Guiding Greybox Fuzzing with Mutation Testing

Isabella Laybourn*
ilaybour@andrew.cmu.edu
Carnegie Mellon University
United States

Vasudev Vikram*
vasumv@cmu.edu
Carnegie Mellon University
United States

Rafaello Sanna
rsanna@u.rochester.edu
University of Rochester
United States

Ao Li
aoli@cs.cmu.edu
Carnegie Mellon University
United States

Rohan Padhye
rohanpadhye@cmu.edu
Carnegie Mellon University
United States

ABSTRACT

Greybox fuzzing techniques commonly use execution feedback such as code coverage to guide the evolutionary search of test inputs. Mutation testing has been shown to be a superior alternative to code coverage as a test adequacy criteria. This paper investigates the idea of augmenting code coverage feedback with mutation analysis in guiding greybox fuzzing with the goal of producing a high quality test-input corpus. Mu2 is a greybox fuzzing procedure that augments the code coverage feedback to save new inputs if they increase the mutation score across all previously generated inputs. We describe and present solutions to the challenges of (1) defining a proper test oracle to run 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 greybox fuzzer Zest on five Java benchmarks. Our results demonstrate that mutation-analysis guidance is capable of producing a test-input corpus with higher mutation score, but there remains room for future work to improve the scalability of the technique for large programs.

ACM Reference Format:
Isabella Laybourn, Vasudev Vikram, Rafaello Sanna, Ao Li, and Rohan Padhye. 2022. Guiding Greybox Fuzzing with Mutation Testing. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, greybox fuzzing [1–4] and its cousin 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.

*These authors contributed equally to the paper.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Fuzzing is traditionally used to discover inputs that crash programs and reveal security vulnerabilities [7–16]. In the absence of new bugs, fuzzers are evaluated based on code coverage achieved during the fuzzing campaign [16, 17]. In fact, some recent work [18, 19] has explicitly focused on generating a high quality corpus of test inputs as the final outcome of the fuzzing campaign. A fixed corpus of auto-generated inputs can be used for quick regression testing as the program under test evolves.

Despite its widespread use, empirical studies have shown that code coverage is not ideal for assessing test suite effectiveness [20] or for comparing the performance of multiple fuzzers [21].

In studies of handwritten tests, the technique of *mutation testing* (a.k.a. *mutation analysis*) [22], which evaluates the ability of tests to catch artificially injected bugs, has shown promise in improving fault detection ability [23–25]. Yet, as Gopinath et al. [26] have recently pointed out, mutation analysis is not commonly used in greybox fuzzing research. We are only aware of one fuzzing paper [18] that uses mutation scores for evaluating test corpus quality, even though the presented algorithm itself only takes into account code coverage as a fitness function.

A natural question thus arises: *if our objective is to produce a test-input corpus with high mutation score, can we use mutation scores as a fitness function for greybox fuzzing?*

This paper investigates the potential of using mutation analysis in guiding greybox fuzzing. 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*

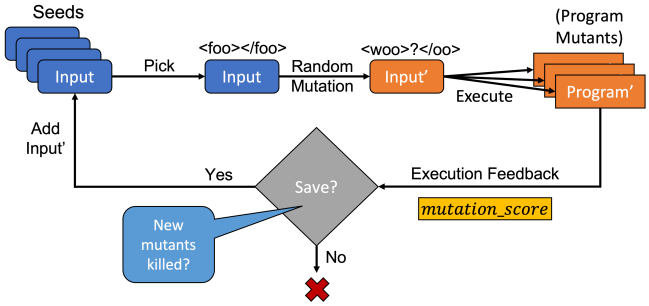

Figure 1: A mutation-analysis-guided fuzzing loop. Each fuzzer-generated input is run through a set of program mutants to compute a mutation score. Inputs are saved to the corpus if they improve mutation score.

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 have implemented this idea for Java by incorporating program mutations from the popular PIT toolkit [27] into a custom guidance in the JQF [5] greybox fuzzing framework. We call our implementation *Mu2*. During this endeavour, we identified several challenges and opportunities that are unique to the problem of incorporating mutation analysis in a fuzzing loop. First, we need a good test oracle beyond identifying program crashes or assertion violations, as is common with conventional greybox fuzzing or property testing. Second, evaluating each fuzzer-generated input on the set of all program mutants is prohibitively expensive, dramatically reducing fuzzing throughput. Further, for scaling to large programs, we need a way to keep the set of all program mutants in memory, ready to execute new inputs. To solve these challenges, we (1) introduce the idea of *differential mutation testing*, (2) dynamically prune the set of mutants to run at each fuzzing iteration and reduce the memory footprint of program mutants via shared loading of library classes.

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—both during the fuzzing loop and by the final corpus—by separately fixing the budget in number of fuzzing trials and time budget.

