening the size of each bar to 1 removes at least all redundant mutants and acts as a lower bound on the number of nonredundant mutants.

RQ4: Differential Mutation Testing

How much stronger is the differential mutation testing oracle than the implicit oracle?

Described in Section 3.2.4, the differential mutation testing oracle is responsible for determining whether an input kills a mutant by comparing the outputs of the executions. We contrast it with the incomplete greybox fuzzing implicit oracle, which only detects uncaught exceptions or failed property checks. To study the strength of the differential oracle, we evaluate the improvement in the number of killed mutants over the implicit oracle.

Figure 6.9 shows the difference in mutant kills across the benchmarks with the two types

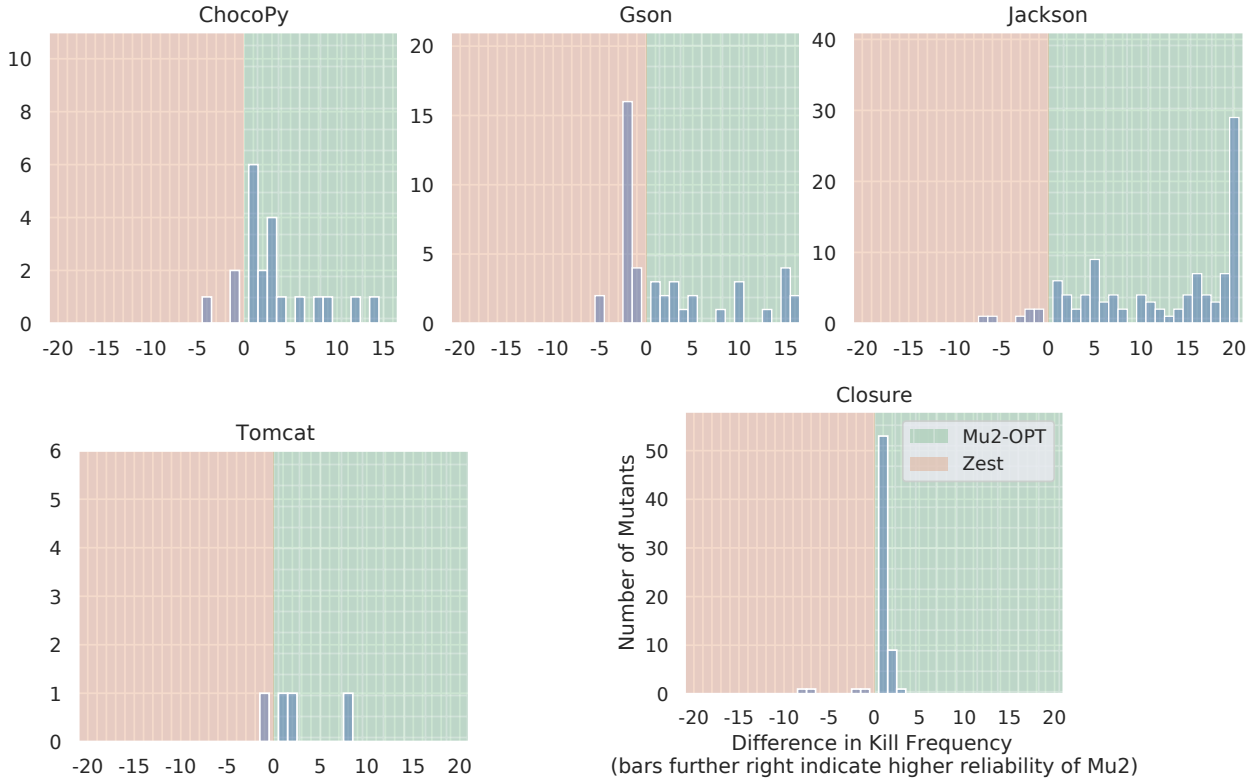


Figure 6.8: Histogram of difference in kill-rate of *nontrivial* mutants between Zest and Mu2-OPT over 20 experiments. X-axis is the difference in repetitions (ranging from -20 to 20), and Y-axis is the number of mutants. Larger positive differences (right) are better for Mu2-OPT, and larger negative differences (left) are better for Zest.

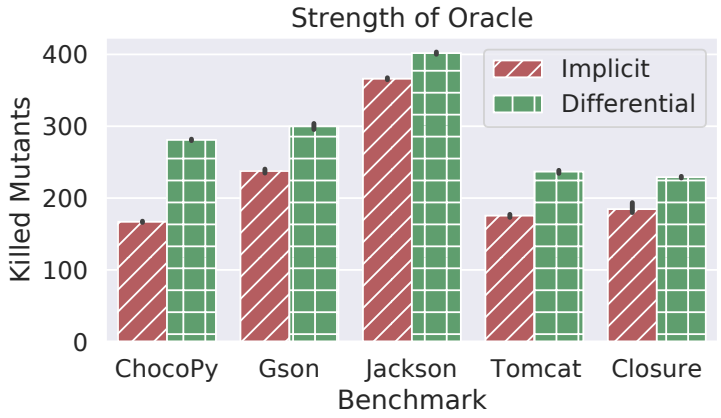


Figure 6.9: Number of killed mutants detected by the differential oracle vs. the implicit oracle across 20 repetitions of 24 hour campaigns with Mu2-Default (higher is better). Error bars represent 95% confidence intervals.

of oracles. The differential oracle is able to kill a significantly higher number of mutants across all 5 benchmarks, with an average increase of 25%. In the ChocoPy benchmark, over 85 more mutants are caught! This is because certain mutants are *unkillable* by the implicit oracle due to their effect on program behavior. We describe one of these mutants below. For brevity, we describe the code functionality, omitting the actual code snippet.

The ChocoPy type-checker has a function to check that the left and right operand types of an expression match when using the "+" operator. If so, the type is returned and assigned to the corresponding expression node in the output AST; otherwise, an error message for the expression is added to the output AST error list. Consider a mutant P' that modifies this function to return `null` instead of the correct type. Executing P' on the well-typed ChocoPy program `[1]+([2]+[3])` results in a type-checking error, since the `[int]` type of `[1]` does not match the mutated `null` return type of `([2]+[3])`. The differential oracle kills P' since the output AST produced by P' contains a type error, whereas P does not. The implicit oracle fails to kill this mutant since no exceptions are triggered.

We conclude that the differential oracle is substantially stronger than a traditional implicit oracle and is valuable for capturing a larger set of mutant program execution behaviors.

