Guiding Greybox Fuzzing with Mutation Testing

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

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

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

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

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

***Abstract*—Greybox fuzzing is a popular approach for automatically generating test inputs for a program. A fuzzer-generated test-input corpus can be used for continuous regression testing. However, the generated inputs are often selected for code coverage, not their ability to detect potential faults. In the literature on test adequacy criteria, mutation testing has shown to be a promising alternative to code coverage: a test corpus can be evaluated on its ability to detect artificially injected faults. This paper is the first to investigate the idea of augmenting code coverage feedback with mutation analysis in the greybox fuzzing loop. We present Mu2, a greybox fuzzing procedure that saves new inputs if they increase the mutation score across all previously generated inputs. The paper describes Mu2's design decisions for (1) defining a test oracle for mutation testing and (2) efficiently executing a large number of program mutants in the fuzzing loop. We evaluate Mu2 against the state-of-the-art Zest fuzzer on five Java benchmarks and find that Mu2 is capable of producing a test-input corpus with higher mutation score, with limits to scalability for larger programs.**

I. Introduction

Greybox fuzzing [1], [2], [3], [4] and coverage-guided property testing [5], [6] have become increasingly popular for automated test-input generation. 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 performing random mutations on some input from the corpus. The mutated input is added to the corpus if the corresponding execution of the program under test increases code coverage.

Fuzzing is traditionally used to discover inputs that crash programs and reveal security vulnerabilities [7], [8], [9], [10], [11], [12], [13], [14], [15], [16]. In the absence of new bugs, fuzzers are evaluated based on code coverage achieved during the fuzzing campaign [17], [18]. However, in the vast majority of fuzzing research, the end goal is to find bugs now [16]; 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 [19], 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`"* [20]. Similarly, OpenSSL uses several distinct fuzzer-generated corpora and their corresponding fuzz drivers for continuous testing [21].

*These authors contributed equally to the paper.

However, are these test corpora really suited for regression testing? When using conventional greybox fuzzers, the only metric being targeted is code coverage. While code coverage is necessary to exercise program functionality, coverage alone does not provide confidence in fault detection ability [22].

Now, the technique of *mutation testing* [23], which evaluates the ability of tests to catch artificially injected bugs (a.k.a. *mutation analysis*), has shown promise in improving test-suite effectiveness [24], [25]. 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*. The 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. Recently, Vikram et al. [26] used mutation scores for evaluating the quality of a fuzzer-generated test corpus; however, their fuzzing algorithm still uses code coverage to guide input generation. A natural question thus arises: *can we*

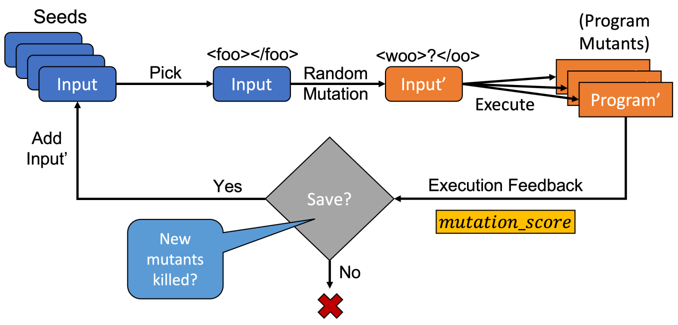

Fig. 1: A mutation-analysis-guided fuzzing loop. Each fuzzer-generated input is run through a set of program mutants to compute a mutation score. Inputs are saved to the corpus if they improve mutation score.

use mutation scores to guide the fuzzer?

This paper aims to combine the fields of greybox fuzzing and mutation testing, by investigating the use of mutation analysis in the fuzzing loop. 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).

We identify and address two main challenges that arise with this 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. Second, evaluating each fuzzer-generated input on the set of all program mutants is prohibitively expensive, thereby reducing fuzzing throughput.

To solve these challenges, we (1) introduce the idea of *differential mutation testing*, which validates the *output* of program execution, and (2) prune the set of mutants to run at each fuzzing iteration using dynamic analysis of the original program's execution.

We have implemented this idea in *Mu2*, a Java-based tool which incorporates program mutations from the popular PIT toolkit [27] into a custom guidance in the JQF [5] greybox fuzzing framework. Mu2 is open source and available at https://github.com/cmu-pasta/mu2.

We evaluate Mu2 on five real-world Java targets using state-of-the-art greybox fuzzer Zest [13], which is also built on top of the JQF framework, as a baseline. In particular, we compare the number of mutants killed by the fuzzer-generated test inputs by separately fixing the budget in terms of running time and the number of fuzzing trials.

Our results indicate: (1) Mu2 has a net improvement of 55 mutants killed across all experiments, which represents a 4% increase in mutation scores on average per benchmark; (2) the differential testing oracle is significantly valuable to Mu2, accounting for about 60% of all mutants killed; (3) on our largest benchmark (the Closure compiler), Mu2 underperforms Zest due to the performance overhead of mutation analysis—we identify that Mu2 can kill more mutants than Zest when controlling for this overhead, indicating the existence of a scalability trade-off. Our results open up new research frontiers for applying aggressive but approximate mutation selection and optimization techniques for augmenting mutation-analysis guided fuzzing.

