Mutation-Guided Fuzzing
Extending Coverage-Guided Fuzzing with Mutation Analysis

Bella Laybourn

Abstract

Fuzz testing is a popular technique for finding software bugs and security vulnerabilities using randomized test-input generation. State-of-the-art fuzzing tools perform coverage-guided fuzzing, but more coverage doesn't necessarily mean better fault detection capability [1]. This project aims to develop a new fuzzing technique that is guided by fault detection via mutation analysis [2] instead of coverage and show that, by replacing code coverage with mutation score as the metric for evaluating and guiding fuzz-generated inputs, test suites can be created that will augment and/or improve on those suites generated using code coverage as a metric. A mutation analysis program built to emulate the default set of mutations in mutation analysis software PIT [3] was used to create a guidance for use in fuzzer JQF [4]. Running JQF with this guidance is compared to running JQF with coverage-guided fuzzing. Results are currently inconclusive and pending testing on larger benchmarks.

Background

This project combines mutation analysis with coverage-guided fuzzing to make mutation-guided fuzzing, with the idea that selecting for higher mutation analysis scores would drive fuzzing toward tests that find a different type of bug than coverage-guided fuzzing alone.

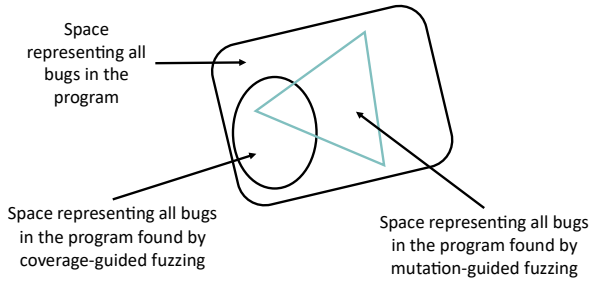

Figure 1: Concept diagram of proposed improvement from mutation-guided fuzzing

Coverage-Guided Fuzzing

Guided fuzzing generates new inputs by evolving from an initial set of inputs in a process similar to survival of the fittest. Each new generation of inputs is derived by mutating the surviving inputs from the last generation. In this approach, a fitness measure is used to determine which inputs survive. Subtle mutations to the previous generation inputs can translate to subtle changes in the program execution and potentially uncover new bugs. Similar to how, in nature, environmental pressures can select species characteristics, it is believed that the selection of a fitness measure will lead to input mutations favoring one type of bug, while the choice of a different fitness measure might lead to favoring a different set of bugs. One popular fitness measure is code coverage based on the premise that the more code a test executes, the better it is for finding bugs. In this scenario, a set of inputs is sequentially acted upon by the program to be tested and the amount of program code executed is tracked along with the presence of any incorrect program operation (occurrence of a bug). If a particular input leads to execution of a previously unexecuted portion of the code or demonstrates a bug, then that mutation is allowed to survive to the next generation and is then mutated to provide new set of inputs.

