Guiding Greybox Fuzzing with Mutation Testing

Vasudev Vikram
Carnegie Mellon University
Pittsburgh, PA, USA
vasumv@cmu.edu

Isabella Laybourn
Carnegie Mellon University
Pittsburgh, PA, USA
ilaybour@andrew.cmu.edu

Ao Li
Carnegie Mellon University
Pittsburgh, PA, USA
aoli@cmu.edu

Nicole Nair
Swarthmore College
Swarthmore, PA, USA
nnair1@swarthmore.edu

Kelton OBrien
University of Minnesota
Minneapolis, MN, USA
obri0707@umn.edu

Rafaello Sanna
University of Rochester
Rochester, NY, USA
rsanna@u.rochester.edu

Rohan Padhye
Carnegie Mellon University
Pittsburgh, PA, USA
rohanpadhye@cmu.edu

ABSTRACT

Greybox fuzzing and mutation testing are two popular but mostly independent fields of software testing research that have so far had limited overlap. Greybox fuzzing, generally geared towards searching for new bugs, predominantly uses code coverage for selecting inputs to save. Mutation testing is primarily used as a stronger alternative to code coverage in assessing the quality of regression tests; the idea is to evaluate tests for their ability to identify artificially injected faults in the target program. But what if we wanted to use greybox fuzzing to synthesize high-quality regression tests?

In this paper, we develop and evaluate Mu2, a Java-based framework for incorporating mutation analysis in the greybox fuzzing loop, with the goal of producing a test-input corpus with a high mutation score. Mu2 makes use of a differential oracle for identifying inputs that exercise interesting program behavior without causing crashes. This paper describes several dynamic optimizations implemented in Mu2 to overcome the high cost of performing mutation analysis with every fuzzer-generated input. These optimizations introduce trade-offs in fuzzing throughput and mutation killing ability, which we evaluate empirically on five real-world Java benchmarks. Overall, variants of Mu2 are able to synthesize test-input corpora with a higher mutation score than state-of-the-art Java fuzzer Zest.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

fuzz testing, mutation testing, test generation

ISSTA '23, July 17–21, 2023, Seattle, WA, United States

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598107

ACM Reference Format:
Vasudev Vikram, Isabella Laybourn, Ao Li, Nicole Nair, Kelton OBrien, Rafaello Sanna, and Rohan Padhye. 2023. Guiding Greybox Fuzzing with Mutation Testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598107

1 INTRODUCTION

Greybox fuzzing [9, 49, 52, 80] and coverage-guided property testing [44, 58] have become increasingly popular for automated testing. Their key idea is to evolve a corpus of test inputs via an evolutionary search that maximizes code coverage: in each iteration, a new input is synthesized by randomly mutating an existing input from the corpus. The mutated input is added to the corpus if the corresponding execution of the test program increases code coverage.

Fuzzing is traditionally used to discover inputs that crash programs and reveal security vulnerabilities [5, 11, 14, 20, 25, 42, 50, 54, 59, 68]. In the absence of new bugs, fuzzers are evaluated based on code coverage achieved during the fuzzing campaign [10, 48]. However, in the vast majority of fuzzing research, the end goal is to find bugs in the moment [42]; not much attention is paid to the inputs saved along the way.

In this paper, 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 [28], and is already adopted by some mature projects. 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`*"* [71]. Similarly, OpenSSL uses several distinct fuzzer-generated corpora and their corresponding fuzz drivers for continuous testing [72]. 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 strongest predictor of fault detection ability [15, 36].

Now, the technique of *mutation testing* [19], which evaluates the ability of tests to catch artificially injected bugs (a.k.a. *mutation*

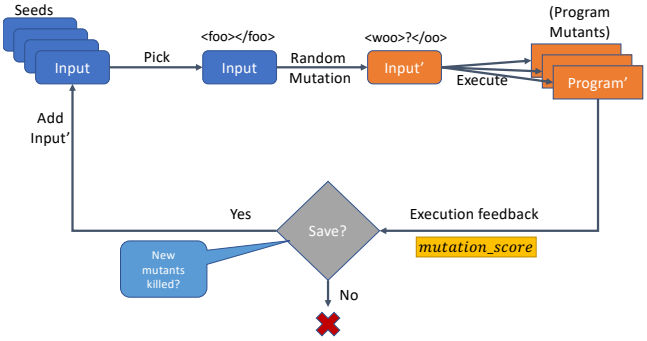

Figure 1: A mutation-analysis-guided fuzzing loop. Each fuzzer-generated input is run through a set of program mutants to compute a mutation score. Inputs are saved to the corpus if they improve mutation score.

analysis), has shown promise as an adequacy criteria for improving test-suite effectiveness [15, 39, 64]. 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*?