To summarize, this paper makes the following contributions:

1) We present Mu2, the first combination of mutation testing and greybox fuzzing.
2) We propose *differential mutation testing* as an oracle and find that it significantly contributes to identifying mutant-killing inputs produced by fuzzing.
3) We describe the performance optimizations Mu2 employs to enable mutation analysis to run in the fuzzing loop.
4) We present an empirical evaluation of Mu2 on 5 real-world Java benchmarks, with Zest [13] as a baseline.

II. Background

A. 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 [2] and libFuzzer [3], but has since been heavily studied and variously extended in academic research [1], [10], [11], [12], [5], [13], [6], [4], [14], [15].

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 [1] based on data from preceding iterations of the fuzzing loop. 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 and data from preceding iterations. For example, in binary inputs, common mutations include random bitflips, inserting or deleting random chunks of bytes at randomly chosen offsets, random insertions of zeros or ones, and random insertions of "dictionary" values such as `INT32_MAX` or string tokens from some application domain. Structure-aware fuzzing tools [28], [13], [6], [29], [30] 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 [31] generator function. The program under test P is then executed with the newly generated 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 [19], [20], [21], [26], [32]. The quality of the final test-input corpus is often evaluated using code coverage [16], [18], though mutation scores—which we describe in the next section—have also been used [26].

Algorithm 1 Coverage-guided greybox fuzzing

```
procedure CGF(Program $P$, Set of inputs *seeds*, Budget $T$)
    $corpus \leftarrow seeds$                                  ▷ Initialize saved inputs
    repeat                                                     ▷ Fuzzing loop
        $x \leftarrow \textsc{PickInput}(corpus)$               ▷ Sample using heuristics
        $x' \leftarrow \textsc{MutateInput}(x)$                 ▷ Synthesize new input
        if running $P(x')$ leads to a crash then
            raise $x'$                                         ▷ Bug found!
        if $\textsc{coverage}(P, x') \not\subseteq \bigcup_{x \in corpus} \textsc{coverage}(P, x)$ then
            $corpus \leftarrow corpus \cup x'$
    until budget $T$
    return *corpus*                                            ▷ Final corpus
```

Algorithm 2 Mutation-analysis-guided fuzzing. Changes to Alg. 1 are highlighted.

```
procedure Mu2 (Program $P$, Set of inputs *seeds*, Budget $T$)
    $corpus \leftarrow seeds$
    repeat
        $x \leftarrow \textsc{PickInput}(corpus)$
        $x' \leftarrow \textsc{MutateInput}(x)$
        if $\textsc{coverage}(P, x') \not\subseteq \bigcup_{x \in corpus} \textsc{coverage}(P, x)$ then
            $corpus \leftarrow corpus \cup x'$
        for all $P' \in \textsc{ProgMutantsToRun}(P, corpus, x')$ do
            if $\textsc{kills}(P', x') \wedge P' \notin \textsc{killed}(P, corpus)$ then
                $corpus \leftarrow corpus \cup x'$
    until budget $T$
    return *corpus*
function KILLED(Program $P$, Set of inputs $X$)
    return $\{P' \mid P' \in \textsc{Mutants}(P) \wedge \exists x \in X : \textsc{kills}(P', x)\}$
```

B. 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*. The technique has long been studied in academia [33], having originally been developed in the 1970s to guide developers in designing a high quality test suite [23].

In assessing test adequacy [34], 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 [35] 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: $\frac{|\{P' \in \textsc{Mutants}(P) \mid \exists x \in X : \textsc{kills}(P', x)\}|}{|\textsc{Mutants}(P)|}$.

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.

There exists a vast amount of academic research on performing mutation testing, as surveyed by Papadakis et al. [36], addressing problems such as (a) *what mutations to perform?* (b) *how to detect equivalent mutants?* and (c) *how to speed up mutation testing?*

One of the most mature and actively developed mutation testing frameworks, PIT [27], 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).

These mutation operators have been chosen based on several empirical studies of effectiveness, sufficiency, and to align with developer expectations [27], [37], [38], [39].

III. Mutation-Analysis-Guided Fuzzing

A. Problem Statement

Can we use mutation analysis to guide greybox fuzzing for the purpose of synthesizing a test-input corpus with high mutation score?

In this paper, we work with the assumption that a high mutation score is a desirable property of a test-input corpus used for regression testing. While examining the relationship between mutation scores and real faults *is not in scope for this paper*, we refer the reader to several empirical studies on this topic [24], [25], [40], [41], [42], [43], [44].

B. Solution Approach

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 $\textsc{ProgMutantsToRun}$ (Line 8) returns a set of program mutants to evaluate with input x'. For now, assume it to return MUTANTS(P), which we defined in Section II-B. 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 will expand on the precise implementation of KILLS(P', x) in Section IV-A and we will refine the definition of PROGMUTANTSTORUN in Section IV-B.

IV. MU2: DESIGN AND CHALLENGES

We have implemented Algorithm 2 for fuzzing Java programs by integrating PIT [27] into JQF [5]. We call this system Mu2, since it mutates both the inputs and the programs under test.

We chose PIT and JQF because of their maturity, extensibility, and their common target platform. As described in Section II-B, PIT is an actively developed mutation testing framework that operates on JVM bytecode. The JQF framework [5] was originally designed for *coverage-guided property-based testing*, which is a structure-aware variant of greybox fuzzing (ref. Section II-A). JQF also instruments JVM bytecode for collecting code coverage. JQF has a highly extensible design for creating pluggable *guidances*, which supports rapid prototyping of new fuzzing algorithms [13], [45], [46], [47], [26], [32], [48].