Appendix C: Evaluation of Proptest-AI

Based on our proposed evaluation methodology, we structure our evaluation around three research questions:

RQ1: *Are LLMs able to synthesize valid and sound property-based tests of API methods when provided documentation?*

RQ2: *Is the execution-based soundness metric for property assertions aligned with human judgment of soundness?*

Table 6.7: Modules selected for Proptest-AI evaluation. Selected modules consist of native Python libraries (e.g. `datetime` and `statistics`) and third-party libraries (e.g. `networkx` and `numpy`). The number of API methods selected from each library as well as average token length of the API method documentation, using the OpenAI tokenizer.

Library	**# API Methods**	**Documentation Length (tokens)**
`dateutil`	2	643.5 ± 155.5
`html`	2	81.5 ± 5.5
`zlib`	3	312.7 ± 116.8
`cryptography.fernet`	3	458.7 ± 100.9
`datetime`	4	212.8 ± 83.2
`decimal`	5	122.2 ± 68.8
`networkx`	5	592.8 ± 439.6
`numpy`	5	766.2 ± 242.4
`pandas`	5	1197.8 ± 569.7
`statistics`	6	318.6 ± 149.5
Total	40	-

RQ3: *Are LLMs able to synthesize property-based tests that cover documented properties of API methods?*

Experimental Setup

Models We chose three state-of-the-art language models for our evaluation: OpenAI's GPT-4 [176], Anthropic's Claude-3-Opus [114], and Google's Gemini-1.5-Pro [113]. [3]

API Methods We selected API methods from 10 different Python libraries to generate property-based tests across a variety of tasks and input data types. Table 6.7 displays the libraries and the number of selected API methods for each library. The libraries include native Python libraries as well as popular third-party libraries such has `pandas` and `numpy`. All selected API methods are deterministic, as this is necessary for property-based testing. The API method documentation was extracted from the online documentation for each API method. In total, we selected 40 API methods. The full list of API methods and documentation is provided in the data artifact.

Approaches We evaluate two methods of prompting the LLM to generate a property-based test shown in Figure 4.3. (1) A single-stage method to generate a single test function and (2) a two-stage method to extract properties and generate a property-based test suite of test functions. With three models and two prompting methods, we have a total of 6 approaches.

[3]We initially included Codellama-34B in our models, but preliminary experiments showed that the validity of generated PBTs was too low to warramt further evaluation.

Table 6.8: Proptest-AI synthesized valid and sound test functions across all API methods. 41.74% of the test functions synthesized using the two-stage prompting approach with GPT-4 achieve 100% validity and 100% soundness.

Model	Approach	Total Test Functions	Valid Test Functions	Valid and Sound Test Functions
Claude-3-Opus	Single Stage	215	53 (24.65%)	16 (7.44%)
	Two Stage	931	423 (45.43%)	289 (31.04%)
Gemini-1.5-Pro	Single Stage	220	39 (17.72%)	25 (11.36%)
	Two Stage	799	209 (26.16%)	124 (15.51%)
GPT-4	Single Stage	212	97 (45.75%)	54 (25.47%)
	Two Stage	733	**400 (54.57%)**	**306 (41.74%)**
Total	-	3,110	1,221 (39.26%)	814 (26.17%)

Samples For each approach, we sample the LLM five times with a temperature of 0.7. In total, we synthesized 1,200 samples across all API methods and approaches. Out of these samples, only 16 either contained no Python or were syntactically invalid Python code.

RQ1: Validity and Soundness

High validity and soundness for property-based tests are important for ensuring that the test can run without errors (e.g., calling a nonexistent API or missing an import for a library). For RQ1, we report the validity and soundness of Proptest-AI synthesized PBTs across all API methods, models, and approaches. To measure validity, we call each property-based test function 1,000 times and check for non-assertion errors; for soundness, we check for assertion errors.

Table 6.8 shows the percent of valid and sound test functions across all API methods for the single-stage and two-stage approaches. We find that the two-stage prompting achieves significantly higher validity than the single-stage generation for all models. This is likely due to the test functions in the suite being smaller and only testing one specific property, thus having a smaller chance of hallucination. Similarly, the soundness of test functions synthesized using the two-stage prompting is much higher.

Overall, GPT-4 achieves the highest validity and soundness for both single-stage and two-stage prompting. The two-stage approach with GPT-4 is able to synthesized a valid and sound test function within 2.4 samples on average. Figure 6.10 shows the distribution of valid and sound test functions from GPT-4 across our 10 different libraries. We observe the performance varying across libraries, with validity and soundness much higher on libraries such as `datetime` and `zlib`.

We additionally report average soundness over each test function as the percentage of 1,000 invocations that do not result in any assertion errors. Figure 6.12 shows the this distribution over all synthesized test functions. Although only 814 out of 1,221 valid test functions achieve 100% soundness (Table 6.8), this distribution shows that the soundness of many of these tests functions close to 100%. This suggests that the property assertions are correct for most inputs and outputs, but may fail to capture potential edge cases.

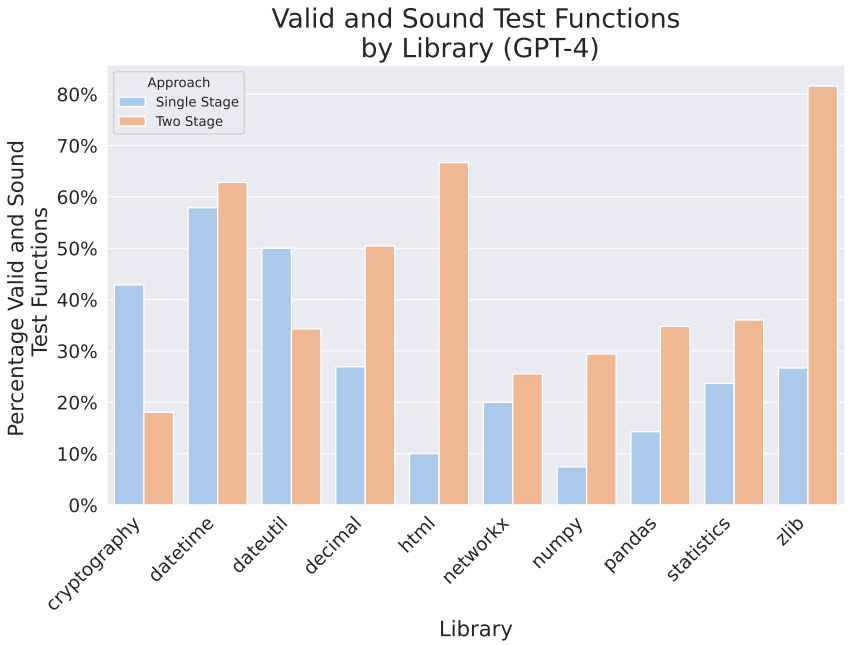


Figure 6.10: Distribution of valid and sound property-based test functions for GPT-4 across all libraries. Certain libraries such as `networkx` and `pandas` have a smaller percentage of valid and sound test functions than `zlib`.