Our results indicate: (1) when fixing the number of trials, mutation-analysis guidance produces a test-input corpus that consistently kills more mutants than the corpus produced by only coverage-guided fuzzing; (2) when fixing the time budget, Mu2 still kills more mutants than Zest in three of five benchmarks, but performs significantly worse on the largest benchmark (Closure), indicating that more work is needed to make this technique scalable; (3) in its fuzzing loop, Zest actually generates all mutant killing inputs that Mu2 would have saved, but Zest does not have enough information to save these inputs.

To the best of our knowledge, this is the *first* work to incorporate mutation testing into greybox fuzzing. Importantly, this paper does not claim that using mutation testing as a drop-in replacement for code coverage in greybox fuzzing is a superior approach for bug finding. Rather, we hope that the insights presented in this paper will provide support for further research incorporating mutation-killing ability as one of the heuristics in greybox fuzzing, and for further research in improving the scalability of performing mutation analysis in a fuzzing loop.

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 [2] and libFuzzer [3], but has since been heavily studied and variously extended in academic research [1, 4–6, 10–15].

Algorithm 1 Coverage-guided greybox fuzzing

1: **procedure** CGF(Program P, Set of inputs *seeds*, Budget T)
2: $corpus \leftarrow seeds$ ▷ Initialize saved inputs
3: **repeat** ▷ Fuzzing loop
4: $x \leftarrow \textsc{PickInput}^1 corpus^0$ ▷ Sample using heuristics
5: $x^0 \leftarrow \textsc{MutateInput}^1 x^0$ ▷ Synthesize new input
6: **if** running $P^1 x^{0_0}$ leads to a crash **then**
7: **raise** x^0 ▷ Bug found!
8: **if** $\textsc{coverage}^1 P, x^{0_0} \notin \bigcup_{x 2 corpus} \textsc{coverage}^1 P, x^0$ **then**
9: $corpus \leftarrow corpus \cup [x^0]$
10: **until** budget T
11: **return** *corpus* ▷ Final corpus

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^0 (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. Similarly, the random mutations performed on x to get x^0 (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 [6, 13, 28–30] 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^0, 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^0 causes new code to be covered (Line 8), then x^0 is saved to the corpus (Line 9); thus, x^0 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^0 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 the number of fuzzing trials (i.e., iterations of the fuzzing loop) or in terms of wall-clock time. The test-input corpus containing all fuzzer-synthesized 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 [18, 19]. The quality of the final test-input corpus is often evaluated using code coverage [16, 21], though mutation scores—which we describe in the next section—have also been used [18].

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*. The technique has long been studied in academia [32], having originally been developed in the 1970s to guide developers in designing a high quality test suite [22].

In the test adequacy problem, 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 [33] with additions to the set X. *Code coverage* is an example of a test adequacy criteria.

In mutation testing, a set of program mutants ($\textsc{Mutants}^1P^0$) is first generated. Each mutant $P^0 \in \textsc{Mutants}^1P^0$ 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 $\hbar P, \texttt{a+b}, \texttt{a-b}, 42\imath$ to refer to this mutation. For purposes of this paper, we use the notation:

$$P^0 = \hbar P, e, e^0, n\imath$$

to refer to a program mutant P^0 as a modification of program P where expression e is replaced with e^0 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^0. If some test $x \in X$ fails when run on mutant P^0, then the mutant P^0 is said to be *killed*, which we denote as $\textsc{Kills}^1P^0, x^0$. If the entire test suite X still passes, then the mutant P^0 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^0. So, the adequacy of test suite X is defined by the *mutation score*, which is computed as the fraction of mutants killed: $\frac{|\{P^0 \in \textsc{Mutants}^1P^0 \mid \exists x \in X : \textsc{kills}^1P^0, x^0\}|}{|\textsc{Mutants}^1P^0|}$.

In general, a mutation score of 100% is rarely achievable because some mutants P^0 may actually be *equivalent* to P—that is, $\forall x : P^1x^0 = P^0{}^1x^0$. 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. Nonetheless, empirical studies have also shown that there exists a positive relationship between high mutation scores and fault-detection ability of the test suite [23–25].

There exists a vast amount of academic research on performing mutation testing, as surveyed by Papadakis et al. [34], 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 set of 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

Given that greybox fuzzing depends on an adequacy criteria ("coverage") to decide if new inputs should be saved to the corpus, and given the over four decades of research in improving test adequacy criteria using mutation analysis, we simply ask: *can we use mutation testing to guide greybox fuzzing?*

