Investigating Coverage Guided Fuzzing with Mutation Testing

Ruixiang Qian*
qrx@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University, China

Quanjun Zhang*
quanjun.zhang@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University, China

Chunrong Fang†
fangchunrong@nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University, China

Lihua Guo
garyglh@163.com
State Key Laboratory for Novel Software Technology
Nanjing University, China

ABSTRACT

Coverage guided fuzzing (CGF) is an effective testing technique which has detected hundreds of thousands of bugs from various software applications. It focuses on maximizing code coverage to reveal more bugs during *fuzzing*. However, a higher coverage does not necessarily imply a better fault detection capability. Triggering a bug involves not only exercising the specific program path but also reaching interesting program states in that path.

In this paper, we use mutation testing to improve CGF in detecting bugs. We use mutation scores as additional feedback to guide fuzzing towards detecting bugs rather than just covering code. To evaluate our approach, we conduct a well-designed experiment on 5 benchmarks. We choose the state-of-the-art fuzzing technique Zest as baseline and construct two modified techniques on it using our approach. The experimental results show that our approach can improve CGF in both code coverage and bug detection.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → **Program analysis**.

KEYWORDS

Fuzzing, Coverage Guided Fuzzing, Mutation testing

---

*Both authors contributed equallny to this research.
†Corresponding author.

---

1 INTRODUCTION

Fuzzing is one of the most popular techniques to test software correctness and reliability [8, 20]. At high level, fuzzing is a process that repeatedly runs the program under test (PUT) with a mess of generated inputs, some of which maybe syntactically or semantically invalid. It relies on a component named *fuzzer* to generate test inputs and execute the PUT. Generally, a fuzzer generates inputs from given seeds and exercises the PUT continuously with the aim of exposing errors of PUTs in a period of time. To date, fuzzing is almost the most widely-adopted technique due to its conceptual simplicity, low barrier for deployment, and efficacy in discovering real-world bugs [20].

However, fuzzing can be rather ineffective at exploring different program paths. This is because a fuzzer, or more specifically a *black-box fuzzer*, exercises a PUT in a totally blind manner. White-box fuzzers adopt systematic effort[8, 12] to aid fuzzing in searching diverse program paths. However, white-box fuzzers can slow down the execution of the PUT due to the heavy program analysis it performs [20]. To make balance, *coverage guided fuzzing* (CGF) [2, 20] uses lightweight instrumentation to gain coverage information from executions and guide fuzzing towards rarely executed paths with attained coverage. Owing to its strengths, many CGF techniques have emerged in recent years[2, 4, 16, 23, 24, 33]. For example, Zest [24] focuses on programs with syntax checks and has revealed semantic bugs in 5 widely used third-party applications; AFL [33] has been reported to detect tens of thousands of vulnerabilities in hundreds of real-world software projects [4].

CGF endeavours to reveal bugs within the PUT via maximizing code coverage [25]. However, a higher coverage does not necessarily imply a better bug detection capability. Many researches have revealed that the correlation between code coverage and bug detection capability is weak [10, 15, 18]. Concentrating only on code coverage provides inadequate feedback for fuzzing, which may potentially lower its effectiveness in detecting bugs.

In this paper, we incorporate CGF with mutation testing to address the aforementioned challenges. *Mutation testing* is a fault-based testing technique which realises the idea of using *mutants* (buggy versions mutated from the PUT) to support testing activities [25]. We exploit mutation testing to identify bug-revealing inputs generated by fuzzers, and use mutation scores as additional feedback to guide fuzzing towards detecting bugs. Our approach comprises two stages: (1) in initialization stage, we create mutants and make preparations for fuzzing, and (2) in fuzzing stage, we

generate inputs and execute them against PUT and mutants. We check fault detection capability for each generated test input and preserve the bug-revealing ones, i.e., the inputs that can kill any mutants.

We conduct a well-designed experiment with 15 fuzz campaigns involving 5 benchmarks to evaluate the proposed approach. We choose the state-of-the-art CGF technique Zest [24] as baseline and construct two modified techniques on it with our approach in both *negative* (generating fewer children from inputs kill mutants which have been killed in previous) and *positive* (generating more children from inputs which are capable of killing any mutants) manners. The experimental results show that mutation testing can help CGF (1) to maximize code coverage faster in 3 out of 5 benchmarks with a negative manner, and (2) to reveal bugs faster in all 5 benchmarks. Besides, the techniques modified with our approach detects 10 more bugs in one of the benchmarks. We summarize the main contributions of this paper are as follows:

- **Novel Approach.** We propose an approach which uses mutation scores as feedback to guide fuzzing towards detecting bugs. To our best knowledge, this is the *first* work which incorporates CGF with mutation testing.
- **Practical Framework.** We implement the proposed approach as a framework which comprises a *mutation engine*, a *testing engine* and a *fuzzing engine*. The components of our framework are scalable and can be generalized to other CGF techniques.
- **Extensive Study.** We conduct an experiment with 15 fuzz campaigns involving 5 benchmarks. The experimental results demonstrate that our approach outperforms CGF in both code coverage and bug detection.

2 BACKGROUND

2.1 Coverage Guided Fuzzing

Algorithm 1 presents the typical process of CGF. It takes a instrumented program p and a set of initial seeds I as inputs, and return saved seeds Q as well as seeds lead to crashes F as outputs. At first, a coverage guided fuzzer add all inputs given in I to Q and initialize F and global coverage Cov as empty. Next, it comes into the main fuzzing loop (line 4 ~ line 18). For each input i in Q, the fuzzer computes the number of mutations n it going to perform on current input (line 8). The computation heuristic is abstracted as a function mutationChance, whose process is shown as Algorithm 2. Next, the fuzzer mutates current input i for n times to get children inputs i_c. For each i_c, the fuzzer executes p with it once to check the attained coverage cov and execution result res. If the res is failure, then i_c is a failing input and will be added into F. This suggests that i_c has crashed the execution and triggered some vulnerabilities of p; If the res is success, the fuzzer further checks whether cov contains some paths that didn't covered before. If new coverage attained, then i_c will be added into Q for subsequent fuzzing, and the total coverage Cov will be updated.