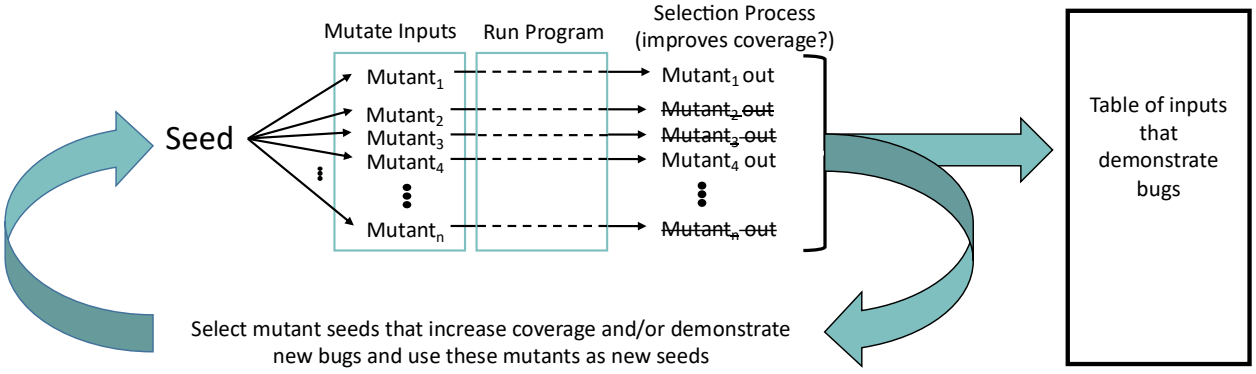

Figure 2: Coverage-Guided Fuzzing

Coverage guided fuzzing tools have gained popularity due to their effectiveness at finding critical bugs and security vulnerabilities in widely used software systems [5]. It is recognized,

however, that coverage-guided fuzzing is not sufficient to find all bugs in a program, and augmentations of coverage-guided fuzzing systems have been introduced that expand bug finding capabilities, such as Zest, which expanded from syntactic to semantic bugs. This project focused on adapting mutation analysis techniques to augment coverage-guided fuzzing in order to increase the number of bugs found.

Mutation Analysis

Mutation analysis is a method for analyzing a test suite (often a set of test inputs) by creating multiple versions of an original program (mutant programs) by (in this project) introducing modifications to the bytecode, thereby creating bugs. These modifications can be applied to different bytecode instructions, including math operators, conditional operators, void method calls, and return values. Each modification is used to create a separate unique version of the program. Multiple versions are made creating a number of mutant programs. The inputs of a test suite are applied to each of these mutant programs, and the more mutant programs a test suite fails (indicates a bug) on, the better the test suite is expected to be.

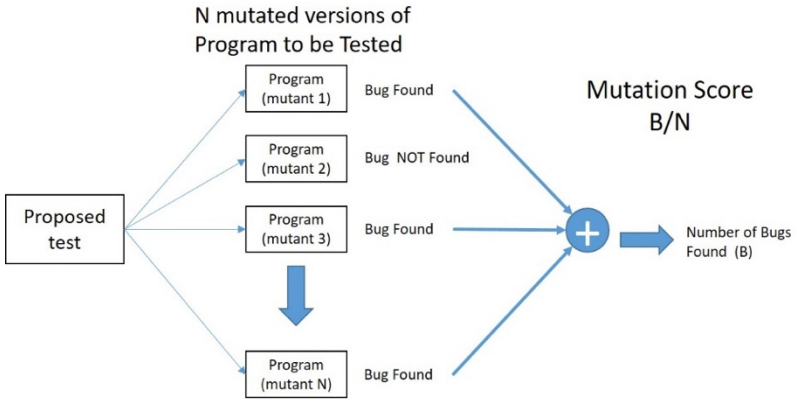

Figure 3: Mutation Analysis

Although the mutated versions of the program are based on artificially generated bugs, it has been shown that a test suite's ability to detect introduced bugs (i.e. mutations) strongly correlates

with its ability to identify bugs in real programs [2]. The associated *mutation score* (ratio of identified mutations to total mutations) is an effective way to measure the relative ability of a test suite to adequately test a real program.

Mutation-Guided Fuzzing: Combing Mutation Analysis with Coverage Guided Fuzzing

To combine mutation analysis with coverage guided fuzzing, the program being tested is replaced by multiple versions of itself with modified bytecode, each containing a different modification. The fitness test is augmented to not only place value on increased code coverage and finding a bug in the original program, but also include any inputs that identify not-previously-identified artificially created bugs in the mutated programs, as finding new bugs improves mutation score. Those inputs that are deemed fit are then mutated to create the next generation of inputs.