Table 6.9: Human labels of individual property assertions in LLM samples.

Model	**Sound**	**Unsound**
Claude-3-Opus	64	23
Gemini-1.5-Pro	23	5
GPT-4	86	18
Total	173	46

The best Proptest-AI approach with two-stage prompting and GPT-4 is able to synthesize valid and sound property-based test functions in **2.4 samples** on average.

RQ2: Soundness Metric Alignment

Our soundness metric reported in RQ1 provides an execution-based measurement for property assertions. However, it is possible that certain property assertions in the test are not executed due to certain program paths taken during execution. To further validate the soundness of individual property assertions, we performed a manual labeling of a sample of synthesized property-based test assertions. Out of all valid property-based test samples, half were selected, evenly distributed for each library. Raters read the documentation for the corresponding API method and labeled the first five assertions as sound or unsound.

Table 6.9 displays the results of the labeling. We note that the difference in the number

of labeled assertions per model is due to the difference in validity and the number of property assertions in each test. We find that out of 219 labeled assertions, 173 (79%) were sound. The soundness of assertions from each model were all similar, ranging from 74–83%. To ensure the reliability of this manual labeling process, we computed the inter-rater agreement between two raters using Cohen's kappa coefficient (κ). The resulting κ value of 0.862 ± 0.133 indicates a high level of agreement between the raters, suggesting that the labeling process was consistent.

We use our property assertion labels to determine soundness for each sample—if any assertions were unsound, the sample was labeled as unsound. We compared these labels to the labels resulting from our soundness metric and found that our metric has a precision of 100% and recall of 97.14%. There was only one false negative determined from the soundness metric due to the sample containing an unsound assertion that was not executed with an edge-case input in the 1,000 invocations.

When labeling property-based tests, we noticed a variety of soundness issues, ranging from hallucination of properties outside of the API documentation to assertions only failing due to very specific edge cases. The first test in Figure 6.11 is an example in which a property assertion checks that when there is no cycle, the number of edges in a graph is less than the number of nodes. This property is true for undirected graphs, but not for directed graphs, which can be generated as inputs in this PBT.

Another general pattern we noticed was that the property assertions held for *most* of the generated inputs, but did not account for certain edge cases. This explains why many test functions have soundness of above 80% in the distribution shown in Figure 6.12. An example of this is the second test in Figure 6.11. From the API docs, a property of the ISO calendar is that the "first week of an ISO year is the first (Gregorian) calendar week of a year containing a Thursday." This PBT does not account for the case in which the first week of the Gregorian year does *not* contain a Thursday, which occurs for a small percentage of generated inputs.

Through manual labeling on a set of LLM samples, we find that our soundness metric is aligned with human judgment, achieving **100% precision and 97% recall.** We observed that unsound assertions arose from hallucination of properties and failures to handle edge cases.

RQ3: Property Coverage

RQ3 focuses on evaluating whether LLM-synthesized PBTs can detect property violations. Tests can be valid and sound while containing very simple assertions. *Property coverage*, described in Section 4.3.2, measures the ability of the property-based test to kill property mutants over all properties. This provides a measurement of the completeness of synthesized property-based test. Following the steps detailed in Section 4.3.2, we prompt GPT-4 to extract five properties from the documentation for each API method and generate 5 property mutants for each property. In total, there are 200 properties we use to calculate overall property coverage.

```python
import ...

# Test for networkx.find_cycle
@given(st.data())
def test_find_cycle(data):
    (...) # Generator logic for graph G
    # verify the properties
    try:
        cycle = nx.find_cycle(G)
    except nx.exception.NetworkXNoCycle:
      # The graph doesn't have enough edges to form a cycle
        assert nx.number_of_edges(G) < num_nodes

# Test for date.isocalendar
@given(y=st.integers(min_value=1, max_value=9999), m=st.integers(min_value=1, max_value=12),
       d=st.integers(min_value=1, max_value=31))
def test_date_isocalendar(y, m, d):
    try:
        d = date(y, m, d)
        iso_year, iso_week, iso_weekday = d.isocalendar()

        assert iso_week >= 1 and iso_week <= 53
        assert iso_weekday >= 1 and iso_weekday <= 7
        assert iso_year == y or (iso_week == 1 and iso_year == y + 1)
    except ValueError:
        pass
```

Figure 6.11: Example snippets from GPT-4 synthesized unsound property-based tests for `networkx.find_cycle` and `datetime.date.isocalendar`. The second property assertion in `test_find_cycle` checks a general property of graphs that was not included in the API documentation that may be unsound when the graph is directed. The last assertion in `test_date_isocalendar` performs a comparison to the Gregorian year, but does not account for the case in which the first week of the Gregorian year does not contain a Thursday.

We first measure *property mutation score* over all valid and sound LLM. The tests must be sound to ensure that property mutants are only killed due to assertions checking the property mutant, rather than errors in the original test. We additionally filter any mutants that are killed due to validity errors to specifically measure the completeness of the property assertions. For each approach, we report the property mutation score as the average percent of property mutants killed over all samples. Table 6.10 shows property mutation score for each approach as well as the total number of mutants that were executed.

Overall, we observe that the valid and sound samples are able to kill 40–80% of property mutants. These results vary across libraries, as shown in Figure 6.13. Samples achieve higher property coverage in native Python libraries such as `datetime` and `decimal`. `networkx` requires stronger assertions that are more difficult to synthesize due to the complexity of graph properties.