In Algorithm 2, both $BASE$ and $FACTOR$ are positive constants. To begin with, the number of mutations for each input i is set to the unified baseline constant $BASE$. Next, to show the favor for inputs that can supply new coverage, another constant $FACTOR$ is used to scale up the numbers of mutations for these inputs through multiplication (line 2 ~ 4). The intuition behind these manipulations

---

Algorithm 1: Coverage Guided Fuzzing

Input: program p, initial inputs I
Output: seed inputs queue Q, failing inputs set F

1   $Q \leftarrow I$;
2   $F \leftarrow \emptyset$;
3   $Cov \leftarrow \emptyset$;
4   **repeat**
5   **foreach** *input i in Q* **do**
6       $n \leftarrow$ mutationChance(i, Q) ;
7   **for** $0 < i < n$ **do**
8         $i_c \leftarrow$ mutate(i, Q) ;
9         $cov, res \leftarrow$ execute(p, i_c);
10   **if** isCrash(res) **then**
11          $F \leftarrow F \cup \{i_c\}$;
12   **else if** existNewCov(cov) **then**
13          $Q \leftarrow Q \cup \{i_c\}$;
14          $Cov \leftarrow Cov \cup \{cov\}$;
15   **end**
16   **end**
17   **end**
18   **until** *exceeding given resources*;
19   **return** Q, F ;

Algorithm 2: Computation of Mutation Chance

Input: seed input i, seed inputs queue Q
Output: number of mutations n

1   $n \leftarrow BASE$;
2   **if** canProduceNewCov(i, Q) **then**
3     $n \leftarrow n \times FACTOR$;
4   **end**
5   **return** n

---

is to keep the quality of generated inputs with the help of "Matthew Effect" [4]. However, traditional CGF use same heuristic to compute n for every inputs. It does not distinguish inputs that are capable of detecting bugs from normal ones, which consequently hinder fuzzing from finding more bug-revealing inputs. We will elaborate the challenges and show our solution at Section 2.3.

2.2 Mutation Testing

Mutation testing is a fault-based testing technique [6, 25]. It uses artificial bugs, called mutants, to evaluate the adequacy of testing activities. Given a PUT and a set of test inputs, mutation testing firstly generates a set of mutants with a mutation engine, and then executes these test inputs against these mutants to compute mutation score.

A mutation engine generate mutants in three steps: Firstly, it selects a set of mutators (syntactic rules which encodes the transformation of the syntax of program) to create mutants. Secondly, it creates a group of mutants according to used mutators. Thirdly, it optimizes mutants through removing redundant mutants. Note that mutant creation may result in mutants that are equivalent to or subsumed by other mutants [25]. Besides, some mutants may

semantically equivalent to the PUT, even though they are syntactically different. These mutants are detrimental to the result of mutation result such that should be removed before execution [25].

The generated mutants are then executed to evaluate the adequacy of test inputs. The adequacy of test inputs is measured by mutation score, which can be computed as follows:

$$score = \frac{mut_k}{mut_s + mut_k} \times 100\%$$

Specifically, mutation score is the ratio of killed mutants (mut_k) to all mutants (the sum of killed and survived mutants). If the execution of a mutant with the given test inputs *fails*, it means the defect represented by this mutant is detected such that the mutant is *killed*; Otherwise the mutant is *survived*, implying that the given inputs are incapable of detecting such a defect. Generally, mutation score can be used to reflect the capability of given inputs in detecting bugs. Suppose only one input was sent to mutation testing, then a non-zero mutation score implies that the input is capable of detecting bugs.

2.3 Motivation

```
1  int foo(int x, int y) {
2      if (x > y)
3          return x;
4      else
5          return x; // Should return y.
6  }
```

Listing 1: A simple method that is supposed to return the larger value of x and y. A bug lies on line 5 where the y is larger one and should be returned.

Motivating example. Consider the code snippet presented at List 1. The method foo takes two integers x and y as inputs and is supposed to return the larger value among them as output. However, there is a bug lies on line 5 in that else clause. It is y rather than x that should be returned as y is no less than x.

This code snippet illustrates the situation that a bug is reached but may not be triggered. Consider we have two test inputs $i_1 = \langle 1, 1 \rangle$ and $i_2 = \langle 1, 2 \rangle$. Both of i_1 and i_2 cover line 5 but only i_2 can trigger the bug by making it observable at output.

Drawbacks of traditional CGF. Suppose test input i_1 and i_2 are generated in sequence during a fuzz campaign. According to Algorithm 1, traditional CGF will not preserve i_2 as it supplies same coverage as i_1. As a result, the bug-revealing input i_2 is discarded, and the fault lies at line 5 may hidden.

```
1  int foo(int x, int y) {
2      if (x > y)
3          return x;
4      else
5          return 0; // Being mutated.
6  }
```

Listing 2: A mutant of code snippet illustrated at List 1 where a "ReturnZero" mutator is conducted at line 5

Our solution. We leverage mutation testing to address the drawbacks of traditional CGF. We firstly create a mutant for PUT, which is shown in List 2. Specifically, this mutant is created by conduct

a "ReturnZero" mutator at line 5 of the method foo. After that we perform mutation testing with the created mutant. In addition to the original PUT (List 1), we execute the created mutant with the same inputs (i_1 and i_2) generated during fuzzing. We check the consistency between the outputs of original PUT and the created mutant to compute mutation score. If outputs are consistent, then the mutant is survived and the mutation score is "0"; Otherwise the mutant is killed and the mutation score is "1". We prefer bug-revealing inputs such that preserve the inputs which are capable of killing the mutant (getting a mutation score of "1"). With our approach, the bug-revealing input i_2 will not be discarded as before, and the bug lies at Lsine 5 can be potentially detected.

3 APPROACH

3.1 Overview

Algorithm 3: Fault Detection Aware CGF

Input: program p, initial inputs I, mutation configuration c
Output: seed inputs queue Q, failing inputs set F

1   $Q \leftarrow I$;
2   $F \leftarrow \emptyset$;
3   $Cov \leftarrow \emptyset$;
4   $M \leftarrow$ buildMutantPool(p, c) ;
5   **repeat**
6   **foreach** *input i in Q* **do**
7     $n \leftarrow$ mutationChance(i, Q) ;
8   **for** $0 < i < n$ **do**
9      $i_c \leftarrow$ mutate(i, Q) ;
10      $M' \leftarrow$ selectMutants(M, c) ;
11      $cov, res, stat \leftarrow$ execute(p, M', i_c);
12   **if** canKillMutants($stat$) **then**
13   markAsCapable(i_c) ;
14       $Q \leftarrow Q \cup \{i_c\}$;
15   **if** canKillNewMutants($stat, M$) **then**
16   markKillNew($i_c, stat$);
17   update($M, stat$);
18   **end**
19   **end**
20   **if** isCrash(res) **then**
21       $F \leftarrow F \cup \{i_c\}$;
22   **else if** existNewCov(cov) **then**
23       $Q \leftarrow Q \cup \{i_c\}$;
24       $Cov \leftarrow Cov \cup \{cov\}$;
25   **end**
26   **end**
27   **end**
28 **until** *exceeding given resources*;
29 **return** Q, F ;