In this paper, we develop and evaluate a framework for incorporating mutation analysis in the fuzzing loop, building on our previous work which first proposed the approach [46]. The idea is as follows (see Fig. 1): 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 [17] into a custom guidance in the JQF [58] greybox fuzzing framework. Mu2 is open source and available at: https://github.com/cmu-pasta/mu2.

This paper 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.

We evaluate Mu2 on five real-world Java targets using state-of-the-art greybox fuzzer Zest [59], 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.

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;

To summarize, this paper makes the following contributions:

(1) We investigate the various challenges of combining mutation testing and greybox fuzzing, and propose solution approaches to include in our framework.
(2) We incorporate *differential testing* as an oracle for mutation testing in the fuzzing loop and find that it significantly improves the strength of the fuzzing oracle.
(3) We employ multiple *sound* performance optimizations that enable mutation analysis to run in the fuzzing loop, and propose *aggressive* optimizations that are able to scale Mu2 to larger programs.
(4) We present an empirical evaluation of Mu2 on 5 real-world Java benchmarks, with Zest [59] as a baseline.
(5) We open-source Mu2 for use by practitioners and to enable reproduction and extension by researchers.

2 BACKGROUND

2.1 Greybox Fuzzing and Corpus Generation

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 [80] and libFuzzer [49], but has since been heavily studied and variously extended in academic research [5, 9, 14, 20, 25, 44, 50, 52, 58, 59].

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), 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 [9]. 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). Structure-aware fuzzing tools [4, 44, 59, 65, 75] perform mutations that preserve the syntax or type safety of inputs, e.g. by mutating parse trees using a grammar or by mutating pseudorandom choices backing a Quickcheck-like [16] generator function. 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.

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
```

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 [28, 55, 71, 72, 74]. The quality of the final test-input corpus is often evaluated using code coverage [10, 42], though mutation scores—which we describe in the next section—have also been used [74].

2.2 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* [19, 37]. In assessing test adequacy [27], 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 [78] 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 $P' \in \textsc{Mutants}(P)$ 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 paper, 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 $\textsc{Kills}(P', x)$. 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: $\frac{|\{P' \in \textsc{Mutants}(P) | \exists x \in X : \textsc{kills}(P', x)\}|}{|\textsc{Mutants}(P)|}$.

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 [17], 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 MUTATION-ANALYSIS-GUIDED FUZZING

3.1 Problem Statement and Scope

In this paper, 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?*

Recently, Gopinath et al. [31] 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 paper. Oracles are discussed in Section 3.3 and performance concerns in Section 3.4. Our evaluation is not dependent on identifying 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 3.4.2 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. [63], we restrict ourselves in this paper 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 [2, 12, 15, 32, 39, 41, 64], and (2) directly use the default set of mutation operators provided by PIT (ref. Section 2.2), which have been chosen based on several empirical studies of effectiveness, sufficiency, and to align with developer expectations [1, 17, 45, 57].

3.2 The Mu2 Framework

To address our problem statement, we present the mutation-analysis-guided greybox fuzzing technique in Algorithm 2. 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 2.2, though we will refine this in Section 3.4.2.

Algorithm 2 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)}
```

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

We have implemented Algorithm 2 for fuzzing Java programs by integrating PIT [17] into JQF [58]. 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 2.2, PIT is an actively developed mutation testing framework that operates on JVM bytecode. The JQF framework [58] was originally designed for *coverage-guided property-based testing*, which is a structure-aware variant of greybox fuzzing (ref. Section 2.1) 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 [43, 55, 56, 59, 69, 74, 82].

In Mu2, Mutants(P) includes all of PIT's default expression mutation operators (ref. Sections 2.2 and 3.1). For heuristics such as PickInput and MutateInput, Mu2 reuses the logic and code from Zest [59], which we also use as a baseline for evaluation (Section 4).

3.3 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 2.

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

```
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 2: Java program that implements insertion sort, annotated with four sample program mutants.

defined in Figure 2 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
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 2.2: executing mutant P'_1 = ⟨`Sort`, `i+1`, `i`, 10⟩ 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 = ⟨`Sort`, `i>=0`, `i>0`, 5⟩ 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 = ⟨`Sort`, `arr[i]`, 1, 7⟩, 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 [6]. 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 [21, 53], 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. The `@Diff` method `runInsertionSort`

```
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: A Mu2 differential mutation test driver and comparison method for the `insertionSort` method (Fig. 2).

Table 1: Geometric mean of speedups achieved by the execution and infection based optimizations (Alg. 3, Line 4) from the PIE model [38] 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×

returns an output value of type `int[]`. The user-defined comparison method `checkEq` simply 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 [79].

