μ^2: Using Mutation Analysis to Guide Mutation-Based Fuzzing

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

ABSTRACT

Coverage-guided fuzzing is a popular tool for finding bugs. This paper introduces μ^2, a strategy for extending coverage-guided fuzzing with mutation analysis, which previous work has found to be better correlated with test effectiveness than code coverage. μ^2 was implemented in Java using the JQF framework and the default mutations used by PIT. Initial evaluation shows increased performance when using μ^2 as compared to coverage-guided fuzzing.

CCS CONCEPTS

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

KEYWORDS

mutation-based fuzzing, mutation testing, differential testing

ACM Reference Format:
Isabella Laybourn. 2022. μ^2: Using Mutation Analysis to Guide Mutation-Based Fuzzing. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3510454.3522682

1 INTRODUCTION

Mutation-based fuzzing is an important quality assurance tool gaining popularity due to its effectiveness in finding critical bugs and security vulnerabilities in widely-used software systems. In mutation-based fuzzing, new inputs generated by randomly mutating known inputs are potentially selected for subsequent mutation according to some criteria. However, defining good selection criteria for a test input corpus is a challenge. We seek to generate corpora that will be effective in finding bugs in new versions of the program. Since bug finding effectiveness can be difficult to measure directly, code coverage is often used as a proxy. Covering code with a test is necessary for the test to find bugs in that code, but previous work shows that high levels of coverage are insufficient to indicate that a test input corpus will be effective at finding bugs [7]. Another commonly used proxy is mutation score, which measures a test's ability to distinguish the original version of a program from mutants, versions with small syntactic variations. Specifically, the mutation score is the percentage of mutants that a test distinguishes from the original. Previous work has found that a test's mutation score is a better predictor of its real fault detection rate than code coverage [8]. This paper explores using mutation score in addition to code coverage as selection criteria for mutation-based fuzzing.

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05.
https://doi.org/10.1145/3510454.3522682

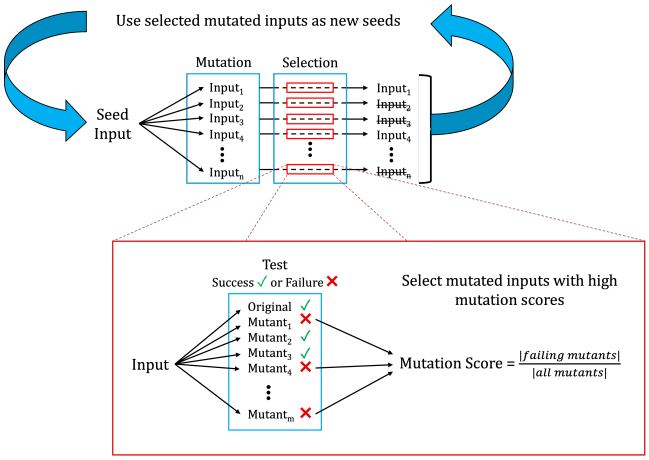

Figure 1: μ^2 uses mutation analysis to select which new test inputs to keep.

2 MUTATION ANALYSIS IN FUZZING

This paper introduces μ^2, a novel mutation-based fuzzing technique that adds mutation analysis to code coverage as selection criteria.

Figure 1 describes how μ^2 works. First, as in standard coverage-guided fuzzing, the algorithm selects a seed input from its corpus of seed inputs. Second, it mutates that input to generate new inputs. Then, for each of the new inputs, it runs the original version to check for new code coverage and identify opportunities for program mutation. Next, each new input is run on each mutant generated according to the identified program mutation opportunities. Finally, those inputs that kill new mutants or extend code coverage are added to the corpus of seed inputs for further mutation in later iterations of the loop. In summary, the standard coverage-guided fuzzing loop is augmented by the calculation of mutation score.

The feedback function in greybox fuzzing has been customized by researchers for a variety of applications [4, 6, 9–12, 15]. However, to the best of our knowledge, this is the first attempt to use mutation analysis for the feedback function. Our implementation is built for Java utilizing fuzzing framework JQF [13], which supports plugins for new mutation-based fuzzing algorithms, and its implementation of semantic fuzzer Zest [14]. Major challenges presented in the implementation of mutation-guided fuzzing include that:

- running thousands of program mutants for each generated input is prohibitively expensive, and
- mutation analysis requires good oracles to determine mutant death, but traditional fuzzing only uses crashes.

2.1 Improving Fuzzing Throughput

For practical fuzzing algorithms, throughput, the rate at which a fuzzer can generate inputs, is critical. In order to increase the throughput of our implementation of μ^2, we are working to optimize our mutation analysis.

The program mutations used are members of the default set for mutation analysis suite PIT [5], but, in μ^2, optimizations are implemented such that we do not start a new JVM for each mutant or mutant type as PIT does. In particular, μ^2 makes use of Java's `ClassLoader` hierarchy, defining three new subclasses of `ClassLoader`.

The `CartographyClassLoader` (CCL) loads the program without any mutation, but adds instrumentation to monitor code coverage and identify potential mutation sites. The CCL loads the app classes only once. Each program mutant has a single mutation associated with a unique `MutationClassLoader` (MCL). Each MCL also loads the app classes only once.

Consider the following example class `Foo`:

```
class Foo {
    static int add(List<Integer> nums) {
        int sum = 0;
        for(Integer i : nums) sum = sum + i;
        return sum;
    }
}
```

The CCL will load `Foo` once and generate an MCL for each potential mutation site in `Foo`, such as `sum + i`, which can be mutated to `sum - i`, and the `return` statement, which can be mutated to return `null` instead of `sum`. A duplicate class `Foo` will be loaded for each of these mutants by the corresponding MCL.