Figure 1 depicts the overview of our approach. Our approach takes a PUT and system configurations as inputs and outputs preserved seeds (i.e., test inputs). System configurations not only contain settings for mutation testing but also settings for fuzzing such
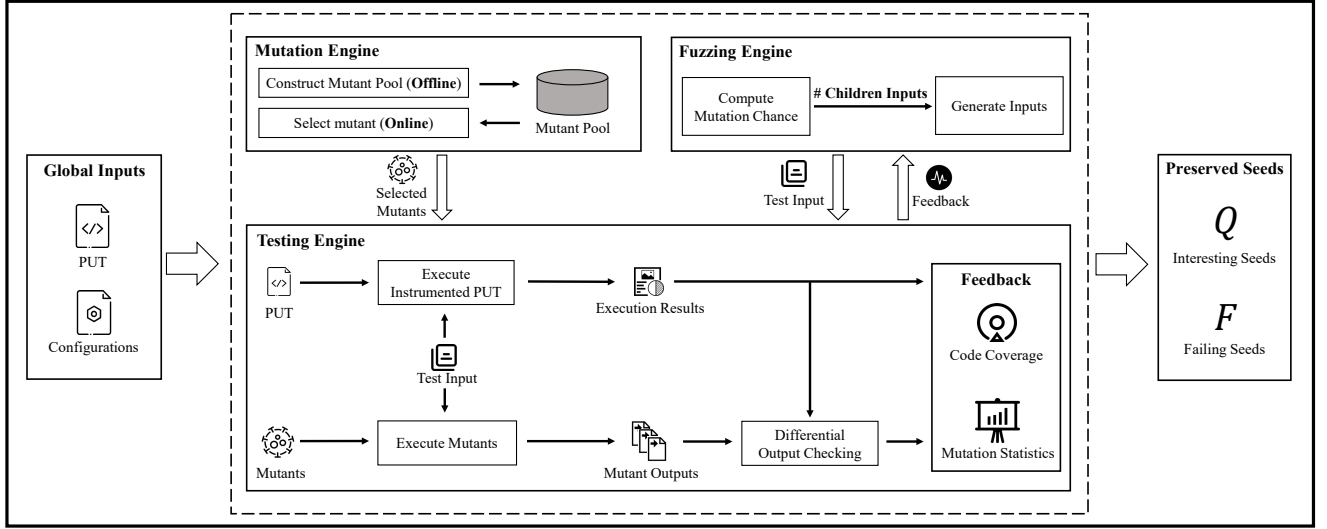
Figure 1: Overview of our approach.

Algorithm 4: Fault Detection Aware Mutation Chance

Input: seed input i, seed inputs queue Q
Output: number of mutations n

1 $n \leftarrow BASE$;
2 **if** canProduceNewCov(i, Q) **then**
3 $\quad n \leftarrow n \times FACTOR$;
4 **end**
5 **if** isBugRevealing(i) **then**
6 $\quad n \leftarrow n \times KILL\_FACTOR$;
7 **else**
8 $\quad n \leftarrow n \div KILL\_FACTOR$;
9 **end**
10 **return** n

---

as allocated resources and initial seeds. We elaborate modified CGF algorithms (Algorithm 3 and Algorithm 4) in Section 3.2. Our approach comprises three main components: (1) a *mutation engine* ⓜ for creating and selecting mutants (Section 3.3), (2) a *testing engine* ⓣ for executing programs and computing feedback (Section 3.4), and (3) a *fuzzing engine* ⓕ for computing mutation chances and generating inputs (Section 3.5).

3.2 Fault Detection Aware CGF

We modified the general CGF workflow to enable (1) the identification of bug-revealing inputs and (2) the preservation of fault detection capability. Algorithm 3 shows the *fault detetion aware* CGF (μCGF in short) modified from Algorithm 1. The additional parts are highlighted in blue. In spite of the PUT p and initial inputs I, μCGF requires a configuration c for mutation testing. It comprises a set of mutators for mutant creation before fuzzing and a strategy for mutation selection (line 10) during fuzzing. Before coming into main fuzz loop, we additionally create a *mutant pool* M for p according to c as preparation. During fuzzing, we perform mutation

testing with each generated input i_c by executing it against selected mutants M' to get statistics $stat$. Note that we only select partial mutants to ensure the efficiency of fuzzing. By $stat$ we record detailed information to indicate whether a specific mutant was killed or survived in the last execution. We identify whether a generated i_c is bug-revealing via checking $stat$ at line 12. Specifically, an i_c is capable of detecting bugs if it has gotten a non-zero mutation score in the last execution. For the i_c capable of detecting bugs, we mark it as "capable" and preserve it into seed queue Q. After that, we further check whether i_c can kill new mutants (which has not been killed by any previous inputs yet) at line 15. For i_c killed new mutants, we mark it in $stat$ and update the statuses for each mutants in M at line 17.

With Algorithm 3 the bug-revealing inputs are marked and now identifiable. On this basis, we further propose Algorithm 4 to preserve fault detection capability of these bug-revealing inputs. We realize this through rewarding bug-revealing inputs by amplifying their chances of being mutated. The inputs which are not bug-revealing will be penalized in an opposite way. Note that the "mutation" here indicates the process of generating children inputs from parent seeds. We set a positive constant $KILL\_FACTOR$ to adjust the computation of mutation chances. At first the number of mutations n is assigned by a fixed constant $BASE$. If an inputs it bug-revealing, we multiply n with $KILL\_FACTOR$ to amplify its influences in the subsequent input generation. Otherwise, we avoid generating children inputs from it by dividing its n with $KILL\_FACTOR$. Note that it is possible that an actually bug-revealing input is marked as "incapable" as we only select a part of mutants from M during fuzzing due to efficiency concern. The mutants the input can kill may not be selected thus the input will be not be treated as bug-revealing. The trade of missing some bug-revealing inputs for efficiency is acceptable as the main goal of Algorithm 3 is to preserve fault detection capability. With Matthew effect [4], only a small faction of bug-revealing inputs preserved can improve the possibility of generating bug-revealing children in numerous cycles of fuzzing.