We finally calculate the property coverage of all approaches on the entire set of 40 Python APIs. With five samples of an LLM for a synthesized PBT, how many properties

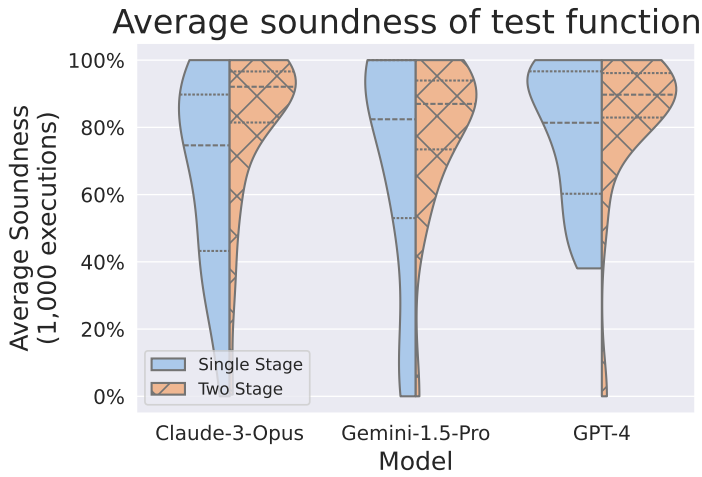


Figure 6.12: Violin plot displaying soundness distribution of valid test functions, aggregated across all API methods (higher is better). Results for both single-stage and two-stage approaches are shown for each model. Overall, GPT-4 with two-stage prompting achieves the highest average soundness of 87% over all valid test functions.

can be covered? For a property to be covered, the sample must be valid, sound, and kill at least one property mutant. This measures the overall ability of the LLM to synthesize a PBT achieving all of our desired properties.

Table 6.11 shows the property coverage for each of our approaches. The best approach of two-stage prompting with GPT-4 is successfully able to synthesize property-based tests that are valid, sound, and cover 20.5% of properties. We believe this is a promising result, as Proptest-AI is able to automate a significant portion of the property-based testing writing process and cover extracted docuemntation properties.

The two-stage prompting approach with GPT-4 is successfully able to automatically synthesize PBTs covering **20.5% of documented properties**.

Appendix D: Havoc Paradox Evaluation

Preliminary Results

The first goal of our preliminary evaluation is to investigate if the havoc effect is present and measurable in Zest. We then also investigate whether the structure-preserving mutations in EI indeed reduce the havoc effect in parametric generators. Finally, we look at some preliminary coverage measurements on our selected fuzzers.

Benchmarks We selected 7 different benchmark programs with 4 different program generators used by prior research [19, 94]. Table 6.12 shows the detailed characteristics, including the benchmark name, the generator used to generate input data, the lines of code

Table 6.10: Property coverage of each model and prompting approach across all valid and sound LLM samples. Five property mutants were generated for each API, and mutants with runtime errors were filtered out.

Model	Approach	Property Mutation Score	Total Mutants
Claude-3-Opus	Single Stage	58.02%	293
	Two Stage	59.69%	258
Gemini-1.5-Pro	Single Stage	43.57%	140
	Two Stage	62.70%	244
GPT-4	Single Stage	79.78%	89
	Two Stage	51.88%	480

Table 6.11: Property coverage of LLM samples over all 40 API methods. In total, there were 200 properties across all API methods. If any of the five samples kills a property mutant, the property is covered.

Model	Approach	Property Coverage
Claude-3-Opus	Single Stage	9%
	Two Stage	13%
Gemini-1.5-Pro	Single Stage	5.5%
	Two Stage	12%
GPT-4	Single Stage	7%
	Two Stage	**20.50%**

of the benchmark program, and the lines of code of the input generator.

Measuring the Havoc Effect in Zest

To understand how destructive the mutations are in Zest, we measure how different mutated inputs are from their parents. To evaluate this, we choose to compute the normalized Levenshtein distance between a mutated input and its parent. We call this *mutation distance*. When the mutant and its parent are the same, the normalized mutation distance is 0. The greater the distance is, the more different the mutant is from its parent. In our measurements, we noticed a number of inputs with 0 mutation distance. We exclude these mutants from our analysis as they are not mutants and could trivially be filtered out by the fuzzer.

Figure 6.14 compares the distribution of normalized mutation distances for all inputs generated by Zest (left, orange) and the inputs saved to the fuzzing corpus (right, lavender). This allows us to compare the distribution of all generator-produced inputs to those inputs that are useful for the fuzzing process (i.e., the saved inputs).

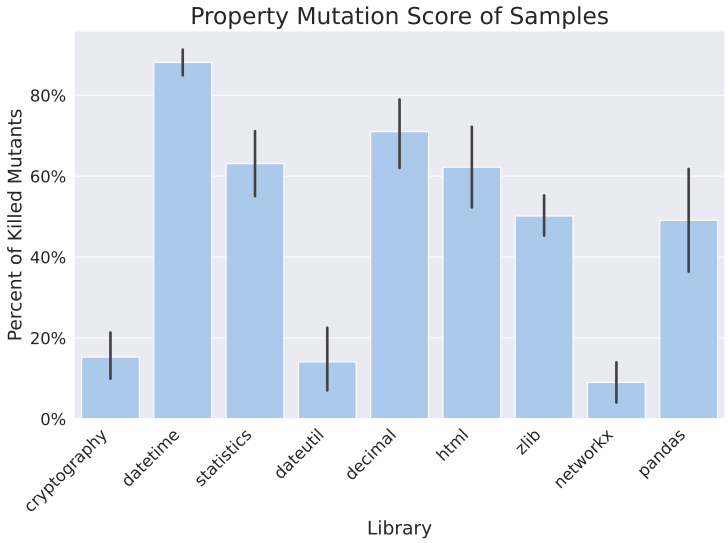


Figure 6.13: Distribution of property mutation score over libraries across all valid and sound tests (higher is better). Generally, the tests achieve higher property mutation score from native Python libraries such as `statistics` and `datetime`. Property mutants from `networkx` require stronger assertions to detect.

Table 6.12: Characteristics of our benchmark applications.

Benchmark	Generator	LOC	Gen-LOC
Ant [177]	XML	140k	136
Maven [178]	XML	93K	136
Rhino Compiler [179]	JavaScript	110K	250
Closure Compiler [90]	JavaScript	250K	250
Chocopy [161]	Python	6K	397
Gson [171]	Json	26K	89
Jackson [172]	Json	49K	89

We observe significant variations in the magnitude of mutation distances across different benchmarks. However, benchmarks with the same generator have similar distributions, underscoring the impact of the generator's implementation on the havoc effect.

For simple targets like gson and jackson, which parse JSON strings into Java objects, the havoc effect appears to be beneficial, as a higher frequency of saved inputs has large mutation distances. Therefore, we do not expect structure-preserving mutations to improve the fuzzing performance for these benchmarks significantly.