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 2. 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) 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]`.

We evaluate the improvement in completeness using the differential oracle over the greybox fuzzing implicit oracle in Section 4.4.

3.4 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 [63], 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.

Algorithm 3 Logic for determining which mutants to run in a given iteration of the fuzzing loop (Alg. 2)

```
1: function ProgMuts2Run(Program P, Old inputs corpus, New input x)
2:     surviving ← mutants(P) \ killed(P, corpus)
3:     killable ← {P' = ⟨P, e, e', n⟩ | (P' ∈ surviving) ∧
4:                 (n ∈ coverage(P, x)) ∧ (infect(P, e, e', x))}
5:     if AGGRESSIVE_OPT is configured then
6:         return filter(killable, AGGRESSIVE_OPT)
7:     return killable
```

3.4.1 Improving performance of mutant execution. When running a mutation testing tool such as PIT [17], 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 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 [30]. In the steady state (after the first 8 inputs), Mu2's in-memory analysis runs with a 9.6× speed-up over PIT.

3.4.2 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:

$$trialTime = \text{time}_{\text{orig}} + M * \text{avgTime}_{\text{mut}} \quad (1)$$

where $M = |$ProgMuts2Run$(P, corpus, x)|$ as per Algorithm 2.

[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.

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 2 (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 3, lines 2–4:

(1) If $P' \in \textsc{killed}(P, \mathit{corpus})$, 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 [38].
(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 [38], 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 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 1 shows the results of preliminary experiments on 4 benchmarks included in our evaluations in Section 4 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* [47].

3.4.3 Aggressive mutant selection optimizations. While the execution and infection optimizations significantly improve the overall throughput of Mu2, the M factor in Equation 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 3 (Line 6) to optionally apply a selection strategy [66, 73] 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. 3.
(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 campaign. This is a novel reduction strategy designed specifically for the fuzzing loop.

Section 4.2 evaluates the impact of these aggressive optimizations.

4 EVALUATION

We evaluate Mu2 on 5 different Java program benchmarks, using state-of-the-art coverage-guided fuzzer Zest [59] as the baseline. We structure our evaluation around four research questions:
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 [7, 61] reference compiler (~6K LoC): The test driver (reused from [74]) 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 [29] JSON Parser (~26K LoC): The test driver parses a input JSON string and returns a Java object output.
(3) Jackson [22] JSON Parser (~49K LoC): The test driver acts similar to that of Gson.
(4) Apache Tomcat [3] WebXML Parser (~10K LoC): The test driver parses a string input and returns the WebXML representation of the parsed output.
(5) Google Closure Compiler [30] (~250K LoC): The test driver (reused from [59] and [74]) takes in a JavaScript program and performs source-to-source optimizations. It then returns the optimized JavaScript code.

Mutation selection. Following previous work on semantic fuzzing [59, 74], 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 [42], 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 2.2), 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

[2]While we note lines of code (LoC) for completeness, only a fraction of this code is reachable from fuzz drivers. Fig. 5 indicates actual code coverage.

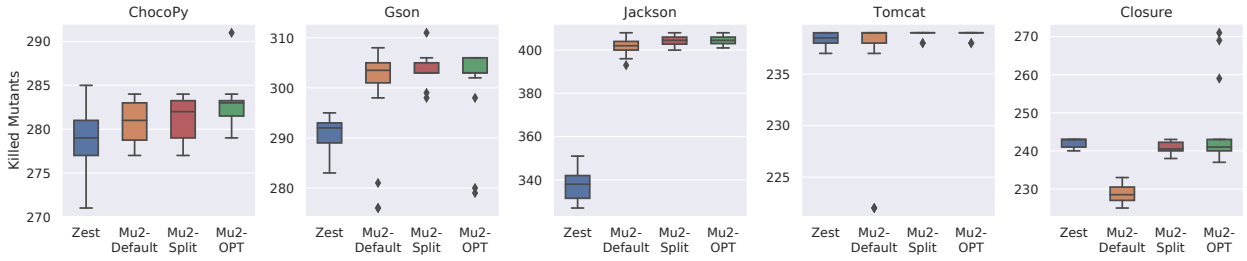

Figure 4: Box plots showing the number of killed mutants by Zest and Mu2-generated test corpora across 20 repetitions of 24-hour fuzzing campaigns (higher is better). Mu2-Split and Mu2-OPT are two variants of Mu2 detailed in Section 4.1.