3.3 Mutant Creation and Selection

Ⓜ provides two modes: an *offline mode* for mutant creation and a *online mode* for on-the-fly mutation selection. The *offline mode* is used for preparing fault detection aware fuzzing. Mutants are created according to mutators specified in mutation configuration c (Algorithm 3). We preserve a mutated version of the PUT together with corresponding mutation information (such as mutated location and used mutator) for each mutant, and construct a mutant pool for subsequent usage. The *offline mode* is performed only once to avoid overheads caused by repeated creations. On the contrary, the *online mode* continuously works during test executions. Ⓜ periodically selects a subset of mutants from the pool according to a selection strategy specified in c. The selected mutants are sent to Ⓣ to for mutation testing. Generally, by selection strategy, we only pick a small set (typically 10) of mutants to ensure the efficiency of fuzzing as the number of total mutants can be very large [6, 25], especially when there are too many locations for mutation.

3.4 Test Execution

Ⓣ is the core of our approach. It extends the test execution component of traditional coverage guided fuzzers. Ⓣ receives test inputs from Ⓕ and sends feedback information, i.e. coverage and mutation statistics, back to Ⓕ to aid input generation. Besides executing the instrumented PUT with the received test inputs, Ⓣ also needs to execute the same inputs against the mutants selected by Ⓜ.

3.4.1 Terms. We formalize the terms used in this section to avoid ambiguous understanding.

DEFINITION 1 (*Program*). A program is an object of test. In this paper, a program can be a PUT p or a mutant mut.

DEFINITION 2 (*Program Behavior*). A program behavior represents a functionality of a PUT, which can be denoted as b. A PUT involves a set of behaviors $\mathbb{B} = \{b_1, b_2, ..., b_n\}$, where $n \geq 1$.

DEFINITION 3 (*Output*). An output o is the result of exercising a program behavior b belongs to a program p with a given test input i, which can be denoted as $o = b[p, i]$.

DEFINITION 4 (*Execution Result*). An execution result $exec$ is a triple $\langle s, \mathbb{O}, e \rangle$, where s is the status of execution which satisfies $s \in \{\text{SUCCESS}, \text{FAILURE}\}$, $\mathbb{O}$ is a set of outputs which can be denoted as $\mathbb{O} = \{o_1, o_2, ..., o_n\}, (n \leq 1)$, and e is the crash occurred during execution. Note that one of $\mathbb{O}$ and e is null as when s is SUCCESS, the execution was successful such that no crash will be raised; whereas in turn, when s is FAILURE, the execution was crashed and no outputs will be produced.

DEFINITION 5 (*Mutant Status*). A mutant status represents whether a artificial bug represented by a mutant mut was detected, which can be denoted as $\epsilon[mut]$. A mutant status ϵ satisfies $\epsilon[mut] \in \{\text{KILLED}, \text{SURVIVED}\}$. We further discuss the judging of mutant statuses in Section 3.4.3.

3.4.2 Running Mutants and PUT. Mutation testing can be very time consuming [6, 25] as it has to run a single test input against every selected mutants once. Note that we do not distinguish *mutation analysis* and *mutation testing* [25] as there is only one test input to run at each execution. We have reduced the number of mutants to execute by strategically selecting a subset of mutants (Section 3.3).

To further improve execution efficiency, we concurrently execute the selected mutants together with the PUT in multi-threads. We collect execution results for both PUT and the selected mutants. Herein, we call execution results for PUT and mutation testing as "test result" and "mutation test result" respectively. A test execution is FAILURE if it was crashed (e.g. throws an exception) during execution, or else it is SUCCESS (Definition 4). We allow mutants to run slightly longer than the PUT does, and provide an *extra execution time ratio* to control the extra time. We make this design for two reasons: (1) To get more confidential mutation statistics. Test executions can accidentally fluctuate due to reasons like system scheduling. Besides, since mutants are created using various mutators (each mutator is a kind of syntactic transformation), some mutants exactly need more time to execute than the others. A longer time can enable more mutants to finish executions without timeout, which consequently reduces false negatives; (2) To keep fuzzing efficient. Syntactic transformations on the PUT may sometimes result in time-consuming artificial bugs such as endless loop in practice [6]. Setting an upper bound time for executing mutants can prevent executions from being blocked by such unexpected issues. For a PUT whose execution time is t, we use extra ratio $r(0 \leq r \leq 1)$ to control the upper bound time t' for executing mutants, which is computed as $t' = t \times (1 + r)$. Note that we consider the mutation test results of mutants which does not finished executing in t' as FAILURE during fuzzing.

3.4.3 Analyzing Execution Results. Test execution results should be further analyzed to identify the nature of test inputs. Through such analysis, We further analyze test execution results to identify whether a test input has supplied fresh coverage or killed mutants. The the process of analysis varies according to execution statuses. Specifically, if test result is FAILURE, the test input is failing input should be saved into failing seed queue F (Algorithm 3). Mutation test results will not be analyzed no outputs was produced; Else, if test result is SUCCESS, we further analyze execution results to determine the statuses of mutants. A mutant is killed if it is detected by certain test cases (Section 2.2). Killing mutants requires the buggy outputs of mutants can be observed explicitly. However, as executed test inputs are generated dynamically during fuzzing, the real expected outputs are impossible to know in advance. Therefore, we use output of the PUT as expected output. We build a *mutant checking mechanism* based on the thoughts of *differential testing*. Differential testing [21] is a random testing technique which reveals potential bugs through comparing the outputs (with same inputs) of among comparable software systems. We treat a mutant as a comparable system to the PUT and identify the statuses of it by comparing its outputs $\mathbb{O}'$ to the outputs of the PUT $\mathbb{O}$. We formalize the goal of the mutant checking as follows:

DEFINITION 6 (*Differential Outputs Checking*). Given a test input i, a set of program behaviors $B = \{b_1, b_2, ..., b_m\}$, a PUT p and a set of selected mutants $M' = \{mut_1, mut_2, ..., mut_n\}$, the goal of *mutant checking* is to determine status for each $mut \in M'$ via comparing its outputs $\mathbb{O}'$ to outputs $\mathbb{O}$ of the PUT p. The result of mutant checking are mutation statistics $stat$ described in Algorithm 3.

Specifically, if the execution of a mutant mut has crashed, or some of its outputs are inconsistent to the PUT, then the mutant is

detected and *killed* by the executed test input *i*. Otherwise, *mut* is ignored such that *survived*. We define mutant surviving and killing as follows:

DEFINITION 7 (*Mutant Surviving*). Given a PUT p whose test result is $exec = \langle s, \mathbb{O}, e \rangle$ and a mutant *mut* whose mutation test result is $exec' = \langle s', \mathbb{O}', e' \rangle$, the *mut* is *survived* if the execution status is SUCCESS and **all** its outputs are consistent to the PUT. We denote *mutant surviving* as $s' = \text{SUCCESS} \wedge \mathbb{O} = \mathbb{O}' \Rightarrow \epsilon[mut] = \text{SURVIVED}$.