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 ← PickInput¹corpus⁰
5:          x⁰ ← MutateInput¹x⁰
6:          if coverage¹P, x⁰⁰ ∉ ⋃_{x∈corpus} coverage¹P, x⁰ then
7:              corpus ← corpus ∪ { x⁰ }
8:          for all P⁰ ∈ ProgMutantsToRun¹P, corpus, x⁰⁰ do
9:              if kills¹P⁰, x⁰⁰ ∧ P⁰ ∉ killed¹P, corpus⁰ then
10:                 corpus ← corpus ∪ { x⁰ }
11:     until budget T
12:     return corpus
13: function killed(Program P, Set of inputs X)
14:     return {P⁰ | P⁰ ∈ Mutants¹P⁰ ∧ ∃x ∈ X : kills¹P⁰, x⁰}
```

In response, 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^0 should be saved to the corpus. The function ProgMutantsToRun (Line 8) returns a set of program mutants to evaluate with input x^0. For now, assume it to return Mutants(P), which we defined in Section 2.2. We then determine whether the input x^0 is the first input to kill some mutant P^0. If P^0 is killed by x^0 and P^0 has not previously been killed by any input in the *corpus* (Lines 9 and 14), then we add x^0 to the corpus (Line 10). Broadly, this algorithm saves fuzzer-generated inputs if they increase either code coverage or mutation score.

While the procedure in Algorithm 2 appears to be a straightforward extension to the CGF algorithm, there are some practical challenges that arise when introducing mutation analysis into the fuzzing loop. In the following section, we will describe our implementation, the unique challenges that arise, and our corresponding solutions. In particular, we will expand on the precise implementation of $\textsc{kills}^1P^0, x^0$ in Section 4.1 and we will refine the definition of ProgMutantsToRun in Section 4.2.

4 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. The implementation is open source and available at: https://github.com/cmu-pasta/mu2.

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 [5] was originally designed for *coverage-guided property-based testing*, which is a structure-aware variant of greybox fuzzing (ref. Section 2.1). 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, 18, 19, 35–38].

```
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   }}
```

Figure 2: Java program that implements insertion sort.

```
1  @Diff // inputs generated by Mu2
2  int[] runInsertionSort(int[] input) {
3      return Sort.insertionSort(input);
4  }
5
6  @Compare // outputs compared with mutant
7  boolean checkEq(int[] out1, int[] out2) {
8      return Arrays.equals(out1, out2);
9  }
```

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

In Mu2, MUTANTS$^1P^0$ includes all of PIT's default expression mutation operators (ref. Section 2.2). 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 5).

4.1 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^0 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* [39].

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

```
1  @Test // JUnit Test
2  void testInsertionSort() {
3    int[] input = {42, 8, 23, 4, 16, 15};
4    int[] output = {4, 8, 15, 16, 23, 42};
5    assertEquals(output, Sort.insertionSort(input));
6  }
```

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:

```
1  @Fuzz // Inputs generated using greybox fuzzing
2  void fuzzInsertionSort(int[] input) {
3    assertTrue(isSorted(Sort.insertionSort(input)));
4  }
```

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_1^0 = ℏ`Sort`, `i+1`, `i`, 9`i` 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_2^0 = ℏ`Sort`, `i>=0`, `i>0`, 5`i` with x would result in an assertion failure in the property test because the result of $P_2^0{}^1x^0$ would be the array `[3, 1, 2]`, which is not sorted. So, both mutants P_1^0 and P_2^0 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_3^0 = ℏ`Sort`, `arr[i+1]=arr[i]`, `arr[i]=arr[i+1]`, 6`i`, 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 $P^0{}^1x^0$ 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 [39]. 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 [40, 41], 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 [42–44] 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 [42].

With differential mutation testing, we are able to kill mutants such as P_3^0 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 $P^0 = \hbar P, e, e^0, n\mathtt{i}$ and an input x, Kills(P', x) returns true iff:

(1) $P^1x^o = y$ and $P^{0\,1}x^o = y$ and $:\textsc{Compare}^1 y, y^{0o}$, where Compare is the user-defined `@Compare` method (e.g. `checkEq` in Figure 3) or `Object.equals()` if one is not defined; or
(2) $P^1x^o = y$ but executing $P^{0\,1}x^o$ results in an uncaught runtime exception being thrown; or
(3) $P^1x^o = y$ but executing $P^{0\,1}x^o$ takes longer than a predefined `TIMEOUT`, which is 10 seconds by default in Mu2.

The timeout is required for killing mutants such as $P_4^0 = \hbar\mathtt{Sort}, \mathtt{i\text{-}\text{-}}, , 7\mathtt{i}$, 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].

4.2 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 Google Closure Compiler has 1164 program mutants even when restricting mutations to only the package containing optimization logic. Evaluating a fuzzer-generated test-input on 1000+ program mutants can require more than 1000 the running time of a single program execution, thereby reducing the overall fuzzing throughput by a factor of 1000+. 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.

4.2.1 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.

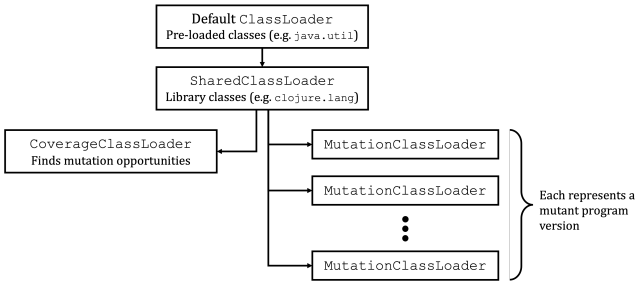

Figure 4: `ClassLoader` hierarchy in Mu2.

Mu2 thus adopts a different strategy than PIT and takes advantage of the Java `ClassLoader` mechanism to load and run program mutants within the same JVM. In particular, Mu2 keeps all program mutants loaded and in-memory for the entire fuzzing campaign, ready to execute on each fuzzer-generated input.

We first define two subclasses of `java.lang.ClassLoader` that perform different tasks:

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^1x^o.

Second, the `MutationClassLoader` (MCL) is a classloader that loads a program mutant P^0. In fact, Mu2 creates as many instances of MCL as there are $\textsc{Mutants}^1P^o$, with each MCL corresponding to one program mutant $P^0 = \hbar P, e, e^0, n\mathtt{i}$. 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^0. The rest of the program is loaded as-is.

This design is fairly non-trivial because it means that in a single JVM, there are M ¸ 1 copies of every Java class in the target program (identified by its fully qualified name), where M is the number of program mutants. Further, much more memory is required to keep all these duplicate classes loaded throughout the fuzzing campaign. For example, the Google Closure Compiler loads 966 application and library classes when fuzzing with Zest. For running on 1160 program mutants, Mu2 requires loading over 1 million classes in memory! A naive implementation triggers `OutOfMemory` errors related to the JVM class metaspace.

Mu2 thus performs an additional optimization for library classes, which are defined as classes that do not transitively depend on any mutated classes via static linking. Library classes do not have to be loaded M ¸ 1 times for M mutants. Assuming that fuzz tests do not affect global state, Mu2 actually loads only one copy of each such library class. This single copy is shared between the original program P and all program mutants P^0. Such classes are loaded by a common `SharedClassLoader`. Figure 4 summarizes the classloader hierarchy design of Mu2.

But is this all worth it? 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

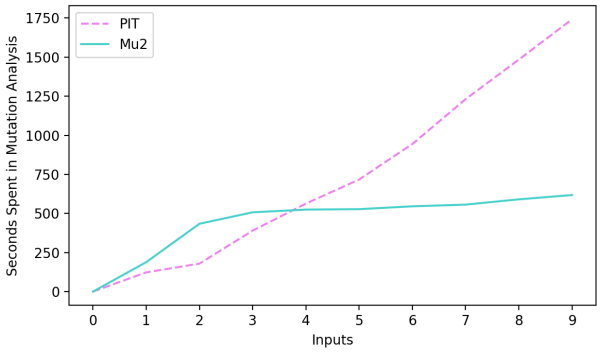

Figure 5: Mu2 is more efficient and scalable than PIT in running a sequence of inputs with in-memory program mutants.

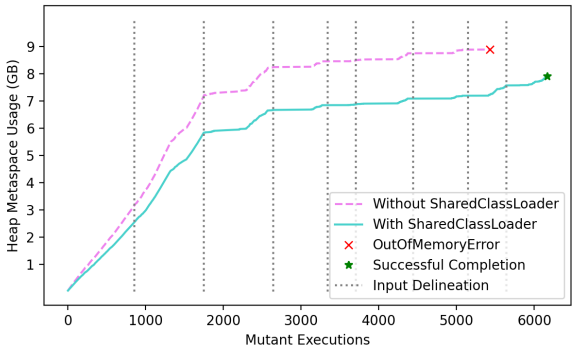

Figure 6: Loading library classes with the `SharedClassLoader` prevents `OutOfMemoryError` related to the JVM class metaspace.

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 optimization—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.

Figure 6 also demonstrates the memory issues when running the same nine inputs for Closure with and without the `SharedClassLoader` optimization. By using the `SharedClassLoader` to load a single copy of library classes, Mu2 is able to scale to programs with 1000 classes.

4.2.2 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 CCL, (2) the input must be executed by each MCL, and (3) the KILLS oracle must be run on each of the mutant results. Thus, we can model the time required to execute each trial as the following:

$$trialTime = \text{time}_{CCL} \ , \ M \quad {}^{1}\text{avgTime}_{MCL} \ , \ \text{avgTime}_{\text{KILLS}}{}^{o}$$

where M = jProgMutantsToRun^{1}P, *corpus*, x^{o}j (ref. Algorithm 2).

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: *survivingMutants* mutants$^{1}P^{o}$ ∩ killed^{1}P, *corpus*^{o}
3: **return** fP^{0} = ℏP, e, e^{0}, ni j $^{1}P^{0}$ 2 *survivingMutants*^{o} ^ ^{1}n 2 coverage$^{1}P, x^{oo}$g

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 mutant to kill it. We can therefore reduce M by dynamically pruning mutants whose execution will necessarily lead to Line 9 evaluating to *false*:

(1) If P^{0} 2 killed^{1}P, *corpus*^{o}, then P^{0} does not need to be executed for any future inputs.
(2) If the program mutant P^{0} applies a mutation to a program location n, but n is *not covered* when executing the original program on x, then P^{0} cannot be killed by x. This corresponds to the *execution*-based pruning in the PIE model [45].

Utilizing these optimizations, we can then define our ProgMutantsToRun function as shown in Algorithm 3. At line 2, killed^{1}P, *corpus*^{o} 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 killed^{1}P, *corpus*^{o} each trial. At line 3, program mutants are filtered to only those that mutate locations which are in coverage$^{1}P, x^{o}$. Mu2 implements this by having the `CoverageClassLoader` instrument classes to collect coverage during the run of the original program.

As an example of the execution optimization, we can return to our insertion sort example from Figure 2. Reusing our mutant P_{3}^{0} = ℏ`Sort`, `arr[i+1]=arr[i]`, `arr[i]=arr[i+1]`, 6i, suppose we have a fuzzer-generated input x = »1, 2, 3¼ (i.e., a presorted array). When the CCL loads `Sort` to execute x on the original program, the while loop at line 5 of Fig. 2 is not entered, and therefore line 6 is not covered at all. Thus, program mutant $P_{3}^{0} \notin$ ProgMutantsToRun^{1}`Sort`, ;, x^{o}; that is, P_{3}^{0} will be skipped when evaluating x in the fuzzing loop regardless of the corpus collected so far. The *execution optimization* provides a significant benefit: for the Jackson benchmark, the optimizations in a reduction from M = 5657 to M = 600, almost a 10x difference!

4.2.3 Parallelizability of mutant evaluation. We also note that executing an input on many program mutants is embarrassingly parallelizable—a thread can be spawned to run each mutant independently. For a thread-pool with k available threads, our trial time calculation can then be refined as follows:

$$trialTime = \text{time}_{CCL} \ , \ {}^{1}M \bullet k^{o} \quad {}^{1}\text{avgTime}_{MCL} \ , \ \text{avgTime}_{\text{KILLS}}{}^{o}$$

5 EVALUATION

We evaluate mutation-analysis-guided fuzzing on 5 different Java program benchmarks, using state-of-the-art 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. The Zest fuzzing campaign runs for a specified time bound (detailed later) to produce the baseline. Mu2 is run for the equivalent number of *trials*. Controlling for trials disregards the performance overhead of introducing mutation analysis into the fuzzing loop; it always takes Mu2 much longer to run the same number of trials as Zest. So, 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.

RQ2: *Does mutation-analysis guidance produce a higher quality test-input corpus over coverage-only feedback in greybox fuzzing* ***when controlling for time?***

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 4.2. To study this, we restrict Mu2 to run for the same amount of time as Zest, and evaluate the final test-input corpus produced by each—as such, this is a fair evaluation of our current implementation.

RQ3: *Is mutation-analysis guidance able to more effectively generate mutant-killing inputs during the fuzzing campaign than coverage-only guidance?*

As opposed to RQ1 and RQ2, which evaluate the ability of each guidance to synthesize a high-quality test-input corpus at the end of fuzzing, RQ3 focuses on the ability to discover higher quality inputs during fuzzing. Is it possible that Zest generates mutant-killing inputs during its fuzzing loop (Algorithm 1), but does not save them because it does not perform mutation analysis? Another way to think about this question is as follows: if we use program mutants as proxies for "bugs", then how does Mu2 compare to Zest on finding these bugs *during* the fuzzing loop? To answer this, we implement a new baseline Zest*, which calculates the mutation scores of *all* inputs that are generated (not just saved) during the campaign. This provides an unfair advantage to the coverage-guided fuzzing baseline, since it requires calculating the mutation score of $\textsc{Mutants}^1P^0$ across all generated inputs, a very expensive process. Zest* thus measures the *upper bound* on the mutation score of Zest.

Benchmarks. We choose five real-world Java programs that feature a mix of syntactic parsing and semantic logic. They are: [1]

(1) Gson JSON Parser (~26K LoC): The test driver parses a input JSON string and returns an output Java class output.
(2) Jackson JSON Parser (~49K LoC): The test driver acts similar to that of Gson.
(3) Apache Tomcat (~350K LoC): The test driver parses a string input and returns the WebXML representation of the parsed output.