In Mu2, MUTANTS(P) includes all of PIT's default expression mutation operators (ref. Section II-B). For other heuristics, such as PICKINPUT and MUTATEINPUT, Mu2 reuses the logic and code from Zest [13], which we also use as a baseline for evaluation (Section V).

A. 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 is known as the *test oracle*. In conventional software testing, a test method contains user-provided inputs and expected outputs, or *ground-truth*, which can be compared with actual outputs. This approach is known as using an *explicit oracle* [49].

For example, consider a JUnit test for the insertion sort method defined in Figure 2, written as below:

```
@Test // JUnit Test
void testInsertionSort() {
  int[] input = {42, 8, 23, 4, 16, 15};
  int[] output = {4, 8, 15, 16, 23, 42};
  assertEqual(output, Sort.insertionSort(input));
}
```

```
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 && (key < arr[i])) {
6          arr[i + 1] = arr[i];
7          i--;
8        }
9        arr[i + 1] = key;
10     }
11     return arr;
12   }}
```

Fig. 2: Java program that implements insertion sort.

The test passes if and only if the test method returns normally, without triggering an assertion violation.

Unfortunately, it is not possible to use this approach in greybox fuzzing because test inputs are randomly generated. Greybox fuzzing therefore generally relies on *implicit oracles* or *property tests* to determine if a test input causes a failure. Implicit oracles aim to detect anomalous behavior such as crashes or uncaught exceptions. For example, if the invocation of `Sort.insertionSort()` caused a `RuntimeException`, the test would fail based on the *implicit* oracle of terminating normally. A property test is an assertion over the output that must hold true for all inputs.

In contrast, consider the following method, which is written in the property-testing style using JQF's `@Fuzz` annotation:

```
@Fuzz // Inputs generated using greybox fuzzing
void fuzzInsertionSort(int[] input) {
  assert(isSorted(Sort.insertionSort(input)));
}
```

The method checks whether the array returned by `Sort.insertionSort()` is indeed sorted, where `isSorted` is a utility method (not shown here) that checks the sortedness of an array in a single pass. The input to this method is randomly generated by JQF, and the test fails if an assertion failure is triggered for *any* fuzzer-generated input.

For Mu2, could we use this same test driver to determine whether a program mutant should be killed? Consider the following examples: executing mutant $P'_1 = \langle \texttt{Sort}, \texttt{i+1}, \texttt{i}, 9 \rangle$ with input array $x =$ `[3, 2, 1]` would result in an uncaught `IndexOutOfBoundsException` (-1) on line 9, 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 $P'_3 = \langle \texttt{Sort}, \texttt{arr[i+1]=arr[i]}, \texttt{arr[i]=arr[i+1]}, 6 \rangle$, which swaps the array indices at line 6. This is clearly a bug in insertion sort, yet the output is always sorted. For example, when x = `[3, 2, 1]`, the result of $P'(x)$ is `[1, 1, 1]`.

```
@Diff // inputs generated by Mu2
int[] runInsertionSort(int[] input) {
    return Sort.insertionSort(input);
}

@Compare // outputs compared with mutant
boolean checkEq(int[] out1, int[] out2) {
    return Arrays.equals(out1, out2);
}
```

Fig. 3: A Mu2 differential mutation test driver and comparison method for the `insertionSort` method (Fig. 2).

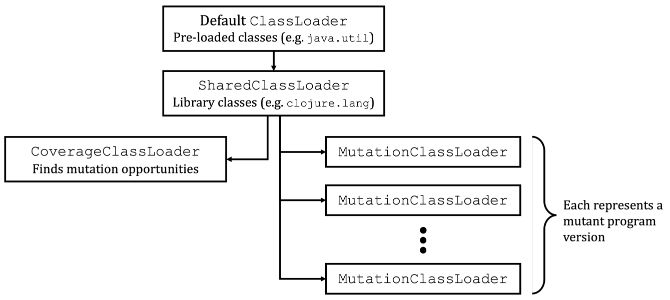
Default ClassLoader
Pre-loaded classes (e.g. java.util)
SharedClassLoader
Library classes (e.g. clojure.lang)
CoverageClassLoader
Finds mutation opportunities
MutationClassLoader
MutationClassLoader
MutationClassLoader
Each represents a mutant program version

Fig. 4: `ClassLoader` hierarchy in Mu2.

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 [49]. For many applications, a complete oracle would need to be as complex (or in some cases exactly the same) as the original program itself.

To solve this problem, we use the well-known concept of *differential testing* to define our oracle. In differential testing [50], [51], different implementations of a program that are expected to satisfy the same specification are executed on a single input. Discrepancies between outcomes observed when processing the same input are considered indications of bugs. Differential testing has been used successfully for fuzzing compilers and JVMs [52], [53], [54] where multiple programs implementing the same specification are available. In Mu2, our different "implementations" are the original program and program mutants; any discrepancy between the original program and a mutant 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` 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 [52].

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 $\textsc{Kills}(P', x)$ which was referenced in Algorithm 2. Given a mutant $P' = \langle P, e, e', n\rangle$ and an input x, $\textsc{Kills}(P', x)$ returns true iff:

1) $P(x) = y$ and $P'(x) = y$ and $\neg\textsc{Compare}(y, y')$, where $\textsc{Compare}$ 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) $P(x) = y$ but executing $P'(x)$ takes longer than a predefined `TIMEOUT` (10 seconds by default in Mu2).