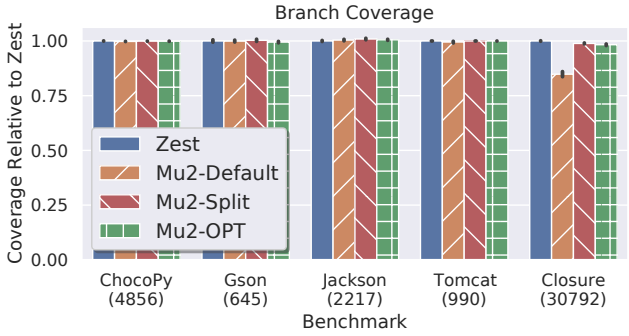

Figure 5: Branch coverage across all benchmarks normalized to the mean coverage achieved by Zest. The number of branches covered by Zest (used to normalize) is listed below each target. Error bars represent 95% confidence intervals.

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 3.4.2.

Reproducibility and Data Availability. We have included a replication package and evaluation data in the repository at: https://zenodo.org/record/7978404. The evaluation data contains logs of fuzzing campaigns used to generate all evaluation figures and tables.

4.1 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 4 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 5 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 5). 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. [31], 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. 4), 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. 5) and mutation scores (Fig. 4) that Mu2-Default had with the Zest baseline.

Another method of scaling Mu2 is to apply the aggressive optimizations detailed in Section 3.4.3. 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. 4). 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 2 displays the average sizes and runtimes for each fuzzer-generated corpus and show that no such bloat occurs in Mu2. While there

Table 2: 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 s ± 2.1 s)	711 ± 47 (9.8 s ± 0.8 s)	725 ± 36 (11.7 s ± 1.3 s)	746 ± 40 (10.4 s ± 1.0 s)
Gson	467 ± 18 (1.7 s ± 0.1 s)	461 ± 17 (1.7 s ± 0.1 s)	469 ± 15 (1.7 s ± 0.0 s)	489 ± 21 (1.7 s ± 0.0 s)
Jackson	598 ± 19 (2.4 s ± 0.1 s)	655 ± 16 (2.4 s ± 0.1 s)	641 ± 19 (2.4 s ± 0.0 s)	673 ± 15 (2.4 s ± 0.1 s)
Tomcat	138 ± 7 (2.6 s ± 0.1 s)	122 ± 6 (2.5 s ± 0.1 s)	136 ± 6 (2.6 s ± 0.1 s)	171 ± 5 (2.7 s ± 0.1 s)
Closure	4885 ± 205 (554 s ± 82 s)	1075 ± 219 (58 s ± 13 s)	4044 ± 146 (360 s ± 27 s)	4037 ± 192 (353 s ± 30 s)

Table 3: 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×

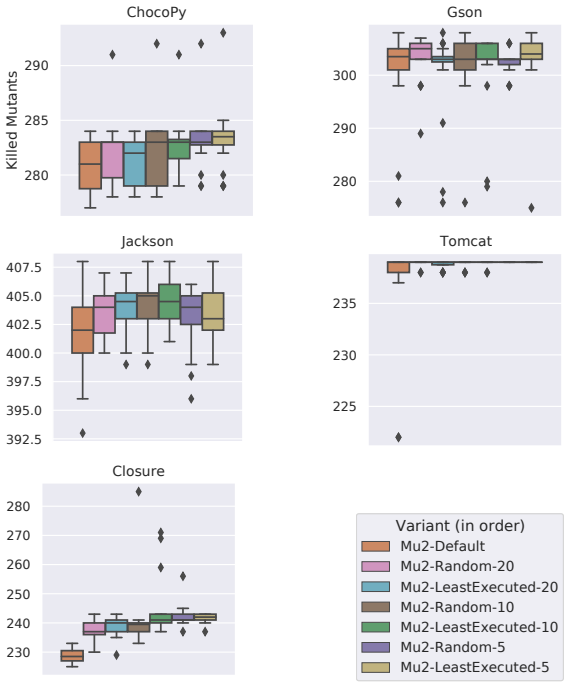

Figure 6: Box plots showing number of killed mutants by each aggressively optimized variant of Mu2 and the default, across 20 repetitions of 24 hour campaigns (higher is better).

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.

4.2 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 mitigating the scalability concerns of Mu2-Default. We created variants Mu2-LeastExecuted-k and Mu2-Random-k, each applying the corresponding filter described in Section 3.4.3, and chose three different values of $k \in \{5, 10, 20\}$.

First, we measure just the performance benefit. Table 3 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 3.4.2 and Table 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 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. 3) due to the smaller number of mutants, the mutation score slightly decreases (Fig. 6) 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.

4.3 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 [40]). 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 7 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 7 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.