(4) ChocoPy [46] reference compiler (~6K LoC): The test driver (reused from [18]) 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 [18]) takes in a JavaScript program and performs source-to-source optimizations. It then returns the optimized JavaScript code.

The input generators, differential test oracles, and the package name filters for applying mutation testing are provided with the associated artifact. We use the same generators, oracles, and filters for both Zest and Mu2.

Duration. We choose a time bound of 3 hours for our baseline Zest algorithm to be consistent with prior work using the baseline [13, 18, 19], which also shows that coverage usually saturates for Zest within this time bound. For trial-controlled experiments, we use the same number of trials as the Zest experiments to run for each Mu2 experiment. For time-controlled experiments, Mu2 also uses a time bound of 3 hours and is limited to using only one thread. Note that actually computing mutation scores for trial-controlled experiments is very resource intensive; for example, running Zest* on Closure for the same number of trials as a 3-hour Zest run requires more than one week!

Repetitions. To account for the randomness in fuzzing, we run each experiment 10–20 times and report aggregate metrics.

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 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, calculated using the third-party JaCoCo library [47].

5.1 RQ1

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

We seek to answer RQ1 by evaluating the final test-input corpus produced by Mu2 compared to that produced by Zest when they are both run for the same number of trials. Though this is unrealistic—it ignores the performance overhead of mutation analysis—an improvement in mutation score would demonstrate the value of mutation-analysis guidance.

The results are shown in Table 1. The first two columns show the total executed and killed mutants for each benchmark across

[1] While we note lines of code (LoC) for completeness, only a fraction of this code is reachable from our fuzzing drivers. Fig. 7 indicates actual code coverage.

	Mutants Executed	Number of Mutants Killed By				
		Any	No Zest &		No Mu2 &	
			Any Mu2	All Mu2	Any Zest	All Zest
ChocoPy	406	302	5	0	0	0
Gson	412	305	3	1	0	0
Jackson	602	449	49	25	0	0
Tomcat	389	252	0	0	0	0
Closure	503	249	3	0	1	0

Table 1: Mutant killing comparison between Zest and Mu2 (trial controlled). The "All", "Any", and "No" modifiers refer to all repetitions of each experiment.

	Mutants Executed	Number of Mutants Killed By				
		Any	No Zest &		No Mu2 &	
			Any Mu2	All Mu2	Any Zest	All Zest
ChocoPy	406	302	5	0	0	0
Gson	412	307	1	1	0	0
Jackson	602	449	49	22	1	0
Tomcat	389	252	0	0	0	0
Closure	503	246	0	0	9	0

Table 2: Mutant killing comparison between Zest and Mu2 (time-controlled). The "All", "Any", and "No" modifiers refer to all repetitions of each experiment.

all repetitions and using either technique. The last four columns displays 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).

From Table 1, we can see that the test-input corpus produced by Mu2 features consistent improvement in mutation score for 4 of the 5 benchmarks. In particular, for Gson and Jackson, Mu2's test corpus *always* kills a number of mutants (1 and 25 respectively) that consistently survive using Zest's test corpus. This is a promising result in favor of mutation-analysis guidance. We noticed that for Tomcat, the killed mutants saturated at 252 in almost all of the repetitions of both Zest and Mu2, leading to no unique mutants being killed by either technique. In only the Closure benchmark, the Zest corpus is able to kill a mutant that survives in all output corpora of Mu2. However, the Mu2 corpus can kill 3 unique mutants in Closure that consistently survive when using Zest.

The difference in branch coverage achieved by these corpora is minimal. Figure 7 displays the number of branches covered (normalized to the baseline coverage achieved by Zest's corpus). Mu2 increases branch coverage for ChocoPy and Jackson (significant as per Student's t-test; $p < 0.05$), but this is only a 1–4% improvement.

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.*

5.2 RQ2

> *Does mutation-analysis guidance produce a higher quality test-input corpus over coverage-only feedback in greybox fuzzing* ***when controlling for time?***

RQ2 focuses on the evaluation of mutation-analysis-guided fuzzing while fixing the time budget. This draws a fairer comparison between Zest and Mu2, since the performance overhead of mutation testing is factored into the results. A higher mutation score of 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.

Looking at Table 2 (which has the same structure as Table 1), we see that Mu2 is able to produce a corpus that kills more mutants in ChocoPy, Gson, and Jackson when alotted the same time budget as Zest. In fact, many of the mutants uniquely killed from the trial-controlled Mu2 experiments are also uniquely killed in the time-controlled Mu2 experiments. This shows promise that mutation-analysis guidance can be utilized in a practical setting.