The mutation distance distribution of all inputs for ant and maven, closely mirrors that of the saved inputs, suggesting that parametric generators do a good job of generating mutants for these benchmarks. Unlike the other benchmarks, we see no peak in the Kernel Density Estimate (KDE) curves at high mutation distances, suggesting the havoc effect is not as present for the XML generator these benchmarks share. In addition, the mutation distance for all inputs exhibits a denser cluster in the lower range, further suggesting that

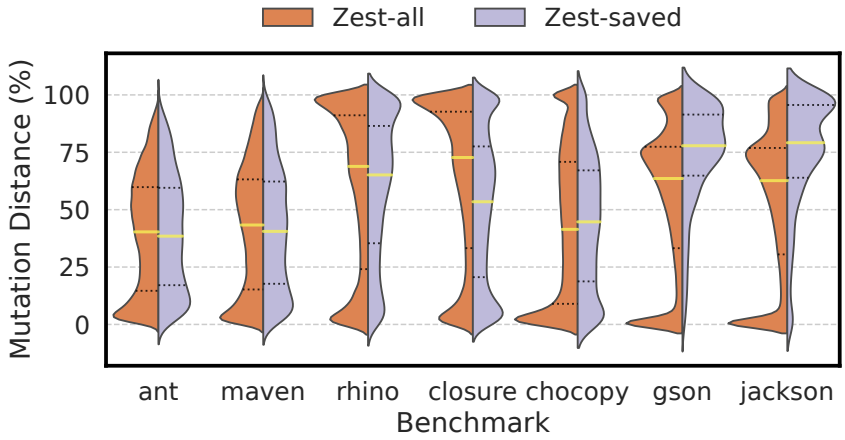


Figure 6.14: Mutation distance distributions for all inputs (left, orange half) and saved inputs (right, lavender half) generated by Zest. A wider area means a higher frequency of that mutation distance. The yellow line highlights the median mutation distance, and the dotted lines are the quartiles.

structure-preserving mutations may not affect these benchmarks.

For more complex targets such as rhino and closure, the frequency curves for all inputs exhibit two distinct peaks—one at extremely small mutation distances and another at large mutation distances. This dual-peaked pattern shows that rhino and closure are affected by the havoc effect as the small byte-level mutations in Zest result in many inputs with large mutation distances. For closure, the median mutation distance for saved inputs is significantly lower than that for all inputs. The KDE curve for saved inputs in closure exhibits a higher peak at smaller mutation distances, showing a greater density of produced inputs with smaller mutations. Thus, closure may benefit more the structure-preserving mutations.

Can We Reduce the Havoc Effect?

Is the havoc effect inherent to all generator-based greybox fuzzers, or does it differ? We conduct a preliminary evaluation of this by comparing the mutation distances of a Zest to EI, which should suffer less from the havoc effect. In Figure 6.15, we plot the mutation distance distribution for all inputs generated by Zest and EI. Figure 6.15 shows that the median mutation distances for all inputs generated by EI are much lower than Zest. This is in line with our expectations, where EI can more effectively localize mutations. Notably, for chocopy, gson, and jackson, the EI part of the violins peak at the bottom, demonstrating a denser cluster of inputs with small mutation distances. The inputs generated by EI for ant and maven have a similarly high frequency of small mutation distances. Though EI's mutation distance distributions for rhino and closure are bimodal, the KDE curves have a higher peak at the smaller mutation distances. In contrast, the KDE curves for Zest do not always reach the highest peak at the bottom and have multiple modes across a wide range of mutation distances. Our findings confirm that the EI-based approach indeed alleviates the destructiveness of mutations witnessed in stream-based approaches such as Zest. More

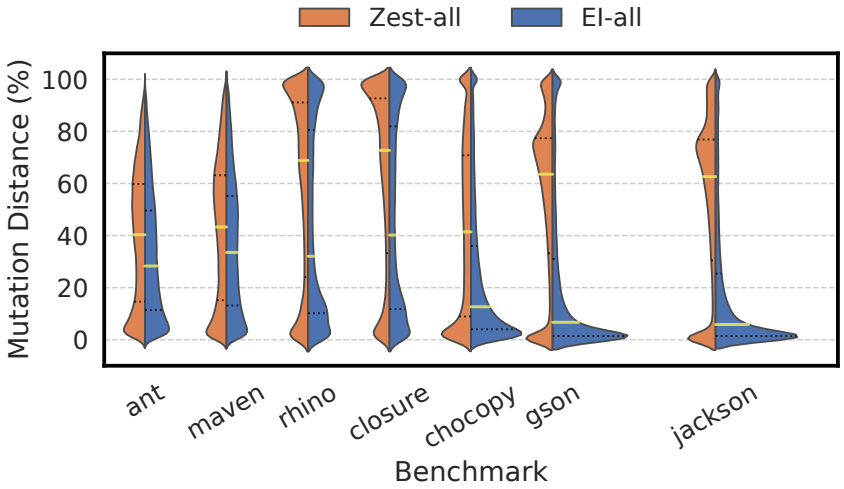


Figure 6.15: Distribution of mutation distances for all inputs generated by Zest and EI. EI generates inputs with small mutation distances at higher frequencies for all benchmarks.

Table 6.13: The average branch coverage and execution count for each fuzzer (rounded to the nearest thousand) in application classes for Closure across 20 fuzzing campaigns after 24 hours.

	Zest	EI	Zeugma	BDF(S)	BDF(D)
Branch Coverage	24,203	24,252	24,990	23,809	23,574
# Executions ($\times 10^3$)	5,542	5,555	10,413	4,687	5,009

importantly, this finding is consistent across different benchmarks and generators.

Zero Mutations. As aforementioned, an unexpected observation gleaned during our preliminary evaluation was the frequent occurrence of inputs with 0 mutation distance. We refer to these mutations where the mutated input is identical to the parent input as the *zero mutations*. We removed zero mutations from our earlier analysis: a duplicate input will never be saved and should ideally be filtered out by the fuzzing algorithm.

Zero mutations occur when the fuzzer's mutation to the input bytestream does not change the generator-produced input, effectively creating a duplicate input. For example, in Figure 2.3a, where altering the first byte to either `00` or `02` results in identical binary trees, because both mappings yield false in the generator (Line 3). Figure 6.16 shows the frequency of zero mutations in inputs generated by Zest and EI. We see that EI generates a much smaller proportion of zero mutations than Zest. This should lead to better fuzzer efficiency. In future work, we will also study the occurrence of zero mutations in BeDivFuzz and Zeugma.