DEFINITION 8 (*Mutant Killing*). Given a PUT p whose test result is $exec = \langle s, \mathbb{O}, e \rangle$ and a mutant *mut* whose mutation test result is $exec' = \langle s', \mathbb{O}', e' \rangle$, the *mut* is *killed* if the execution status is FAILURE or **existing** at least one output which is inconsistent to that of p. We denote *mutant killing* as $s' = \text{FAILURE} \vee \exists j \to o_j \neq o'_j, o_j \in \mathbb{O}, o'_j \in \mathbb{O}' \Rightarrow \epsilon[mut] = \text{KILLED}$.

3.4.4 Constructing feedback. At the end of each execution, Ⓣ gathers attained coverage *cov* together with fault detection statistics *stat* to construct feedback. These feedback information will be sent to Ⓕ to guide input generation.

3.5 Input Generation

Ⓕ enables fault detection aware fuzzing with feedback information sent back from Ⓣ. It updates interesting seeds queue Q as well as failing inputs queue F according to Algorithm 3. Inputs capable of killing mutants will be allocated more energy in order to generate more children from them (Algorithm 4). Note that Ⓕ is a general component which can be replaced by any coverage guided fuzzer.

4 EXPERIMENTAL SETUP

In this paper, we evaluate how our approach influences the performance of CGF technique. In particular, we focus on the following research questions:

- **RQ1**: How mutation testing influences CGF in covering code?
- **RQ2**: How mutation testing influences CGF in terms of killing mutants?

Implementation. We implement the proposed approach in Java. Specifically, we implement fuzzing components with JQF [23], which is a widely used fuzzing framework for Java. We implement mutant creation with PIT, which is the most widely used mutation testing engine for Java programs [6]. For test engine Ⓣ, we implement it using test runner supplied by Junit 4 and enable parallel execution with APIs supplied by JDK. For on-the-fly mutant selection, we implement it as an interface `MutantSelectionStrategy`. For differential outputs checking, we implement it as an interface `Criterion`. `MutantSelectionStrategy` can be extended to build subtypes to implement different mutant selection strategy. Similarly, `Criterion` can be extended to meet special needs of checking different kinds of outputs. To conduct experiments, we implement `BasicRandomStrategy` to select fixed number of mutants from mutant pool during fuzzing. For PUT with serializable output, we build a subtype, `SerializableCriterion`, to serialize the outputs of the PUT and its mutants and compare the bytes after serialization.

Benchmarks. The details of the benchmarks are shown at Table 1. All our benchmarks are from Github[9] or previous works. One of our authors manually build 5 fuzz cases for 5 benchmarks. In particular, C01, C02 and C03 are experimental objects used in [29]; C04 and C05 are widely adopted open-sourced subjects. The version of Apache Commons Math3 3.6.1 and Apache Commons Numbers is 1.0. Column **#Mutants** illustrates the total number of mutants we created for each benchmark using PIT default mutators[1], and the last column shows the number of non-blank lines of code.

Table 1: Details of Benchmarks.

CID	Fuzz Case	#Mutants	LoC of Bench.
C01	SortingFuzz	69	88
C02	MatrixInverseFuzz	75	65
C03	SuffixArrayFuzz	215	219
C04	SimpleRegressionFuzz	49783	208891
C05	DivFuzz	577	30396

Techniques. We choose state-of-the-art CGF technique Zest [24] as baseline. Zest is the only fuzzing technique for Java to our best knowledge. We modified Zest with our approach from two aspects, and construct two techniques, namely N-μZest and P-μZest. "N" and "P" indicate how we enhance fuzzing campaigns with mutation testing: for N-μZest we influences fuzzing with mutation testing "negatively", that is, we punish the inputs that kill mutants which have been killed in previous by lessening children inputs generated from them; [3]. for P-μZest we influences fuzzing with mutation testing "positively", that is, we reward the inputs which could kill mutants (no matter whether the mutants have been killed in previous). N-μZest and P-μZest guide CGF in very different ways: with N-μZest we guide fuzzing to explore space far away from the mutants that have been detected in previous, which follows the observation of [3]; with P-μZest, on the contrary, we want to guide fuzzing towards exploring deeper paths.

We instantiate Algorithm 4 differently to implement N-μZest and P-μZest. We choose 20 (the multiplication factor for number of children to produce for favored inputs used by `ZestGuidance`) as the modifier factor used in Algorithm 4. In particular, for N-μZest we (1) divide the number of generated children inputs with 20 when inputs kill old mutants, and (2) multiply the number of generated children kill with 20 when inputs detect fresh mutants; for P-μZest, we just multiply the number of generated children inputs with 20 when inputs kill any mutants.

Experiments. We combine fuzz cases with techniques to construct 15 fuzz campaigns (each campaign is a pair $\{case, tech\}$). We run each campaign for 3h following the setup of [24] to obtain seed corpus. To remove the impact of randomness, we repeat each campaign for 10 times. We use `BasicRandomStategy` to select 10 mutants during each campaign and use 11 threads (10 for mutants and 1 for original PUT) to run these mutants along with original PUT concurrently. We set *extra execution time ratio* as 0.1, which means the time upper bound of executing a mutant is 10% more than executing the original PUT. After finishing all fuzzing campaigns, we run scripts to reproduce each preserved inputs to compute average branch coverage and mutant killing rate obtained in 10 runs. All

---

[1]https://pitest.org/quickstart/mutators/

experiments were run on a cloud machine with 32GB RAM and 16-core Intel Core Processor CPU.