The timeout is required for killing mutants such as $P'_4 = \langle \texttt{Sort}, \texttt{i--}, , 7\rangle$, which deletes the decrement of `i`, leading to an infinite loop on the input `[3, 1, 2]`.

When using Mu2 to generate tests for the Google Closure Compiler, the use of a differential testing oracle with a simple `String.equals()` comparison on the Closure-optimized JavaScript output improves the mutation score by 36% (66 additional mutants killed) as compared to the implicit oracle that was used for the same target in the Zest paper [13].

B. Performance

The biggest challenge with incorporating mutation testing inside a fuzzing loop is performance. Mutation testing is in general a very expensive technique. For example, the Jackson JSON parser can have 5000+ program mutants. Evaluating a fuzzer-generated test-input on this many program mutants can require more than 5000× the running time of a single program execution, thereby reducing the overall fuzzing throughput by a factor of 5000+. 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.

1) Improving performance of mutant execution: When running a mutation testing tool such as PIT [27], 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 (which may be unavoidable for system-level end-to-end tests). 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 `ClassLoader` mechanism to load

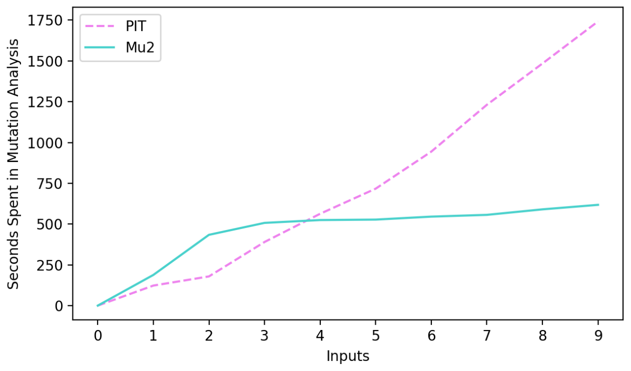

Fig. 5: Mu2 is more efficient and scalable than PIT in running a sequence of inputs with in-memory program mutants.

and run program mutants within the same JVM. First, the `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, the `MutationClassLoader` (MCL) is a classloader that loads a program mutant P'. 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'. The rest of the program is loaded as-is. Further, assuming that fuzz tests do not affect global state, Mu2 actually loads only one copy of each library class using a common `SharedClassLoader`. Figure 4 summarizes the classloader hierarchy design of Mu2.

To validate our design, we ran vanilla PIT on nine arbitrarily chosen inputs for the Google Closure Compiler. We compared the running time with that required to compute the mutation score for the same inputs using Mu2's design. In both cases, we specified that mutations must only be performed on classes from the package `com.google.javascript.jscomp`, which contains the core logic of compiler optimizations—with this filter, we get 1160 program mutants. Figure 5 shows the comparison of running times. The figure clearly shows that Mu2 scales better than PIT in running a sequence of inputs, having a much lower slope in the steady state. Without this optimization, it would be quite impractical to run mutation-analysis-guided fuzzing.

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:

$$\textit{trialTime} = \text{time}_{\text{orig}} + M * \text{avgTime}_{\text{mut}} \tag{1}$$

where $M = |\textsc{ProgMutantsToRun}(P, \textit{corpus}, x)|$ (ref. Algorithm 2).

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

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

```
1: function ProgMutantsToRun(Program $P$, Set of inputs $corpus$, New input $x$)
2:     $surviving \leftarrow \textsc{mutants}(P) \setminus \textsc{killed}(P, corpus)$
3:     return $\{P' = \langle P, e, e', n\rangle \mid (P' \in surviving) \wedge$
4:                 $(n \in \textsc{coverage}(P, x)) \wedge$
5:                 $(\textsc{infect}(P, e, e', x))\}$
```

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

1) If $P' \in \textsc{killed}(P, \textit{corpus})$, 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 [55].
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 [55], which is implemented as a dynamic analysis of the execution of the original program $P(x)$.

With these conditions, we can then define our ProgMutantsToRun function as shown in Algorithm 3. At line 2, $\textsc{killed}(P, \textit{corpus})$ is subtracted from the initial set of all mutants. The set of killed mutants is maintained in a data structure in Mu2 to avoid recomputing $\textsc{killed}(P, \textit{corpus})$ each trial. At line 4, program mutants are filtered to only those that mutate locations which are in $\textsc{coverage}(P, x)$. At line 5 of ProgMutantsToRun, the infect function uses dynamic analysis to check condition (3).

Implementing infection-based pruning in Mu2 requires significant additional instrumentation in the original program, since the infection values must be evaluated and compared each time that the expression e is executed. 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.

To determine the impact of infection-based pruning on trial time, we ran a set of preliminary experiments (3 hours, 10 repetitions) on 5 benchmarks with and without the infection optimization. The fuzzing throughput (inputs/sec), which is the reciprocal of average trial time, is visualized in Figure 6. In all benchmarks, infection-based pruning benefits Mu2's throughput and is thus included in the implementation.

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.