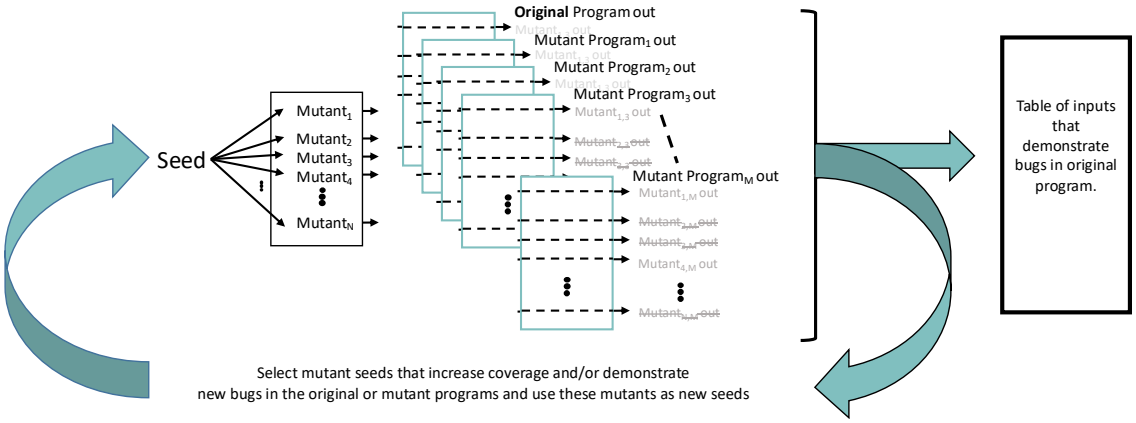

Figure 4: Mutation-Guided Fuzzing

Implementation of Mutation Guided Fuzzing

JQF

JQF was used as the fuzzing framework for this project. It is a feedback-directed fuzz testing platform for Java [4] and uses guidance plugins over a fuzzing framework in order to run guided

fuzzing. By plugging in different guidances, different types of guided fuzzing can be implemented. A mutation guidance plugin was built by modifying the plugin for Zest. Zest is an algorithm that biases coverage-guided fuzzing towards producing semantically valid inputs; that is, inputs that satisfy structural and semantic properties while maximizing code coverage [5]. In order to run mutation-guided fuzzing using JQF's framework, the framework had to be modified to allow for multiple runs of the same fuzzer-generated input (once for each mutation). This was accomplished by moving the run functionality from the framework to the guidance plugins and adding a default implementation running the input once (as was done originally in the framework) so the existing guidance plugins would not be disrupted.

Mutators

PIT is a mutation analysis system that provides test coverage for Java and the JVM [3]. The default set of mutators defined by PIT was used to build the library of mutators for this project. This default set was re-implemented for use in creating the mutation guidance plugin for JQF. Each mutator takes the form of replacing one bytecode instruction with a series of bytecode instructions (length ≥ 1). Implementing this was complicated slightly by some mutators having additional requirements (such as void function removal applying solely to those function invocations associated with void functions), but these additional requirements were also stored in the mutators' programming. A *mutation opportunity* is defined for this project as a place in code that meets all the requirements for applying a mutator.