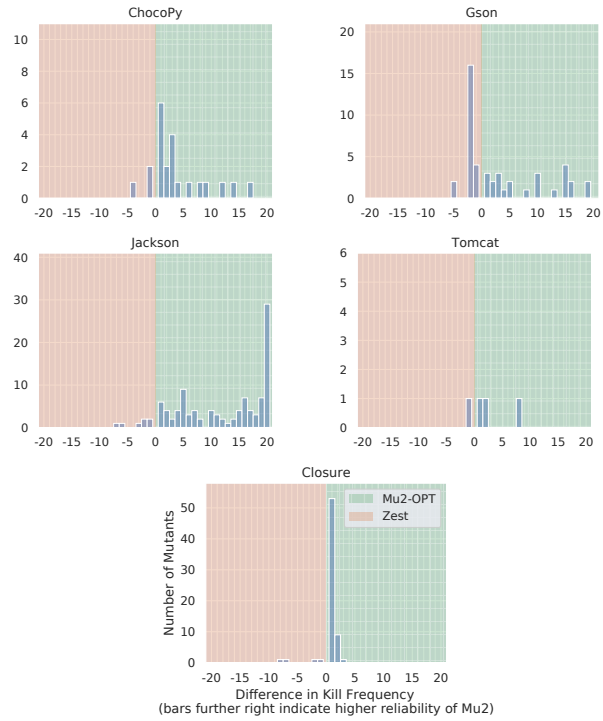

Figure 7: Histogram of difference in kill-rate of *nontrivial* mutants between Zest and Mu2-OPT over 20 experiments. X-axis is the difference in repetitions (ranging from -20 to 20), and Y-axis is the number of mutants. Larger positive differences (right) are better for Mu2-OPT, and larger negative differences (left) are better for Zest.

4.4 RQ4: Differential Mutation Testing

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

Described in Section 3.3, 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 8 shows the difference in mutant kills across the benchmarks with the two types 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

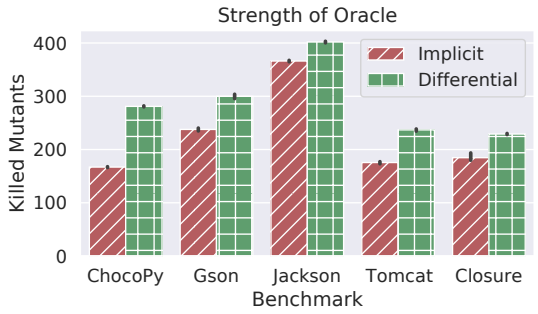

Figure 8: Number of killed mutants detected by the differential oracle vs. the implicit oracle across 20 repetitions of 24 hour campaigns with Mu2-Default (higher is better). Error bars represent 95% confidence intervals.

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.

5 THREATS TO VALIDITY

Threats to construct validity. First, the measurement of mutation score is of course dependent on the set of mutation operators being applied to generate program mutants [62]. We aim to mitigate this threat by using the default set of operators in the widely used PIT framework, as justified in Section 3.1. Second, our test oracles (ref. Section 3.3) report an outcome of `TIMEOUT` if a mutant execution does not terminate within a predefined limit. Such a bound is necessary to catch infinite loops (e.g., for mutants that negate loop conditions). However, if this bound is too small, then it is possible in theory that some mutants could be marked as "killed" by a fuzzer-generated input even if their execution would eventually produce a correct output. To mitigate this threat, we compute the mutation scores for the final test-input corpus by re-running saved inputs on all program mutants using a larger timeout. We also manually analyzed a sample of reported timeouts to confirm correspondence to infinite loops—we found no false kills.

Threats to internal validity. Our evaluation uses mutation score when comparing the quality of the generated test-input corpora since our goal was to synthesize a test-input corpus with high mutation score (ref. Section 3.1). We assume that a high mutation score is a valuable objective for fuzzers. However, there is a potential bias from using mutation score as an evaluation metric, as Mu2 benefits from incorporating mutation testing in the fuzzing loop. Our results nevertheless capture the performance overhead impact of mutation-analysis-guided fuzzing on mutation score and code coverage.

Our implementation simply reused all the fuzzing hyperparameters (e.g., PickInput and MutateInput in Algorithms 1 and 2) that were set by the baseline Zest fuzzer. Tuning these heuristics could affect our results, but the size of this search space is too large for us to explore systematically. We stick with the baseline-provided defaults for simplicity and make sure to use the same hyperparameters for both Zest and Mu2 so that our conclusions are exclusively based on the inclusion of mutation-analysis guidance in Mu2 only.

Threats to external validity. Since our implementation is based on JQF [58] and PIT [17], which both target JVM bytecode, we used Zest as the baseline. We do not know if our conclusions will generalize to other programming languages or fuzzing platforms, such as the family of tools based on AFL [80] and libFuzzer [49]. The available mutation testing infrastructure for C/C++ appears to be less mature than that for Java/JVM. Another threat to external validity arises from our selection bias in choice of benchmark programs. Our targets have input and output formats which make them amenable to differential mutation testing. This is not always true for all applications that can be fuzzed—e.g., PDF viewers and other programs whose output is graphical. The study of the general test oracle problem [6] is outside the scope of this paper.