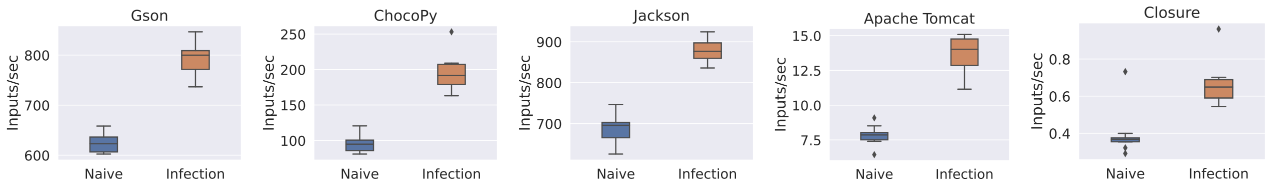

Fig. 6: Fuzzing throughput for each of the 5 benchmarks with and without the infection optimization (higher is better).

V. Evaluation

We evaluate Mu2 on 5 different Java program benchmarks, using state-of-the-art coverage-guided fuzzer Zest [13] as the baseline. We structure our evaluation around three research questions explained below:

RQ1: *Does mutation-analysis guidance produce a higher quality test-input corpus than coverage-only feedback in greybox fuzzing?*

To answer this question, we compare the mutants killed by the test-input corpora produced by Mu2 and Zest fuzzing campaigns that are run for equal amounts of time (24 hours). This question investigates the practical effectiveness of generating a test-input corpus with Mu2, including the overhead of performing mutation analysis in the fuzzing loop but also incorporating the performance optimizations described in Section IV-B.

RQ2: *What is the contribution of the differential oracle to the mutation scores of the fuzzer-generated corpora?*

To explore the effectiveness of the differential oracle (ref. Section IV-A) to mutation score, we report the distribution of mutants killed by the differential and implicit oracles. Our implicit oracle detects any uncaught Java exceptions that are not documented by the APIs invoked in the fuzz drivers (conventionally called “crashes” in fuzzing literature).

Additionally, we analyze examples of mutant and input pairs that are killed by the differential oracle but not by the implicit oracle. This provides an insight into why differential testing is a stronger oracle for mutation-analysis-guided fuzzing.

RQ3: *Does mutation-analysis guidance produce a higher quality test-input corpus over coverage-only feedback in greybox fuzzing* ***when fixing the number of trials?***

For this research question, the Zest fuzzing campaign is restricted to the same number of trials that Mu2 runs during the 24-hour time bound. Fixing the number of trials disregards the performance overhead of introducing mutation analysis into the fuzzing loop. Because it always takes Mu2 much longer to run the same number of trials as Zest, this evaluation provides an unfair advantage for Mu2 and is of course unrealistic. However, it allows us to measure the upper bound on the utility of mutation-analysis-guided fuzzing *alone*—that is, the benefit of saving inputs at Line 10 of Algorithm 2 assuming that the analysis conducted at Line 9 is free.

a) Benchmarks: We choose five real-world Java programs for our experiments:

While we note lines of code (LoC) for completeness, only a fraction of this code is reachable from fuzz drivers. Fig. 8 indicates actual code coverage.

1) Gson JSON Parser (˜26K LoC): The test driver parses a input JSON string and returns a Java class output.
2) Jackson JSON Parser (˜49K LoC): The test driver acts similar to that of Gson.
3) Apache Tomcat WebXML Parser (˜10K LoC): The test driver parses a string input and returns the WebXML representation of the parsed output.
4) ChocoPy [56] reference compiler (˜6K LoC): The test driver (reused from [26]) reads in a program in ChocoPy (a statically typed dialect of Python) and runs the semantic analysis stage of the ChocoPy reference compiler. It is modified to return a typed AST represented in JSON.
5) Google Closure Compiler (˜250K LoC): The test driver (reused from [13] and [26]) takes in a JavaScript program and performs source-to-source optimizations. It then returns the optimized JavaScript code.

b) Mutation selection: Following previous work on semantic fuzzing [13], [26], 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.

c) Duration: Following best practices [16], we use a time bound of 24 hours for our time-controlled experiments. For the fixed trial experiments, we use the same number of trials as the 24-hour Mu2 experiments for each Zest experiment.

d) Repetitions: To account for the randomness in fuzzing, we run each experiment 20 times and report aggregate metrics.

e) Metrics: For our evaluations, we compute the mutation scores and branch coverage across each fuzzer-generated test-input corpus. Rather than calculating the raw mutation score of each corpus, we report the *relative* improvement of mutation score. To show the difference in killed mutants between each corpus, we report the following two values:

1) The number of program mutants that were killed across *all* repetitions of one technique and *none* of the other.
2) The number of program mutants that were killed across *any* repetitions of one technique and *none* of the other.

Item 1 illustrates a consistency of one technique in killing particular mutants that always survive when using the other technique; item 2 is a superset of item 1 that includes any mutants across the repetitions that were killed. Additionally, we report the overall number of executed and killed mutants across all repetitions of both techniques to provide a sense of scale. Finally, we also report the relative branch coverage achieved for each benchmark.

A. RQ1

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

The results are shown in Table I. The first two columns show the total executed and killed mutants for each benchmark across all repetitions and using either technique. The last four columns display a difference in killed mutants across the repetitions of the experiments. For example, under "No Zest" and "All Mu2", the value describes the number of mutants killed during *no* repetitions of Zest and *all* repetitions of Mu2. We also use color coding to highlight cells where Mu2 performs better (green) or worse (red).

Looking at Table I, we see that Mu2 is able to produce a corpus that kills more mutants in ChocoPy and Jackson when alotted the same time budget as Zest. 19 of the mutants killed by *every* repetition of Mu2 in the Jackson benchmark survive in *all* repetitions of the Zest fuzzing campaigns. For Gson, there is 1 mutant killed by *all* Mu2 repetitions over Zest. We also noticed that for Tomcat, the killed mutants saturated at 239 in almost all of the repetitions of both Zest and Mu2, leading to no unique mutants being killed by either technique.