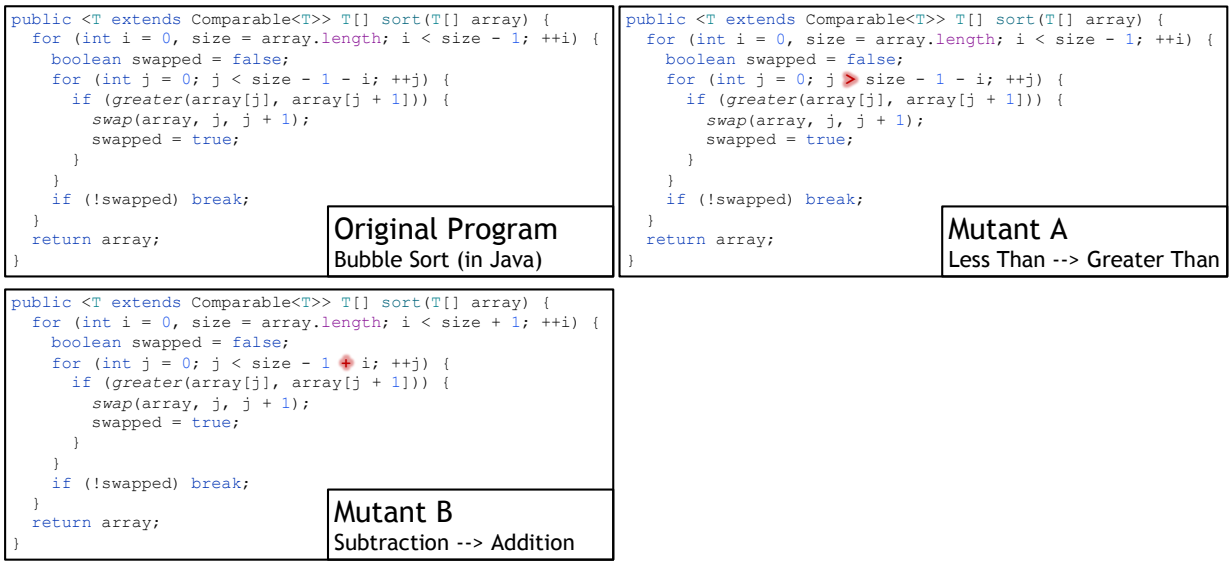

Figure 5: Example Mutations of a BubbleSort implementation [6]

ClassLoaders

This project is implemented in Java. Java programs are compiled into bytecode and run on the JVM. ClassLoader objects are responsible for finding and loading the bytecode for the classes. Mutation was implemented by creating MutationClassLoaders that would modify the bytecode by applying a specified mutator at a specific mutation opportunity before returning it. ObjectWeb ASM provides a library for reading and writing bytecode, and this was used in the MutationClassLoaders for the bytecode modification. In addition to the MutationClassLoaders, a CartographyClassLoader loads the bytecode normally, but reads it (again using ObjectWeb ASM) and finds the mutation opportunities. The CartographyClassLoader is responsible for creating the MutationClassLoaders based on the mutation opportunities it finds.

The mutation guidance also included modifications to extend the run and result-handling functions. The run function was extended to include running with the CartographyClassLoader and each generated MutationClassLoader and the result-handling function was extended to

include saving fuzz inputs that kill mutants. The ClassLoaders were extended with the instrumentation used in JQF's InstrumentingClassLoader that allows for termination of programs that exceed some high constant number of control jumps in order to prevent hanging on an infinite loop.

Mutate Goal

A Maven goal to run mutation analysis on a set of saved inputs was also added to JQF so the results of mutation-guided fuzzing could be reproduced.

Results

Product

The mutation guidance and goal can be accessed at https://github.com/saphirasnow/JQF/tree/bella. The mutation guidance can be run by following instructions for Zest, adding the flag `-Dengine=mutation` on `mvn jqf:fuzz` terminal commands. The mutate goal can be accessed with `mvn jqf:mutate` and the same `-Dclass` and `-Dmethod` flags used when running the fuzzing.

Example run formats:

```
mvn jqf:fuzz -Dclass=package.class -Dmethod=method -Dengine=mutation
mvn jqf:mutate -Dclass=package.class -Dmethod=method
```

Evaluation

Mutation-guided fuzzing and coverage-guided fuzzing were both run on a sort microbenchmark with no notable differences. This is most likely due to the small size of the benchmark. This

project was focused on the building of functionality for mutation-guided fuzzing. For next steps, the goal is to run on larger benchmarks to better observe any differences between coverage-guided and mutation-guided fuzzing. This will require greater efficiency in programming as well as mutant selection, as running generated tests repeatedly on all the possible mutants in a large-scale program currently exceeds reasonable time and memory constraints.

Sources

1. L. Inozemtseva and R. Holmes. “Coverage is not strongly correlated with test suite effectiveness”. ICSE 2014.
2. 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?” FSE’14
3. PIT Mutation Testing — https://pitest.org
4. Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. **JQF: Coverage-Guided Property-Based Testing in Java**. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’19), July 15–19, 2019, Beijing, China. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3293882.3339002
5. 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’19), July 15–19, 2019, Beijing, China. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293882.3330576
6. V. Upadhyay and P. Nikita. Bubble Sort — https://github.com/TheAlgorithms/Java/blob/master/ Sorts/BubbleSort.java