One advantage of this approach is that the CCL discovers potential mutations dynamically, so μ^2 only considers mutations in classes loaded while running the test input. Mutations in code not covered by running the test would not be killed, so spending extra time to run the programs loaded by their corresponding MCLs is unnecessary. However, using a ClassLoader for each mutant in this manner presents a key challenge: using enormous amounts of memory when loading library classes with every MCL. For example, `Foo` uses `java.util.List`, but `List` isn't mutated, and having an identical duplicate of `List` for every MCL unnecessarily consumes memory. To combat this problem, we use the intermediate `ClassLoader` (ICL) as the parent ClassLoader for both the CCL and the MCLs (depicted in Figure 2). It is designed to load library classes just once while forcing the CCL and MCL to load program classes.

2.2 Using Stronger Oracles

Traditional fuzz drivers use test functions with the signature $X \rightarrow$ `void` for some input type X and expect the test function to throw an exception when the outcome is unexpected. To facilitate a stronger oracle than "no exception" for mutation analysis, μ^2 instead uses test functions that produce an output that can be compared across mutants. To this end, it defines two types of test functions: the output function with a signature $X \rightarrow Y$ where Y is an output type, and the comparison function with signature $(Y, Y) \rightarrow$ `Boolean` to return whether the two outputs are the same. For example, consider a traditional test function for `Foo`:

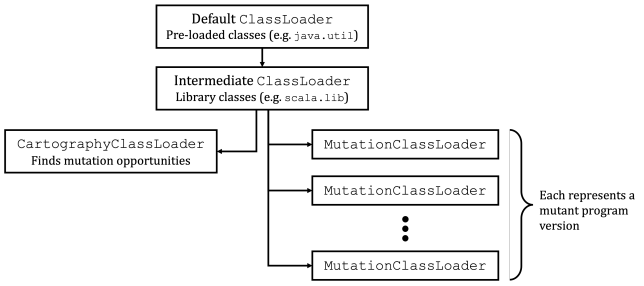

Figure 2: μ^2 runs different versions of the program under test by creating multiple `MutationClassLoaders`.

```
void testFooAdd(int a, int b) {
   assert(Foo.add(asList(a, b)) == Foo.add(asList(b, a)))
}
```

This test checks a property of the addition function, but μ^2 can directly compare the results of different mutants using a pair of functions such as these:

```
Integer outputFooAdd(int a, int b) {
    return Foo.add(a, b)
}
Boolean compareFooAdd(Integer out1, Integer out2) {
    return out1 == out2;
}
```

Here, `outputFooAdd` will be invoked on the versions of `Foo` loaded by the CCL and each MCL and `compareFooAdd` will be used to compare the output of the original version of `Foo` (loaded by the CCL) to the output of each mutated `Foo`.

3 FUTURE WORK AND EVALUATION

Performance in mutation analysis of μ^2 was compared on commonly available mid-size codebases to Zest, which uses coverage-guided fuzzing (see Table 1). These preliminary results show that μ^2 regularly caught more mutants than coverage-guided fuzzing when given time to generate the same amount of test inputs. Because some mutants will never be killed, we found it useful to examine the number of mutants killed by each strategy for a given codebase.

Codebase	Generated Test Inputs	Mutants Caught By Both*	Zest Only	μ^2 Only
PatriciaTrie [1]	10000	94.9	3.4	**7.5**
ChocoPy [16]	30000	248.5	4.5	**17.2**
Gson [2]	1000000	240.1	5.4	**19.2**
CommonsCSV [3]	1000000	152.8	0.0	**1.2**

Table 1: Results for mutants caught by Zest and μ^2 averaged over 10 random seeds. *caught by both Zest and μ^2

μ^2's current implementation requires more time and memory for the same number of inputs as Zest. Goals for future work include optimization of μ^2, comparison between μ^2 and standard coverage-guided fuzzing using the same amount of time rather than inputs, comparison by metrics other than mutation analysis, and further examination of patterns in the types of mutants killed by μ^2.

REFERENCES

[1] 2019. org.apache.commons.collections4.trie.PatriciaTrie. GitHub Repository. v4.3.

[2] 2021. com.google.gson. GitHub Repository. v2.8.9.

[3] 2021. org.apache.commons.csv. GitHub Repository. v1.9.0.

[4] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring Deep State Spaces via Fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1597–1612. https://doi.org/10.1109/SP40000.2020.00117

[5] 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* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[6] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2829–2846. https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[7] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271

[8] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[9] 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* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/3213846.3213874

[10] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-Box Fuzzing towards Combinatorial Difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1024–1036. https://doi.org/10.1145/3377811.3380421

[11] Raveendra Kumar Medicherla, Raghavan Komondoor, and Abhik Roychoudhury. 2020. *Fitness Guided Vulnerability Detection with Greybox Fuzzing*. Association for Computing Machinery, New York, NY, USA, 513–520. https://doi.org/10.1145/3387940.3391457

[12] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2019. DifFuzz: Differential Fuzzing for Side-Channel Analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 176–187. https://doi.org/10.1109/ICSE.2019.00034

[13] 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 398–401. https://doi.org/10.1145/3293882.3339002

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

[15] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (oct 2019), 29 pages. https://doi.org/10.1145/3360600

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