Figure 8 shows branch coverage of the generated corpora for these experiments. For the above mentioned 4 benchmarks, Mu2 achieves the same coverage as Zest. This shows that Mu2 can prove useful in a practical setting.

However, we do observe that the Zest corpora are able to kill 4 mutants that consistently survive in Mu2 corpora for the Closure benchmark. There is also a decrease in branch coverage for Closure—the Mu2 corpora on average achieve around 17% less branch coverage than Zest. We see later in Section V-C that this is due to the performance overhead of running mutation analysis during the fuzzing loop, since all of these mutants are killed in the fixed-trial experiments.

Out of all the 1526 killed mutants across benchmarks, Mu2 is able to kill 55 more mutants than Zest while achieving similar coverage in all but the largest benchmark. This corresponds to an average increase of 4% mutation score per benchmark. We thus conclude that *mutation-analysis guidance can help in producing a test-input corpus with higher mutation score than that produced by coverage-guided fuzzing; however, there are challenges with scalability when applying Mu2 to larger target programs.*

B. RQ2

What is the contribution of the differential oracle to the mutation scores of the fuzzer-generated corpora?

Figure 7 illustrates the distribution of mutants killed by each of the two oracles. We observe that across the 5 benchmarks, an average of 60% of the mutants are killed by the differential oracle. In Apache Tomcat, over 80% of the killed mutants are accounted for by the differential oracle. In Closure, while the contribution is smaller, the differential oracle still kills a non-trivial one-third of the mutants.

	Mutants Executed	Number of Mutants Killed By				
		Any	No Zest &		No Mu2 &	
			Any Mu2	All Mu2	Any Zest	All Zest
ChocoPy	345	299	7	0	0	0
Gson	338	305	2	1	2	0
Jackson	574	439	53	19	1	0
Tomcat	352	239	0	0	0	0
Closure	407	244	0	0	4	0

TABLE I: Mutant killing comparison between Zest and Mu2 run for equal time. The "All", "Any", and "No" modifiers refer to all repetitions of each experiment.

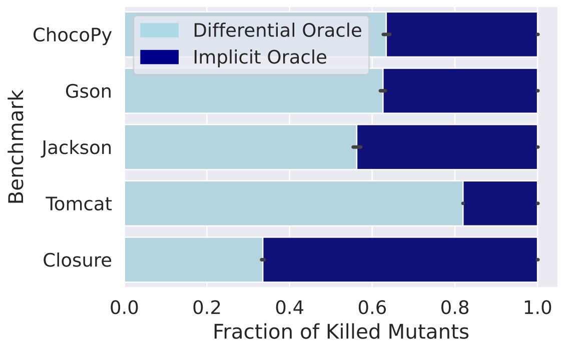

Fig. 7: Average fraction of mutants killed by the differential mutation testing oracle for all benchmarks. Error bars represent 95% confidence intervals.

To understand why the differential oracle contributes significantly to the fraction of killed mutants, we observe some examples of killed mutants and their corresponding test-inputs. For the sake of brevity, we describe a simplified version of the functionality of the code and omit the actual code snippets.

1) ChocoPy: The ChocoPy Type-checker returns an AST (represented by JSON) that contains specified types for each AST node and a list of type-checking errors. There is a function to check that the the left and right operand types match when using the "+" operator. If they do, the type is returned and assigned to the expression node in the AST; otherwise, an error message for the corresponding expression is added to the AST error list. Consider a mutant P' that modifies the return value of this function to return `null` instead of the correct type. Executing P' on a program x that contains the well-typed expression `[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 $P'(x)$ contains a type error, whereas $P(x)$ does not. The implicit oracle fails to kill this mutant since the type error is part of the output and does not trigger any uncaught exceptions. More fundamentally, any mutants that modify the ChocoPy logic to fail type-checking on well-typed programs cannot be killed by the implicit oracle!

2) Gson: The Gson parser contains logic within its function `readEscapeCharacter` to convert escaped Unicode sequences into characters. The function iterates over the characters in the sequence and offsets each by a constant before adding it to the output Unicode value. Suppose the mutant P' modifies `readEscapeCharacter` to change the direction of the offset to each character. For an input x that contains the Unicode sequence `"\uAAAA"`, $P'(x)$ produces a different output Unicode character than $P(x)$, but does not trigger any exceptions. By comparing the output characters, the differential oracle is able to kill P' while the implicit oracle does not.

3) Tomcat: The Apache Tomcat WebXML parser takes in an XML string as input and returns a representation of the parsed WebXML object. One function parses the `<url-pattern>` tags in the input and encodes the URL strings into UTF-8. A mutant P' modifies this function `encodeURL` to return an empty string instead of the encoded UTF-8 URL. When P' executes, the mutated empty string does not result in any exceptions, but it is propagated to the output WebXML representation. Thus, an XML input with a non-empty `<url-pattern>` tag is saved in Mu2 due to the differential oracle detecting the difference in the URL strings.

We conclude that a significant portion of mutants cause unexpected behavior related to the output of the program rather than crashes. This analysis shows how using the differential oracle over a traditional implicit oracle can enable fuzzers to save test inputs that exercise a larger set of interesting behaviors.