5 EVALUATION

5.1 RQ1: Effectiveness in Code Coverage

To illustrate the performances of different techniques in covering code, we reproduce the seed inputs created during fuzzing campaigns and attach the most recent time points to them. For example, suppose that a certain fuzz campaign generates 4 input seeds id_0000~id_0003 during 0~1s, then we attach the average coverage achieve by id_0000~id_0003 to time point 1s. Figure 2 shows the average branch coverage achieved during different fuzz campaigns. Each row of sub-figures represents the changing tendencies in different time durations from **1 minute** (1m) to **3 hours** (3h). We use red, blue and green lines to illustrate coverage curves produced by Zest, P-μZest and N-μZest respectively. The x-axis represents time points whereas y-axis the achieved average branch coverage rates. We obtain the following observations:

- **Compare among durations.** Branch coverage increases drastically at the beginning of each fuzz campaigns, especially in the first minute (the first column of Figure 2).
- **Compare among techniques.** In terms of covering code, N-μZest performs slightly better then Zest, while P-μZest performs slightly worse. In C01, C03, C05, the maximum coverage rates achieved by each techniques are nearly the same and the curves appear to overlap with each other. In C02, although the maximum coverage rates achieved by each technique are nearly the same, the increasing tendency of N-μZest is much more drastic than that of Zest. By contrast, the increasing tendency of P-μZest is much gentler. In C04, although the increasing tendency of Zest is the most drastic compared to N-μZest and P-μZest in the first minute of campaign, it is quickly caught up by P-μZest in the next few minutes. What's more, the maximum coverage rate achieved by Zest is also worse than that of N-μZest.
- **Compare among cases.** The performance in covering codes varies for different fuzz cases, appears as two aspects: (1) **Increasing tendencies.** In C01, C05 and C05, coverage curves overlap with each other and the maximum coverage rates are similar. Code coverage rates in these cases rise rapidly in the first minute to achieve maximum and become flat in the rest of campaigns. On the contrary, coverage curves in C04 and C04 keep rising for more than 30 minutes. (2) **Maximum coverage rates.** In C01~C03, the maximum coverage rates are more than 60%, whereas the maximum coverage rates in C04 and C05 are far less (below 15%), especially in C04, which is no more than 0.10% (0.096%).

Analysis. According to the observations above we can see that mutation testing does be able to influence the performance of Zest in terms of branch coverage. The enhancement from opposite directions (N-μZest and P-μZest) presents different results. Specifically, N-μZest covers more branches compare to Zest in some cases. The negative enhancement with mutation testing directs fuzzing towards covering area far from the mutants that have already been killed, which is consistent to [3]. However, a positive enhancement with mutation testing (P-μZest) can make fuzzing covering fewer

branches (C02 and C04). This is because P-μZest amplifies the number of children inputs generated from inputs that kill old mutants, which then increases the chance to exploring the branches that has already been covered. Moreover, the maximum branch coverage rates depend on the logic of fuzz cases (which is also know as the distribution of test cases [5]). For example, C04 takes a set of double values as inputs to exercise public methods supplies by SimpleRegression. As a result, C04 is not likely to fuzz branches that belong to types that are not invoked by SimpleRegression.

> ***Answer to RQ1:*** Mutation testing can improve the effectiveness of CGF techniques in covering code in a negative manner. The maximum branches achieved are relevant to the properties of benchmarks.

5.2 RQ2: Effectiveness in killing Mutants

Figure 3 shows average killing rates achieved during fuzz campaigns. Like analysis for branch coverage, Figure 3 also illustrates curves in different time durations. The y-axis of the sub-figures represents the average killing rates. From Figure 3 we can also get observations from the following perspectives:

- **Compare among durations.** Like branch coverage, the increases of mutation killing rates also happen at the beginning of each fuzz campaigns. However, killing rates curves go flat at much latter time points compares to branch coverage. In C01 and C05, curves for N-μZest and P-μZest still rise after fuzzing for more than 1 hour.
- **Compare among techniques.** Both the enhanced techniques (N-μZest and P-μZest) are better than the baseline technique Zest, and N-μZest is better than P-μZest. In C01 and C05, the killing rates of the enhanced techniques rise faster than Zest and finally come to a higher killing rates. In C02, Zest is worse than N-μZest but better than P-μZest during the first 10 minutes. Three curves finally merge together after fuzzing for 30 minutes. In C03 and C04, although the killing rate of Zest rises fastest in the first minute, it is immediately caught up by the enhanced techniques in the next few minutes.
- **Compare among cases.** We get observations similar to Figure 2 for killing rates. The increasing tendencies and the maximum kill rates also varies for different fuzz cases. The killing rate curves for C02~C04 rise drastically at the first minute of fuzzing campaigns and rapidly become flat in the next few minutes, whereas for C01 and C05 the curves keep increasing for the while 3 hours. In terms of maximum kill rates, in C02 and C03 all the three techniques can achieve peek values that are larger than 70%. C04 achieves the smallest maximum kill rates around 0.12%.

Analysis. Compares to Zest, the enhanced techniques kill more mutants with the guidance of mutation testing. However, although both N-μZest and P-μZest achieve higher kill rates than Zest, the type of the mutants which are additionally killed should be different. Moreover, the object for fuzzing as well as the logic of fuzz cases are also important factors for killing mutants.