Preliminary Coverage Comparison

To assess the impact of destructive mutations on code coverage, we selected four fuzzing techniques: Zest [19], EI, BeDivFuzz [33], and Zeugma [35]. For BeDivFuzz, we consider two

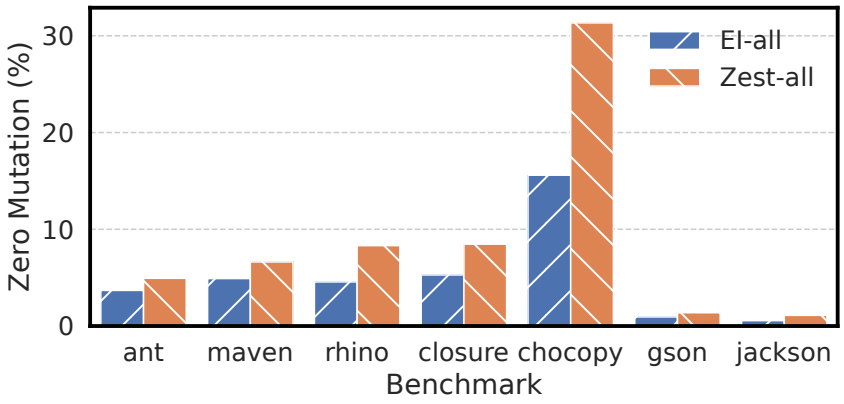


Figure 6.16: Percentage of zero mutations (i.e., mutants identical to parent) out of all generated inputs for EI and Zest.

configurations used by the prior research [33, 35]: BDF(S), which only includes structural mutation, and BDF(D), which includes both structural mutation and feedback of input structure novelty. Each technique was used to fuzz Closure for 24 hours, repeated 20 times. Since Zeugma is implemented in a different fuzzing framework, we also recorded the number of total executions to understand potential alternative causes for variations in branch coverage, such as differences in fuzzing speed.

Table 6.13 shows the average branch coverage for each technique. Although EI achieves higher average coverage than Zest, the Mann-Whitney U test result shows no significant difference. BeDivFuzz shows lower executions compared to Zest and EI, with significantly lower branch coverage. Zeugma significantly outperforms Zest and EI in branch coverage, but the number of executions achieved by Zeugma is also significantly higher (almost 2×)—we posit that at least some of the speed is due to engineering improvements, since Zeugma is the only tool from the ones we studied that is not built on top of JQF but instead implemented afresh from the ground up. Thus, we are not certain whether Zeugma's higher branch coverage in a fixed time budget is attainable solely to the linked-crossover technique or the difference in execution speed. We plan to conduct further experiments to isolate and control for this difference.

The goal of our evaluation is to determine how the havoc effect influences the performance of generator-based fuzzers, identify the conditions under which it negatively impacts code coverage, and assess whether existing structure-preserving techniques effectively mitigate this effect. Our proposed evaluation aims to answer the following questions.

RQ1: How destructive, in terms of edit distance, are the mutations performed by our studied techniques?

RQ2: How much do the mutations performed by our studied techniques preserve validity?

RQ3: How does the coverage achieved by our studied techniques compare under the 48-hour configuration?

RQ4: Do the more complex mutations of our studied techniques incur runtime overhead?

In order to answer these questions, we will follow the following experimental procedures.

Expanded evaluation with more targets and benchmarks. We plan to expand our

evaluation to analyze the impact of the havoc effect on the performance of generator-based fuzzers. To do this, we will introduce two baselines: Random [5, 180] and Zest-mini. Random performs property-based testing without feedback guidance, while Zest-mini limits the mean mutation size to four bytes (equivalent to the size of an integer in Java). For all targets listed in Table 6.12, we will measure the mutation distance for Zeugma, BeDivFuzz, and Zest-mini. We will also extend our coverage measurement (ref. Section 6.4) to all benchmarks. Our goal is to study the relationship between the magnitude of the havoc effect and the branch coverage across all benchmarks using all of the studied fuzzing tools.

Improved mutation distance measurement. Extending the mutation distance measurement to BeDivFuzz, Zest-mini, and Zeugma presents a challenge, as these tools tend to mutate a different average number of bytes compared to Zest and EI. To eliminate this source of bias, we introduce two distinct metrics: *parametric mutation distance*, which quantifies the mutation distance within the bytestream, and *input mutation distance*, which measures the mutation distance based on the generated input itself.

Validity-preserving mutation measurement. We would like to understand the impact of structure-preserving mutations on validity. A key advantage of parametric generator-based fuzzing is its capability to generate a higher percentage of valid inputs. In the proposed evaluation, we would like to understand if the structure-preserving mutations enhance this capability. Following prior work [19], we will use the validity signals (e.g., type-checking result in Chocopy) from each of the targets (Table 6.12) to determine if a given input is deemed valid or not.

Fair coverage evaluation between frameworks. One issue in our preliminary results is that different tools are built on top of different fuzzing frameworks. For example, Zeugma uses a lighter instrumentation strategy, achieving higher fuzzing speed compared to Zest, EI, and BeDivFuzz (which are implemented on top of JQF, thus paying a performance penalty for the extensibility of the underlying framework). To ensure a fair comparison of branch coverage without reimplementing all techniques in a single framework, we introduce two additional coverage measurements: *count-based coverage measurement* and *normalized coverage measurement*.

In the count-based coverage measurement, we fix the number of inputs generated by each technique and evaluate the corresponding code coverage achieved by each approach.

For the normalized coverage measurement, we will compare the execution speeds of Zest and Zeugma-without-linked-crossover (Zeugma-X in the original paper [35]) to isolate the performance improvements introduced by the Zeugma framework without introducing a new mutation technique. The difference between these two configurations will give us an adjustment factor to compensate for the difference in the underlying instrumentation framework, which we will then apply to artificially slow down Zeugma (by reducing its total running time) when comparing with the JQF-based techniques for coverage measurement. Note that while this slowing down Zeugma appears "unfair", recall that our goal is not to identify the best fuzzing tool for a practitioner to use, but instead to study the nuances of the havoc effect on coverage—the handicap should enable this study without bias.

Appendix E: Cedar Generators Evaluation

We structure our evaluation around two research questions:
RQ1: *Which generators result in the highest input validity and diversity?*
RQ2: *What is the impact of coverage guidance on input diversity?*

Experimental Setup

Targets

1. Cedar Policy Validator: this component validates a set of policies against a schema that defines possible types and their hierarchies. The inputs to the validator are a set of policies and a schema.
2. Cedar Authorizer: this component executes an authorization request given a set of policies and user data. The inputs to the authorizer are a set of policies, user data (also known as entities), and a request.
3. Cedar Evaluator: this component interprets a Cedar expression given user data and a request. The inputs to the evaluator are an expression, entities, and a request.