C. RQ3

Does mutation-analysis guidance produce a higher quality test-input corpus over coverage-only feedback in greybox fuzzing ***when fixing the number of trials?***

Looking at Table II, we see identical results to Table I for ChocoPy, Gson, Jackson, and Tomcat. The main improvement is observed in the Closure benchmark: we see that the Mu2 corpora are able to kill 3 mutants that survive in all repetitions of the Zest fixed-trial campaigns. Additionally, from Figure 8, we see difference in branch coverage is minimal in all the benchmarks between the Mu2 and the fixed-trial Zest corpora. Thus, we see room for further research in optimizations to Mu2 that will help the technique achieve higher performance.

From these results, we conclude that *under an ideal assumption of no overhead, mutation-analysis guidance helps in producing a test-input corpus with higher mutation score than that produced by coverage-guided fuzzing*. Section VIII discusses further implications of this result.

VI. Threats to Validity

a) Threats to construct validity: We identify two construct-validity threats to our measurement of mutation score. First, the measurement of mutation score is of course highly dependent on the set of mutation operators being applied to generate program mutants [57]. We aim to mitigate

	Mutants Executed	Number of Mutants Killed By				
		Any	No Zest &		No Mu2 &	
			Any Mu2	All Mu2	Any Zest	All Zest
ChocoPy	345	299	7	0	0	0
Gson	338	305	2	1	2	0
Jackson	574	439	53	19	1	0
Tomcat	352	239	0	0	0	0
Closure	403	240	3	0	0	0

TABLE II: Mutant killing comparison between Zest and Mu2 run for equal trials. The “All”, “Any”, and “No” modifiers refer to all repetitions of each experiment.

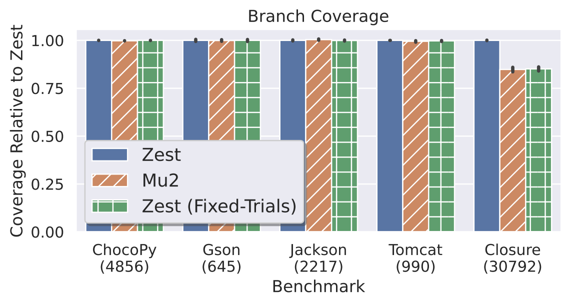

Fig. 8: Branch coverage across all benchmarks normalized by the coverage achieved by Zest. The number of branches covered by Zest (used to normalize) is listed below each target. Error bars represent 95% confidence intervals.

this threat by using the default set of operators in the widely used PIT framework (ref. Section II-B, which have been carefully chosen based on empirical studies of efficiency and sufficiency [27], [37], [38], [39]. Second, our test oracles (ref. Section IV-A) report an outcome of `TIMEOUT` if a mutant execution does not terminate within 10 seconds. Since we cannot solve the halting problem, such a bound is necessary to catch infinite loops (e.g., for mutants that negate loop conditions). However, it is possible that this bound is conservative and we accidentally mark some mutants 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 used manual analysis on a sample of the timeout-based kills to confirm correspondence to infinite loops.

b) Threats to internal validity: 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 conclusions, 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.

c) Threats to external validity: Since our implementation is based on JQF [5] and PIT [27], 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 [2] and libFuzzer [3]. 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 inputs and outputs 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 [49] is outside the scope of this paper.

VII. Related Work

a) Greybox fuzzing: The field of coverage-guided greybox fuzzing has a vast literature, as surveyed by Manès et al. [4]; a more recent and evolving publication list is maintained by Wen [58]. The majority of fuzzing research focuses on improving heuristics such as seed-picking power schedules [1], input mutations [17], [11], [12], and coverage feedback [10], [14]. FuzzFactory [59] 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. To the best of our knowledge, mutation testing has not been used to guide greybox fuzzing.

b) Improving the performance of mutation testing: A lot of research has been conducted to speed up mutation testing [33], [36], [60], [61], [62]. 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 [63]; we do not currently make an attempt to identify these statically. Just et al. [24] introduce the *propagation*, *infection*, *execution* (PIE) model to prune mutants using dynamic analysis. Mu2 implements the *execution* and *infection* optimizations from this work. MeMu [64] 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. [65] prioritize mutants to reach test completeness faster. All these optimizations are sound—they retain the accuracy of mutation scores.

Other research directions aim to optimize mutation-analysis time while potentially trading off accuracy or soundness. For example, weak mutation [66] has been proposed to terminate mutant evaluation quickly by observing the intermediate state after executing the mutated program locations. Predictive mutation testing [67] uses machine learning to estimate which mutants are most likely to be killed. Many techniques have been developed for *mutation reduction* [36], [60], [61]—where only a subset of mutants are evaluated based on some program-specific criteria. Recently, Guizzo et al. [68] have proposed an evolutionary approach to automate the generation of optimal cost reduction strategies. While Mu2 restricts itself to sound optimizations only, we believe that incorporating such aggressive optimizations into mutation-analysis-guided greybox fuzzing is a promising direction for future work.

c) Using mutation testing in automated test generation: μ-test [69] and EvoSuite [70] are both evolutionary test-generation techniques that can use mutation scores as an objective as well as a fitness function. Unlike these tools, which generate test *methods* for exercises program API, greybox fuzzing focuses on the generation of test inputs given a fixed entry point. Recently, Gopinath et al. [71] discussed various challenges with incorporating mutation analysis in fuzzing research. Our paper explicitly addresses two of these challenges: that of defining a strong test oracle and efficiently executing mutants in the fuzzing loop. In a registered report, Groce et al. [72] propose to split fuzzing resources between the original software and mutants of the software to reach deeper program behaviors; however, they do not use mutant-killing ability of inputs as a guidance for the fuzzing algorithm.