However, we can observe that the Zest corpus is able to kill 9 mutants that consistently survive using the Mu2 corpus for the Closure benchmark. This is due to the performance overhead of running mutation analysis during the fuzzing loop, since 8 of these mutants *are* killed in the trial-controlled Mu2 experiments.

Additionally, Figure 7 shows consistent decrease in branch coverage in the larger Tomcat and Closure benchmarks. The Mu2 corpus achieves around 20% less branch coverage than Zest on Closure.

We thus conclude that *on small target programs, mutation-analysis guidance helps in producing a test-input corpus with higher mutation score than that produced by coverage-guided fuzzing even with a fixed time budget. However, more work is needed in scaling up mutation analysis in the fuzzing loop to larger target programs.* We hope that our proposed improvements in Section 4.2 serve as a basis for future techniques to build on.

5.3 RQ3

> *Is mutation-analysis guidance able to more effectively generate mutant-killing inputs during the fuzzing campaign than coverage-only guidance?*

For Mu2, the number of mutants killed by any fuzzer-generated input is the same as the number of mutants killed in its final test-input corpus, since we save all inputs that improve the mutation score. However, the same is not true for Zest. In order to get this data for Zest, we create the baseline Zest*, which calculates the mutation score of *all* inputs generated by Zest in the fixed time budget of 3 hours. If the trial-controlled Mu2 experiments are able to achieve higher mutation scores than Zest*, then the mutation-analysis guidance is strong enough to *discover* mutant-killing inputs that coverage-only guidance is incapable of.

A naive implementation of Zest* would require running every Zest-generated input on all MUTANTS^1P^0, which is very expensive. In practice, we simulate Zest* by implementing a modification of Algorithm 2. Specifically, we keep line 9, which can determine if a Zest-generated input kills a new mutant, but we remove line 10, so that such inputs are not saved to the final corpus. We run Zest*

	Mutants Executed	Number of Mutants Killed By				
		Any	No Zest* &		No Mu2 &	
			Any Mu2	All Mu2	Any Zest*	All Zest*
ChocoPy	406	302	0	0	0	0
Gson	412	309	0	0	0	0
Jackson	602	449	0	0	0	0
Tomcat	406	302	0	0	0	0
Closure	503	249	0	0	1	0

Table 3: Mutant killing comparison between trial-controlled Mu2 and Zest*. The "All", "Any", and "No" modifiers refer to all repetitions of each experiment.

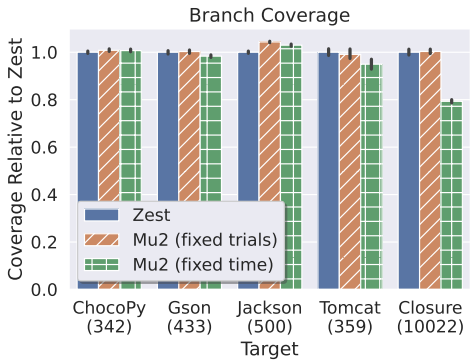

Figure 7: Branch coverage across all benchmarks normalized by the coverage achieved by Zest. The number of branches covered by Zest (used to normalize) is listed below each target.

for the same number of trials as the 3-hour Zest fuzzing campaign, using the same pseudorandom seeds. Effectively, we are able to compute the number of mutants killed by all Zest-generated inputs. The Zest* simulation can still take much longer than 3 hours per experiment; for Closure, it can take 7.5 CPU-days for a single run.

The results in Table 3 compare the mutation scores achieved by trial-controlled Mu2 with the trial-controlled runs of Zest*. We can observe that there is almost no difference in mutation score between the two techniques. There is the one mutant in Closure killed by Zest* that survives in Mu2, which is the same mutant as in Table 1. We found this to be surprising, since it indicates that Zest is actually able to generate inputs that kill all the mutants that Mu2 can kill in the same number of trials, but Zest does not have the information to save these high-quality inputs to its corpus.

We can draw the conclusion that *mutation-analysis guidance does not generate more mutant-killing inputs in the fuzzing loop than those generated using coverage-guiding fuzzing.*

Practically, the conclusions to the three RQs indicate that mutation analysis guidance is effective at generating a high-quality test-input corpus because it can identify inputs that improve mutation score as soon as they are generated—on small target programs, even despite the performance overhead—but that Zest's algorithm is sufficient for generating high-quality inputs in the fuzzing loop, as is more relevant for bug-finding campaigns.

5.4 Case Study on Closure Mutant

In this section we show a sample case of an input saved by Mu2 killing a mutant that survives in all output corpora of Zest. We also provide an intuition for why that may be the case.

Figure 8 shows a simplified `funcCallArgMap` function from the Closure compiler source. For each call site of a JavaScript program, `funcCallArgMap` takes as input the names of the parameters declared by the callee function and the arguments provided by the caller. The function then creates a map of the names of the parameter and argument nodes of the call site. Note that JavaScript allows passing in an arbitrary number of arguments to a function. In order to support this, Closure uses the corresponding parameter name if the index of the argument is within the number of parameters (Line 7); otherwise a new unique variable name is created for each extra argument (Line 9). The `newId` function returns a unique ID every time the function is called. Line 18 shows the original implementation of `newId` method which returns and increases the static `id`. Line 19 illustrates a mutant generated that mutates the return value of `newId` to the constant 0.