6 RELATED WORK

Greybox fuzzing. The field of coverage-guided greybox fuzzing has a vast literature, as surveyed by Manès et al. [52]; a more recent and evolving publication list is maintained by Wen [77]. The majority of fuzzing research focuses on improving heuristics such as seed-picking power schedules [9], input mutations [5, 48, 50], and coverage feedback [14, 25]. FuzzFactory [60] generalizes the feedback of greybox fuzzing beyond code coverage to domain-specific metrics that satisfy certain conditions. Our proposed mutation-analysis guidance fits into this framework.

Greybox fuzzing for regression testing. A family of techniques have been developed for directing fuzz testing towards specific code locations [8, 13, 76] or code commits [83], which can be used for identifying regressions. However, this still requires running a full fuzzing campaign, which can take hours or days. In contrast, we focus on synthesizing a high-quality test-input corpus which can be quickly executed in CI—usually taking a few seconds or minutes—as is often already practiced (ref. Section 1).

Guiding fuzzing with mutation testing. We first proposed the idea of using mutation testing to augment greybox fuzzing in a student research competition [46]; independently, Qian et al. [67] published a similar idea at a regional symposium. However, we believe the current paper is the first to thoroughly evaluate the performance and scalability of incorporating mutation testing in the fuzzing loop. In particular, we identified that the evaluation in Qian et al.'s paper [67] uses an *unsound* comparison to the baseline Zest; they use mutation analysis with multiple threads but run Zest only single threaded for the same time bound, hence giving higher CPU time to their technique and obscuring the effects of the increased overhead of performing mutation testing. Additionally, they use a selection strategy to choose 10 mutants at random, but

do not measure the impact on the overall mutation score, since they never run all killable mutants. We were unable to perform a head-to-head evaluation between Mu2 and their technique since their implementation is not open source.

Using mutation testing in automated test generation. In a registered report, Groce et al. [33] propose fuzzing specially mutated targets to find inputs triggering interesting control flow not in the original program, and then use those inputs as seeds for coverage-guided fuzzing. However, they do not target maximizing mutant kills—for example, a mutant which only changes return codes gets low fuzzing priority in their approach because it won't affect control flow [33]. In contrast, Mu2 aims to find inputs that differentiate program *output* on potentially semantics-altering mutants, which often change data values but not necessarily control flow. Our approach is therefore orthogonal to Groce et al.'s and could potentially even be combined.

μ-test [24] and EvoSuite [23] are evolutionary test-generation techniques that can use mutation scores as an objective as well as a fitness function. μ-test, which is based on Javalanche [70], uses a form of differential testing to compare the coverage traces of the original program and a mutant. Unlike these tools, which generate *unit test methods* for exercising program API, greybox fuzzing focuses on the generation of *inputs for system testing*, given a fixed entry point.

Improving the performance of mutation testing. A lot of research has been conducted to speed up mutation testing [18, 37, 63, 66, 73]. The approaches fall into three categories: (1) reducing the number of mutants to generate, (2) pruning mutants to run on a given test, and (3) speeding up mutant evaluation on a given test. For example, many techniques have been developed to avoid generating *redundant* or *equivalent* mutants [51]; we do not currently make an attempt to identify these statically. Just et al. [39] introduce the *propagation*, *infection*, *execution* (PIE) model to prune mutants that are test-equivalent using dynamic analysis. Mu2 implements the *execution* and *infection* optimizations from this work. MeMu [26] speeds up PIT's mutation analysis by memoizing unmutated methods with long execution time; this is a promising approach that could be integrated into Mu2. Kaufman et al. [40] prioritize mutants to reach test completeness faster. All these optimizations are *sound*—they do not avoid analyzing mutants that may be killable.

Other research directions aim to reduce mutation-analysis costs while potentially trading off soundness. For example, weak mutation [35] has been proposed to terminate mutant evaluation quickly by observing the intermediate state after executing the mutated program locations. Many techniques have been developed for *mutation reduction* [63, 66, 73]—where only a subset of mutants are evaluated based on some program-specific criteria. In this paper, we have evaluated the *random sampling* approach and a novel *least-executed* approach to mutant selection. Recently, Guizzo et al. [34] have proposed an evolutionary approach to automate the generation of optimal cost reduction strategies. Further, predictive mutation testing [81] uses machine learning to estimate which mutants are most likely to be killed. Incorporating such advanced models into the Mu2 framework are promising directions for future work.