VIII. Conclusion and Future Work

We presented the first technique that uses 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.

Our results indicate that using mutation analysis as a saving criteria for inputs is a promising approach to improve test-input corpus quality. Additionally, the differential oracle provides Mu2 with a strong way of detecting interesting behavior in program mutant execution. However, there are scalability concerns when running Mu2 on larger programs, chiefly due to the limitation on fuzzing throughput when performing mutation analysis in the loop.

In our design, we proposed optimizations to improve fuzzing throughput by dynamically pruning the number of mutants to be executed. We restricted ourselves to *sound* optimizations, where mutants are pruned only if we can guarantee that they will not be killed by a given input. If the constraints of such soundness guarantees are relaxed, there is opportunity to apply more aggressive optimizations. In section VII, we discussed a number of *unsound* optimizations [36][60][61][66][67][68]—such as using machine learning or weak mutation to predict the likelihood of a mutant being killed—which can improve the performance of mutation testing at the potential cost of discarding some mutant-killing inputs. There is a rich trade-off space between accuracy of such prediction and the scalability of the overall technique. We believe that our results open up new frontiers for future work to investigate the applicability of such advanced optimizations in mutation-analysis-guided fuzzing.

References

[1] 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.

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

[3] LLVM Compiler Infrastructure, “libfuzzer,” https://llvm.org/docs/LibFuzzer.html, 2016, accessed February 11, 2022.

[4] 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.

[5] 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, 2019, pp. 398–401. [Online]. Available: http://doi.acm.org/10.1145/3293882.3339002

[6] 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, 2019.

[7] 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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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. New York, NY, USA: ACM, 2019, pp. 329–340. [Online]. Available: http://doi.acm.org/10.1145/3293882.3330576

[14] 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.

[15] 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, 2021, pp. 131–142.

[16] 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.

[17] 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.

[18] 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.

[19] 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

[20] 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

[21] 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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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

[27] 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.

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

[29] J. Wang, B. Chen, L. Wei, and Y. Liu, “Superion: Grammar-aware greybox fuzzing,” in *41st International Conference on Software Engineering*, ser. ICSE '19, 2019.

[30] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, “Smart greybox fuzzing,” *IEEE Transactions on Software Engineering*, 2019.

[31] K. Claessen and J. Hughes, “Quickcheck: A lightweight tool for random testing of haskell programs,” in *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP, 2000.

[32] H. L. Nguyen and L. Grunske, “BeDivFuzz: Integrating behavioral diversity into generator-based fuzzing,” in *44th IEEE/ACM International Conference on Software Engineering*, ser. ICSE'22, 2022, to appear.

[33] 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.

[34] J. B. Goodenough and S. L. Gerhart, “Toward a theory of test data selection,” *IEEE Transactions on software Engineering*, no. 2, pp. 156–173, 1975.

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

[36] 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.

[37] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, “An experimental determination of sufficient mutant operators,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[38] 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

[39] 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.

[40] 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. New York, NY, USA: Association for Computing Machinery, 2005, p. 402–411. [Online]. Available: https://doi.org/10.1145/1062455.1062530

[41] 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.

[42] 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.

[43] 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.

[44] 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. New York, NY, USA: Association for Computing Machinery, 2020, p. 237–249. [Online]. Available: https://doi.org/10.1145/3324884.3416667

[45] 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.

[46] 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.

[47] 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.

[48] 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, to appear.

[49] 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.

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

[51] 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.

[52] 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

[53] 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, New York, NY, USA, 2014, p. 216–226. [Online]. Available: https://doi.org/10.1145/2594291.2594334

[54] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[55] 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.

[56] 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. New York, NY, USA: Association for Computing Machinery, 2019, p. 41–45. [Online]. Available: https://doi.org/10.1145/3358711.3361627

[57] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 354–365.

[58] C. Wen, "Recent papers related to fuzzing," https://wcventure.github.io/FuzzingPaper/, 2022, retrieved March 16, 2022.

[59] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "Fuzzfactory: domain-specific fuzzing with waypoints," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019. [Online]. Available: https://doi.org/10.1145/3360600

[60] 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.

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

[62] F. Cutigi Ferrari, A. Viola Pizzoleto, and J. Offutt, "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)*, 2018, pp. 1–10.

[63] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 23–42, 2013.

[64] A. Ghanbari and A. Marcus, "Faster mutation analysis with memu," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 781–784. [Online]. Available: https://doi.org/10.1145/3533767.3543288

[65] 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. New York, NY, USA: Association for Computing Machinery, 2022, p. 1743–1754. [Online]. Available: https://doi.org/10.1145/3510003.3510187

[66] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 371–379, 1982.

[67] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 898–918, 2018.

[68] G. Guizzo, F. Sarro, J. Krinke, and S. R. Vergilio, "Sentinel: A hyper-heuristic for the generation of mutant reduction strategies," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 803–818, 2022.

[69] 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

[70] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11, 2011.

[71] 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

[72] A. Groce, G. T. Kalburgi, C. Le Goues, K. Jain, and R. Gopinath, "Registered report: First, fuzz the mutants," in *International Fuzzing Workshop*, ser. FUZZING'22, 2022.