```
1 (f => f)(1, 2)
```

The above line of code shows a simple JavaScript program that provides full branch coverage of the `funcCallArgMap` function. Specifically, the JavaScript program defines an identity lambda function and calls the function with the arguments `(1, 2)`. While processing this expression, the first argument `1` is mapped to parameter `f` and the second argument `2` is mapped to a new unique variable `var_0`. Unfortunately, this input cannot be used to kill the mutant shown in Figure 8 because there is only one extra argument. This example program was similar to one saved in both the Zest and Mu2 corpus.

```
1 (f => f)(1, 2, 3)
```

We compare the previous to the above line of code, which shows a simple JavaScript program saved by Mu2 that kills the mutant shown in Figure 8. This JavaScript program also declares an identity lambda function, but invokes it with *three* arguments instead of two. While processing the function call, the mutant assigns `var_0` to both the second and the third arguments, failing the assertion in Line 11. Zest does *not* save this mutant since it achieves no new code coverage due to the existing coverage of the previous input. However, Mu2 does save this input due to its mutant killing ability.

6 THREATS TO VALIDITY

Threats to construct validity. During the fuzzing loop, our test oracles (ref. Section 4.1) 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.

```
Map funcCallArgMap(String[] params, Node[] args) {
  Map argMap = new HashMap();
  for (int i = 0; i < args.size; i++) {
    Node arg = args[i];
    String name;
    if (i < params.size) {
      name = params[i];
    } else {
      name = "var_" + newId();
    }
    assert(!argMap.contains(name));
    argMap.put(name, arg);
  }
  return argMap;
}
static int id = 0;
String newId() {
  return id++; // Original code
  return 0; // Mutant
}
```

Figure 8: The Closure compiler that tries to create a function call argument map. This function creates unique variable names for extra arguments that the function does not declare.

Threats to internal validity. First, all our experiments used the baseline duration of 3 hours to remain consistent with other research that evaluates JQF and Zest on similar benchmarks [13, 18, 19, 35]. Researchers have argued for shorter [36] or longer [16] durations, but this choice is mostly arbitrary. Choosing a different duration could potentially affect our conclusions, though it is unlikely to be significant given that coverage improvements usually saturate after about one hour on most benchmarks. Second, 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.

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 [39] is outside the scope of this paper.

7 RELATED WORK

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 [48]. The majority of fuzzing research focuses on improving heuristics such as seed-picking power schedules [1], input mutations [11, 12, 17], and coverage feedback [10, 14]. FuzzFactory [49] 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. However, to the best of our knowledge, mutation testing has not been used to guide greybox fuzzing.

Improving the performance of mutation testing. A lot of research has been conducted to scale up mutation testing [32, 34]. 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. Madeyski et al. [50] survey a number of techniques for statically avoiding *equivalent mutants*: A mutant P^0 of program P is said to be *equivalent* to the original program P if it is functionally equivalent; in such a case, no test input x can kill such a mutant—$\forall x : : \textsc{kills}^1 P^0, x^o$. Mutation operators can also be designed to avoid *redundancy*, where a pair of mutants P^0 and P^{00} are equivalent to each other such that $\forall x : \textsc{kills}^1 P^0, x^o \;\;,\;\; \textsc{kills}^1 P^{00}, x^o$. Just et al. [23] introduce the *propagation*, *infection*, *execution* (PIE) model to prune mutants using dynamic analysis. Mu2 implements only the *execution* optimization at this time. Weak mutation [51] has been proposed to terminate mutant evaluation quickly by observing the intermediate state after executing the mutated program locations.

Using mutation testing in automated test generation. μ-test [52] and EvoSuite [53] 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.

8 CONCLUSION AND FUTURE WORK

We presented the first technique to incorporate mutation testing into guiding greybox fuzzing. Our implementation Mu2 integrates PIT mutation testing into the JQF framework, and is evaluated against the baseline coverage-guided fuzzing Zest algorithm. Our findings are the following:

(1) Mutation-analysis-guided fuzzing, while ignoring runtime overhead, is able to synthesize a higher quality test-input corpus than coverage-guided fuzzing.
(2) Mutation-analysis-guided fuzzing, under a practical time-controlled setting, is able to produce a higher quality test-input corpus in smaller benchmarks. However, additional work improving its scalability is necessary for improving performance in larger benchmarks.
(3) Coverage-only guidance is able to effectively generate mutant-killing inputs as well as mutation-analysis guidance, but lacks the information to save them to the test-input corpus.

Our results indicate that using mutation analysis as a saving criteria for inputs is a promising approach to improve test-input corpus quality. For our future work, we plan to improve the runtime performance of Mu2 by applying the optimization techniques discussed in Section 7. Implementing the ideas of weak mutation [51], PIE [23], redundant mutations [54], and predictive mutation testing [55] can substantially decrease the number of mutants to run in the fuzzing loop. We hope that our insights will inspire further research in incorporating mutation testing into greybox 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] 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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[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] 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

[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] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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

[43] 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

[44] 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.

[45] 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.

[46] 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

[47] M. R. Hoffmann, B. Janiczak, and E. Mandrikov, "Eclemma-jacoco java code coverage library," 2011.

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

[49] 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

[50] 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.

[51] 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.

[52] 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

[53] 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.

[54] R. Just and F. Schweiggert, "Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 490–507, 2015.

[55] 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.