Mutation Analysis of Smart Contracts

Alex Groce

March 12, 2019

Abstract

In mutation analysis (also called mutation testing), large numbers of small syntactic changes to a program are introduced. These variations (called mutants) are considered "killed" if a test suite can distinguish them from the original, unmodified program, and surviving if it cannot. Mutation score (percent killed mutants) is thus a measure, stronger than code coverage, of the fault detection power of a test suite. Mutation analysis is typically used to evaluate the quality of a test suite, or perhaps a test-generation technique, and most often, at least until recently, only in an academic research setting. This work proposes (1) use of *differential* mutation result comparison to compare and improve static analysis tools, (2) use of statistics of such mutation scores to approximately compare the "implicit specification" strength of various programming languages, and (3) use of mutation analysis over combined static and dynamic analyses, combined with prioritization of surviving mutants, to strengthen smart contract testing and verification efforts.

1 Introduction

In an ASE 2015 [1] paper and a 2018 journal extension [2] of that paper, Groce et al. proposed examining individual mutants that survive a formal verification or automated test generation process to detect and correct weaknesses in a specification or test generator.

2 Differential Mutation Comparison

We can say that a static analysis tool kills a mutant when the number of non-informational warnings or errors produced with respect to the code increases for the mutated version, compared to the original code. This difference is most informative and easily interpreted when the original code produces no warnings or errors (it is "clean"); in other cases, the tool may detect the mutant, but only change a previously generated message on the code in question, or on other code, due to the change, leading to an underestimate of the tool's effectiveness.

The value of the differential comparison lies in a few key points. First, this is a measure that does not reward a tool that produces too many false positives.

The tool cannot simply flag all code as having a problem or it will perform poorly at the task of *distinguishing* the mutated code from non-mutated (and presumably at least *more* correct) code. Based on verification and testing uses of mutations, it is safe to say that at least 50, and likely 60-70% or more, of mutants that are not semantically equivalent to the original code are actually fault-inducing, so the task presented to a static analysis tool is the generalization of the task we ideally expect static analysis to perform: to identify faulty code, without executing it, and, most critically, to distinguish faulty from correct code. Obviously, many faults cannot be identified statically without a complete specification, or at low cost, but the measure of performance here is relative to other tools applied to the same code.

This approach can also be used to improve tools. If a tool fails to detect a mutant, but another tool does kill that mutant, the pattern involved can be examined to see if it offers a potential to improve the non-detecting tool. In some cases, the change would produce too many false positives, but it is worth considering, especially when more than two tools are available to compare, and a majority do kill a mutant.

3 Cross-Language Comparison

The best Solidity analysis tool (Slither) detects 10% or more of mutants in clean code. Very preliminary results suggest that even well-engineered, widely-used tools in other languages (Python and C thus far) detect more in the range of 3-5% of mutants. It is likely that the almost domain-specific nature of Solidity code, and the ability to thus use an implicit specification of behavior, the stronger type system (than C or Python), and the lack of difficult-to-analyze code such as UI, extensive data storage and manipulation, etc. makes Solidity code genuinely more suited to purely static analysis.

4 Full-Spectrum Mutation Analysis

References

[1] Alex Groce, Iftekhar Ahmed, Carlos Jensen, and Paul E McKenney. How verified is my code? falsification-driven verification. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 737–748. IEEE, 2015.

[2] Alex Groce, Iftekhar Ahmed, Carlos Jensen, Paul E McKenney, and Josie Holmes. How verified (or tested) is my code? falsification-driven verification and testing. *Automated Software Engineering Journal*, 25(4):917–960, 2018.