Guidances We used the `cargo-bolero` crate to run pure random generation and coverage guidance with libFuzzer.

Configurations Each fuzzing campaign was run for 12 hours and repeated 5 times to account for randomness.

Results

Input Validity and Diversity To evaluate the three types of generators, we measured input validity and diversity using purely random generation. Table 6.14 shows the efficiency and validity on all of the targets. First, we observe that the Derived generators consistently shows high throughput but poor validity across all benchmarks. This is due to the Cedar syntactic and semantic constraints that are not encoded into the derived generator logic.

We also observe that the fail-fix generator achieves the highest percentage of valid inputs on all targets. This is by design, as the generator invocation will default to a valid constant if encountering a sequence of random choices that would result in an invalid scenario. Due to this, there is a large difference in execution speed between fail-fast and fail-fix generators; fail-fast generators will exist generator invocations early in order to generate a new input. However, the number of *valid* inputs per second is significantly higher for the fail-fix generator due to its high validity rate overall.

Figure 6.17 visualizes the uniqueness of various features of the inputs produced by each of our generators. Our results indicate that derived generators suffer from a high rate of duplicates in their output. For the Cedar authorizer target, about 87% of the inputs produced by derived generators were duplicates. In contrast, the hand-written fail-fast and

Target	Generator	Total Exec/s	Valid Exec/s	Valid %
ABAC	Derived	289.56	0.79	0.27%
	Fail-fast	672.43	174.00	25.88%
	Fail-fix	282.45	**281.66**	**99.72%**
Eval	Derived	301.80	7.54	2.50%
	Fail-fast	1033.39	329.01	31.84%
	Fail-fix	742.40	**741.05**	**99.82%**
Validation DRT	Derived	405.05	7.96	1.97%
	Fail-fast	889.17	**118.95**	13.38%
	Fail-fix	50.06	45.60	**91.08%**

Table 6.14: Performance comparison of different generator types across various targets. Measurements are averaged over five repetitions.

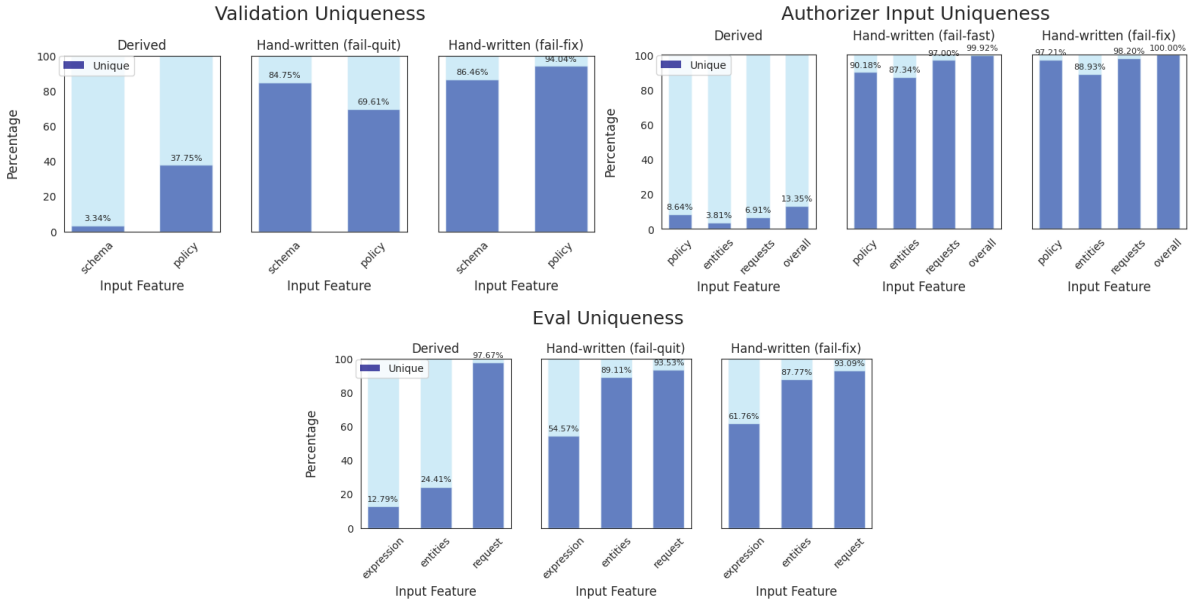


Figure 6.17: Bar plots showing uniqueness of each input for all three targets.

fail-fix generators demonstrated substantially higher input diversity, with almost 99% of generated inputs being unique.

Impact of Coverage Guidance Figure 6.18 demonstrates a surprising result: purely random generation produces a significantly higher percentage of unique inputs compared to coverage-guided generation. This finding challenges the conventional wisdom that coverage guidance can lead to more diverse test cases. We believe this occurs because the mutations on the parametric input are only changing part of the input array that controls the randomness in the generator; purely random generation, on the other hand, samples an entirely new input array.

To better understand the high rate of duplicates during coverage-guided fuzzing, we analyzed a sample of the generated inputs to the Cedar authorizer. We observed three distinct patterns in how mutations affect parametric inputs. First, some mutations of parametric

inputs simply produced duplicate parametric inputs, accounting for approximately 5% of cases. Second, in 20% of cases, mutations on parametric inputs resulted in changes to the randomly generated schema without affecting the actual authorizer inputs (policies, entities, requests). Most significantly, we found that in 80% of cases, mutations on parametric inputs neither changed the schema nor the authorizer inputs, effectively traversing the same generator path and producing equivalent inputs. This reveals a surprising insight: a large percentage of parametric inputs can map to the exact same concretized input, and most likely depends on how the generator is written. In the case of fail-fast generators, many mutations on the parametric input could cause the generator to exit early due to an invalid path. In the case of fail-fix generators, mutations on the parametric input could result in the same constant input in the case that an invalid path is taken.