7 CONCLUSION

We investigated the challenges of incorporating mutation analysis to guide greybox fuzzing. Our implementation, Mu2, integrates PIT mutation testing into the JQF framework, and is aimed at producing a test-input corpus with high mutation score. In our design, we incorporated a differential testing as an oracle for killing mutants and proposed optimizations to improve fuzzing throughput by dynamically pruning the number of mutants to be executed. We applied both *sound* and *aggressive* optimizations for Mu2 to help scale it to larger programs. After conducting a thorough evaluation on Mu2 and several variants, we found that mutation-analysis feedback can improve the mutation score of a test-input corpus and more reliably kill nontrivial mutants than coverage-guided fuzzing.

One of the challenges identified by Gopinath et al. [31] was to "improve visibility of mutation analysis among fuzzing researchers." We hope our work increases awareness of mutation analysis techniques in the fuzzing community and encourages other researchers to develop more advanced hybrid techniques.

ACKNOWLEDGMENTS

This research was funded in part by NSF grant CCF-2120955, a seed grant from CMU's CyLab, and an Amazon Research Award.

REFERENCES

[1] Paul Ammann. 2015. 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. https://mutation-workshop.github.io/2015/program/MutationKeynote.pdf

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is Mutation an Appropriate Tool for Testing Experiments?. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, 402–411. https://doi.org/10.1145/1062455.1062530

[3] Apache Foundation. 2022. Tomcat. https://github.com/apache/tomcat. Retrieved August 31, 2022.

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

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

[6] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[7] U. C. Berkeley. 2019. ChocoPy. https://chocopy.org/. Reference compiler JAR retrieved on January 12, 2022.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1032–1043.

[10] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *44th IEEE/ACM International Conference on Software Engineering (ICSE'22)*. to appear.

[11] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 725–741.

[12] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. 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, 597–608.

[13] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108.

[14] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[15] Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. 2020. 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* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, 237–249. https://doi.org/10.1145/3324884.3416667

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

[17] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*. 449–452.

[18] Fabiano Cutigi Ferrari, Alessandro Viola Pizzoleto, and Jeff Offutt. 2018. A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–10. https://doi.org/10.1109/ICSTW.2018.00021

[19] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[20] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 131–142.

[21] Robert B Evans and Alberto Savoia. 2007. 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*. 549–552.

[22] FasterXML. [n. d.]. Jackson: JSON for Java. https://github.com/FasterXML/jackson. Retrieved August 31, 2022.

[23] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*.

[24] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven Generation of Unit Tests and Oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. ACM, 147–158. https://doi.org/10.1145/1831708.1831728

[25] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2577–2594.

[26] Ali Ghanbari and Andrian Marcus. 2022. Faster Mutation Analysis with MeMu. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. 781–784. https://doi.org/10.1145/3533767.3543288

[27] John B Goodenough and Susan L Gerhart. 1975. Toward a theory of test data selection. *IEEE Transactions on software Engineering* 2 (1975), 156–173.

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

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

[30] Google. 2022. Google Closure Compiler. https://github.com/google/closure-compiler. Retrieved August 31, 2022.

[31] Rahul Gopinath, Philipp Görz, and Alex Groce. 2022. Mutation Analysis: Answering the Fuzzing Challenge. *CoRR* abs/2201.11303 (2022). arXiv:2201.11303 https://arxiv.org/abs/2201.11303

[32] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Mutations: How Close are they to Real Faults?. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*. 189–200. https://doi.org/10.1109/ISSRE.2014.40

[33] Alex Groce, Goutamkumar Tulajappa Kalburgi, Claire Le Goues, Kush Jain, and Rahul Gopinath. 2022. Registered Report: First, Fuzz the Mutants. In *International Fuzzing Workshop (FUZZING'22)*.

[34] Giovani Guizzo, Federica Sarro, Jens Krinke, and Silvia R. Vergilio. 2022. Sentinel: A Hyper-Heuristic for the Generation of Mutant Reduction Strategies. *IEEE Transactions on Software Engineering* 48, 3 (2022), 803–818. https://doi.org/10.1109/TSE.2020.3002496

[35] William E. Howden. 1982. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering* SE-8, 4 (1982), 371–379.

[36] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering (ICSE'14)*. 435–445.

[37] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[38] René Just, Michael D Ernst, and Gordon Fraser. 2014. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*. 315–326.

[39] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. 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 (FSE'14)*. 654–665.

[40] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. 2022. Prioritizing Mutants to Guide Mutation Testing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, 1743–1754. https://doi.org/10.1145/3510003.3510187

[41] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. 2018. How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering* 23, 4 (2018), 2426–2463.