> ***Answer to RQ2:*** Mutation testing can improve the effectiveness of CGF techniques in killing mutants with both negative and positive manners. The maximum numbers of mutants killed are relevant to the properties of benchmarks.
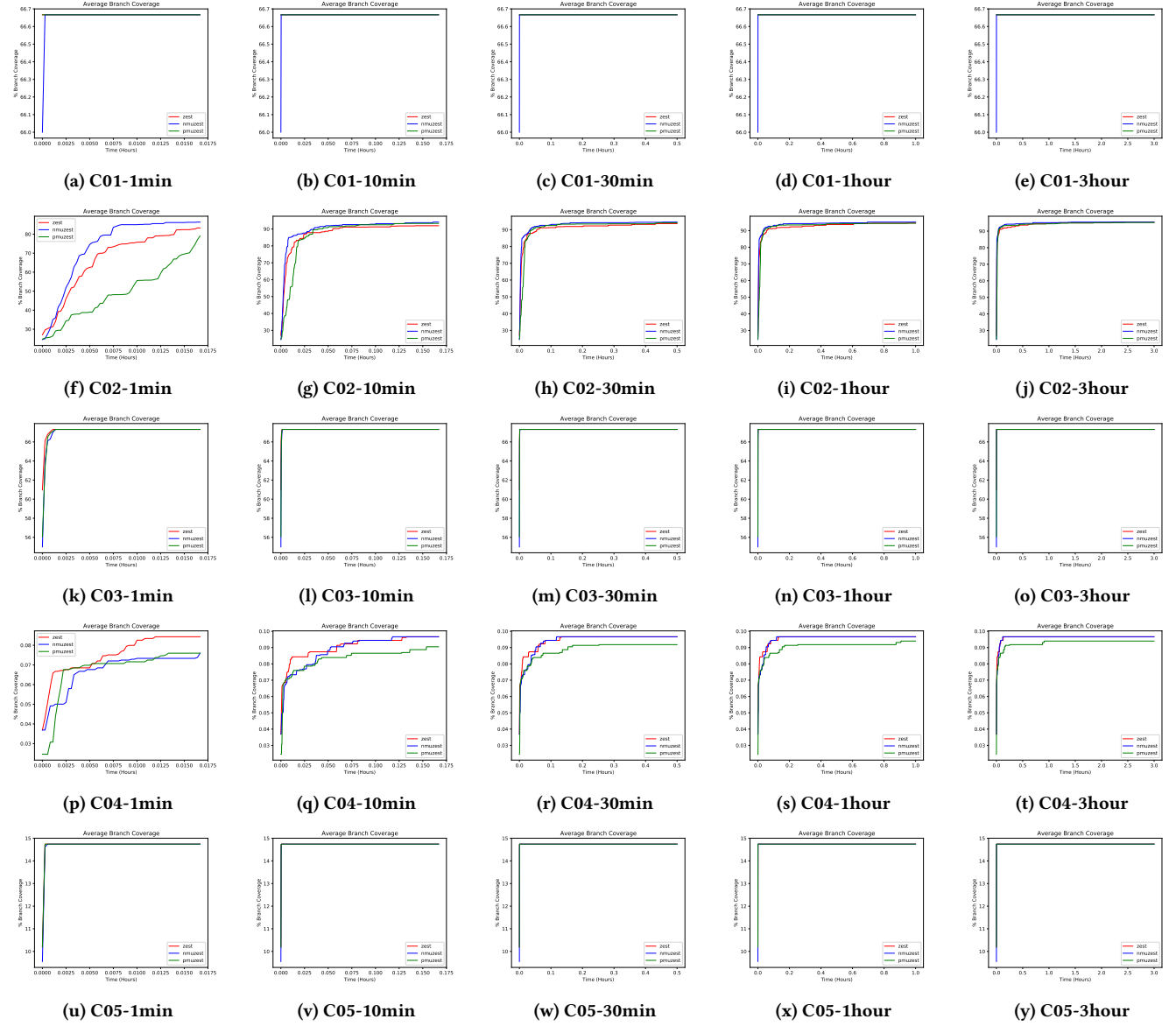
Figure 2: Average branch coverage rates for different fuzz campaigns in different time durations. The x-axis of each figure represents time points and the y-axis represents the achieved branch coverage rates.
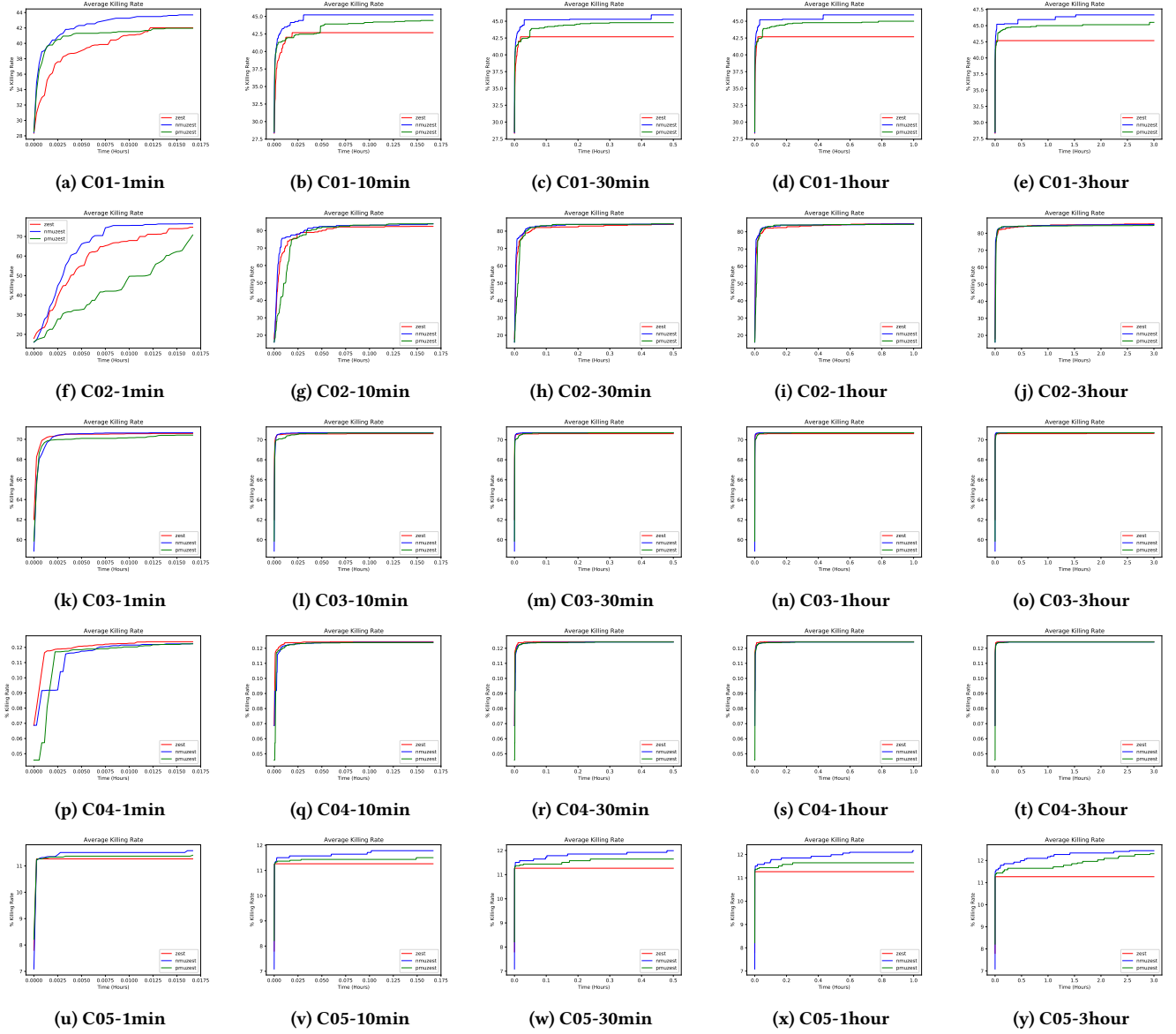
6 RELATED WORK

In this paper, we enhances CGF via identifying bug-revealing inputs and amplifying their effect by mutation testing. Our work is related to *coverage guided fuzzing*, *mutation testing* and *differential testing*.