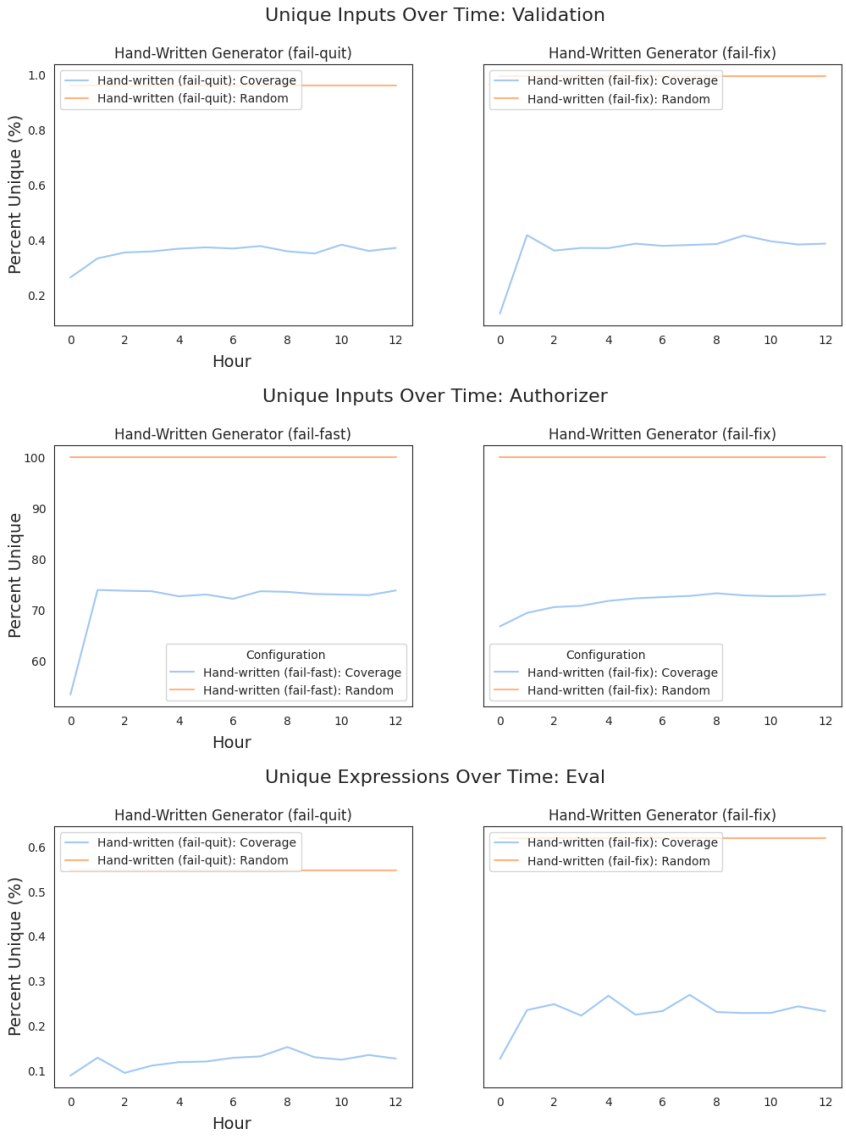


Figure 6.18: Line plots showing percentage of unique inputs over time when using coverage guidance and hand-written generators. The orange line is a baseline average when using purely random generation. We can observe that coverage guidance results in a significantly lower percentage of unique inputs.

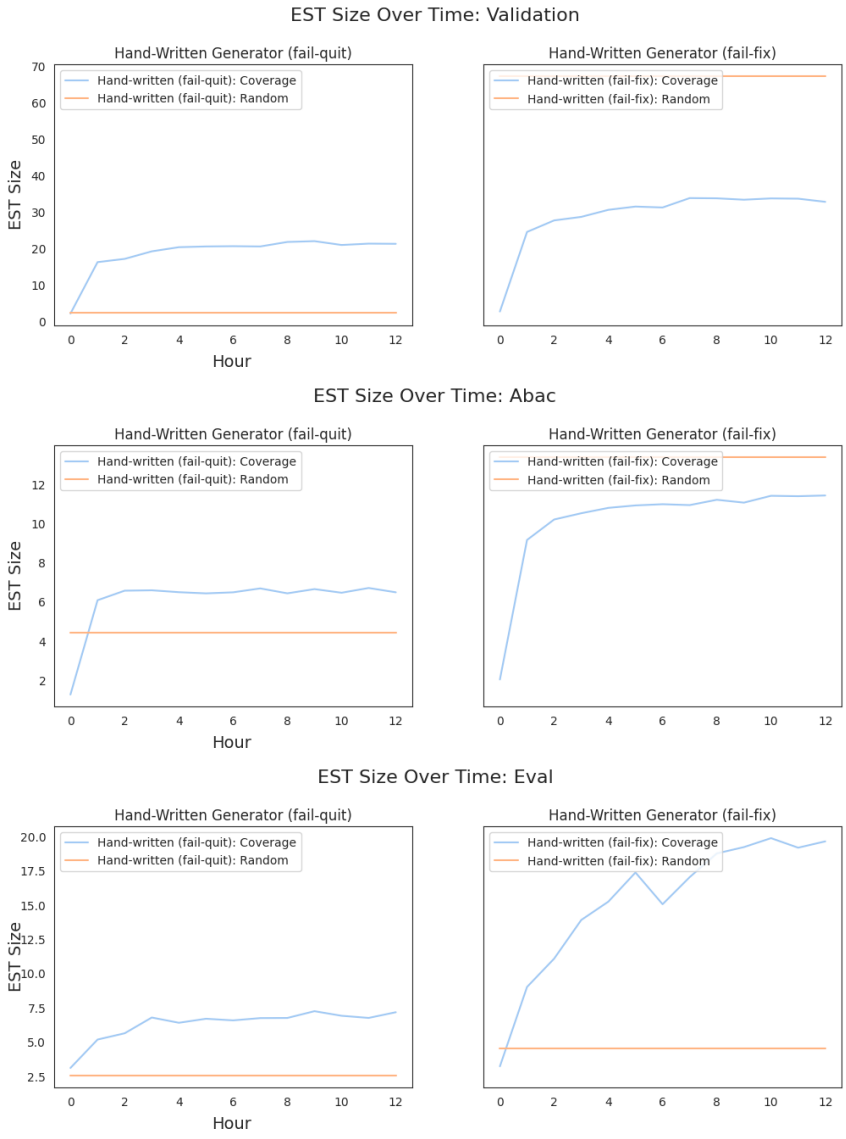


Figure 6.19: Line plots showing syntax tree size over time when using coverage guidance and hand-written generators. The orange line is a baseline average when using purely random generation. We observe that the size when using coverage guidance gradually increases over time.