[42] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[43] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers. In *44th IEEE/ACM International Conference on Software Engineering (ICSE'22)*. to appear.

[44] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. 2019. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[45] Thomas Laurent, Mike Papadakis, Marinos Kintis, Christopher Henard, Yves Le Traon, and Anthony Ventresque. 2017. Assessing and improving the mutation testing practice of PIT. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 430–435.

[46] Isabella Laybourn. 2022. μ^2: Using Mutation Analysis to Guide Mutation-Based Fuzzing. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, 331–333. https://doi.org/10.1145/3510454.3522682

[47] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 216–226. https://doi.org/10.1145/2594291.2594334

[48] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.

[49] LLVM Compiler Infrastructure. 2016. libFuzzer. https://llvm.org/docs/LibFuzzer.html. Accessed February 11, 2022.

[50] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

[51] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2013. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering* 40, 1 (2013), 23–42.

[52] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).

[53] William M. McKeeman. 1998. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL* 10, 1 (1998), 100–107.

[54] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

[55] Hoang Lam Nguyen and Lars Grunske. 2022. BeDivFuzz: Integrating Behavioral Diversity into Generator-based Fuzzing. In *44th IEEE/ACM International Conference on Software Engineering (ICSE'22)*. to appear.

[56] Hoang Lam Nguyen, Nebras Nassar, Timo Kehrer, and Lars Grunske. 2020. MoFuzz: A fuzzer suite for testing model-driven software engineering tools. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1103–1115.

[57] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. 1996. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 2 (1996), 99–118.

[58] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-guided Property-based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. 398–401. https://doi.org/10.1145/3293882.3339002

[59] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, 329–340. https://doi.org/10.1145/3293882.3330576

[60] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 174 (2019), 29 pages. https://doi.org/10.1145/3360600

[61] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2019. ChocoPy: A Programming Language for Compilers Courses. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E* (Athens, Greece) *(SPLASH-E 2019)*. Association for Computing Machinery, 41–45. https://doi.org/10.1145/3358711.3361627

[62] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 354–365.

[63] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[64] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. 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, 537–548.

[65] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* (2019).

[66] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. 2019. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software* 157 (2019), 110388.

[67] Ruixiang Qian, Quanjun Zhang, Chunrong Fang, and Lihua Guo. 2022. Investigating Coverage Guided Fuzzing with Mutation Testing. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware* (Hohhot, China) *(Internetware '22)*. Association for Computing Machinery, 272–281. https://doi.org/10.1145/3545258.3545285

[68] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*. 861–875.

[69] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. 2020. Quickly generating diverse valid test inputs with reinforcement learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1410–1421.

[70] David Schuler and Andreas Zeller. 2009. Javalanche: Efficient Mutation Testing for Java. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, 297–298. https://doi.org/10.1145/1595696.1595750

[71] SQLite Authors. 2019. How SQLite is Tested. https://www.sqlite.org/testing.html#the_fuzzcheck_test_harness. https://web.archive.org/web/20200427011538/https://www.sqlite.org/testing.html#the_fuzzcheck_test_harness Retrieved August 31, 2022.

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

[73] Macario Polo Usaola and Pedro Reales Mateo. 2010. Mutation Testing Cost Reduction Techniques: A Survey. *IEEE Software* 27, 3 (2010), 80–86. https://doi.org/10.1109/MS.2010.79

[74] Vasudev Vikram, Rohan Padhye, and Koushik Sen. 2021. 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, 723–735. https://doi.org/10.1109/ICSE43902.2021.00072

[75] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *41st International Conference on Software Engineering (ICSE '19)*.

[76] Pengfei Wang, Xu Zhou, Kai Lu, Tai Yue, and Yingying Liu. 2022. SoK: The progress, challenges, and perspectives of directed greybox fuzzing. *arXiv preprint* (2022). https://doi.org/10.48550/arXiv.2005.11907

[77] Cheng Wen. 2022. Recent Papers Related To Fuzzing. https://wcventure.github.io/FuzzingPaper/. Retrieved March 16, 2022.

[78] Elaine J Weyuker. 1986. Axiomatizing software test data adequacy. *IEEE transactions on software engineering* 12 (1986), 1128–1138.

[79] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation (PLDI '11)*. Association for Computing Machinery, 283–294. https://doi.org/10.1145/1993498.1993532

[80] Michal Zalewski. 2014. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. Accessed February 11, 2022.

[81] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2018. Predictive mutation testing. *IEEE Transactions on Software Engineering* 45, 9 (2018), 898–918.

[82] Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 722–733.

[83] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, 2169–2182. https://doi.org/10.1145/3460120.3484596

Received 2023-02-16; accepted 2023-05-03