Coverage Guided Fuzzing. Fuzzing has been a hot research area in the past few decades [20]. It was introduced in the early 1990s [22]. At first, fuzzing was dedicated for testing a PUT fully randomly with given inputs, which called seeds. To make fuzzing more systematic, CGF guides fuzzing with code coverage attained with lightweight instrumentation. AFL [33] employs a novel type of compile-time instrumentation and genetic algorithms to discover interesting test

inputs. Based on AFL, AFLFast [2] proposes strategies to systematically bias fuzzer towards exercising low-frequency paths. Libfuzzer [13] is an in-process, coverage-guided, evolutionary fuzzer which linked with the library under test, and feeds fuzzed inputs to the library via a specific fuzzing entrypoint. It tracks reached code areas and generates inputs by mutating the corpus of seeds in order to maximize code coverage. Both AFL [33] and Libfuzzer [13] are typical coverage guided fuzzers which are widely used in literature [2, 4, 24, 30, 32].

Some researches introduce extra information coverage to facilitate fuzzing on finding specific types of bugs. MUZZ [4] provides three types of instruments to discover and reveal multi-thread

Figure 3: Average killing rates for different fuzz campaigns in different time durations. The x-axis of each figure represents time points and the y-axis represents the achieved mutation killing rates.

vulnerabilities of the PUT during fuzzing. MemLock [30] tracks memory usage during fuzzing in order to trigger uncontrolled memory usage bugs. PerfFuzz [16] endeavors to generate pathological inputs which exercise hot spots of the PUT or with a higher total execution path length. To trigger vulnerabilities at semantic stage, Zest [24] proposes an approach with property-based testing [5] which build and mutate inputs in a semantic-valid way in order to detect vulnerabilities at semantic stage. This paper aims to enhance CGF to discover more bug-revealing inputs enhanced by mutation testing, which is distinct from the researches mentioned above.

Mutation Testing. Mutation testing measures the adequacy of testing with *mutation score*, or *mutation coverage*. In this regard,

many researches use mutation testing as a manner of feedback or guidance. For example, Mike and Yves [26] build a mutation-based fault local technique with a test suite constructed from mutation testing. Gordon and Andreas [7] use mutation testing to guide oracle constructions. Some test optimization techniques also benefit from mutation testing, such as test suite reduction [28] and test case prioritization [17]. Unlike these techniques, the goal of our approach is to use mutation testing to guide fuzzing towards finding inputs that are capable of finding bugs.

Differential Testing. Differential testing [21] detects bugs via comparing comparable systems. It amends the absence of oracles by using the outputs of different systems to validate the behaviors

of themselves interactively. Differential testing is widely adopted in situations that oracles are hard to obtain, such as *compiler testing* [27, 31], *DNN testing*[1, 19] and *regression testing* [11, 14]. In this paper, we utilize differential testing to enable the detection of mutants. We propose a configurable differential outputs checking frameworks. It checks the consistency of outputs during fuzzing. Mutants of which the outputs are inconsistent to those of the PUT is detected and will be considered as killed.

7 CONCLUSION

In this paper, we incorporate mutation testing with fuzzing in order to guide fuzzing towards detecting bugs. We conduct a well-designed experiment on 5 benchmarks with 3 techniques (2 modified techniques and 1 baseline) to evaluate the proposed approach. The experimental results show that mutation testing can make progress in terms of both code coverage and bug detection.

Our approach is general and can be extended to other CGF techniques. A larger scale experiments are worth conducting on other real world benchmarks as well as coverage guided fuzzers. Meanwhile, it is also important to investigate how different types of mutants can influence our approach. We leave these as future works.

ACKNOWLEDGMENTS

REFERENCES

[1] Muhammad Hilmi Asyrofi, Zhou Yang, and David Lo. 2021. Crossasr++: A modular differential testing framework for automatic speech recognition. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 1575–1579.

[2] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Trans. Software Eng.* 45, 5 (2019), 489–506.

[3] FT Chan, Tsong Yueh Chen, IK Mak, and Yuen-Tak Yu. 1996. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology* 38, 12 (1996), 775–782.

[4] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020.* USENIX Association, 2325–2342.

[5] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.* ACM, 268–279.

[6] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016.* ACM, 449–452.

[7] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010.* ACM, 147–158.

[8] Vijay Ganesh, Tim Leek, and Martin C. Rinard. 2009. Taint-based directed whitebox fuzzing. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings.* IEEE, 474–484.

[9] github. visited at 2022 March. Github. https://github.com/.

[10] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2015. Guidelines for Coverage-Based Comparisons of Non-Adequate Test Suites. *ACM Trans. Softw. Eng. Methodol.* 24, 4 (2015), 22:1–22:33.

[11] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. Differential regression testing for REST APIs. In *ISSTA '20: 29th ACM SIGSOFT International*

[12] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44.

[13] Libfuzzer group. 2017. LibFuzzer. https://llvm.org/docs/LibFuzzer.html.

[14] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and practices of differential testing. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019.* IEEE / ACM, 71–80.

[15] Hadi Hemmati. 2015. How Effective Are Code Coverage Criteria?. In *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015.* IEEE, 151–156.

[16] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018.* ACM, 254–265.

[17] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015.* IEEE Computer Society, 46–57.

[18] Ping Ma, Hangyuan Cheng, Jingxuan Zhang, and Jifeng Xuan. 2020. Can this fault be detected: A study on fault detection via automated test generation. *J. Syst. Softw.* 170 (2020), 110769.

[19] Yu-Seung Ma, Shin Yoo, and Taeho Kim. 2021. Selecting test inputs for DNNs using differential testing with subspecialized model instances. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 1467–1470.

[20] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Trans. Software Eng.* 47, 11 (2021), 2312–2331.

[21] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107.

[22] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[23] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019.* ACM, 398–401.

[24] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019.* ACM, 329–340.

[25] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. *Adv. Comput.* 112 (2019), 275–378.

[26] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Softw. Test. Verification Reliab.* 25, 5-7 (2015), 605–628.

[27] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+1 -version Differential Testing of Both JavaScript Engines and Specification. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021.* IEEE, 13–24.

[28] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014.* ACM, 246–256.

[29] Weisong Sun, Xingya Wang, Haoran Wu, Ding Duan, Zesong Sun, and Zhenyu Chen. 2019. MAF: method-anchored test fragmentation for test code plagiarism detection. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET).* IEEE, 110–120.

[30] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: memory usage guided fuzzing. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020.* ACM, 765–777.

[31] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 435–450.

[32] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.* USENIX Association, 745–761.

[33] Michal Zalewski. 2017. American Fuzzy Lop 2.5.2b. https://lcamtuf.coredump.cx/afl/.