Mutation Analysis of Smart Contracts

Alex Groce

March 14, 2019

Abstract

In mutation analysis (also called mutation testing), large numbers of small syntactic changes to a program are introduced, under the assumption that if the original program is correct, the modified program will be faulty, and the change will be the location of a fault. 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. We focus on smart contracts due to certain characteristics that are unusual or even unique to smart contract code, that make them especially suitable for a mutation-based approach.

1 Introduction

Mutation testing [4, 2] uses small syntatic changes to a program to introduce synthetic "faults," under the assumption that if the original version of a program is (mostly) correct, most small changes will therefore introduce a fault. For the most part, mutation analysis has been used to evaluate test suites by computing a score (the number of mutants the suite detects, or "kills"). Most such use has been in research efforts, rather than practical testing efforts, though there has been sporadic use by interested developers. In an ASE 2015 [7] paper and a 2018 journal extension [8] 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. The approach was able to expose bugs in a heavily-tested module of the Linux kernel [1] and improve a heavily used test generator for the `pyfakefs`

file system. Recently, mutation analysis has been adopted in industrial settings, though not for actual examination of all surviving mutants [13, 12], a practice that is hard to scale to large code bodies.

1.1 Why Smart Contracts?

Mutation analysis and smart contracts are arguably a natural fit, for several reasons:

- Smart contract code (especially in Solidity) is usually relatively *small*. The explicit relationship between amount of computation performed, bytecode size, and deployment/execution cost in Ethereum introduces a strong incentive to minimize source code size. Only two contracts in the etherscan data we examined had more than 3500 lines of code, and the largest contract had under 12KLOC. The cost of mutation analysis, in terms of generating mutants, evaluating whether they are killed, and esepcially determining whether surviving mutants indicate weaknesses in testing, is directly proportional to code size, so smaller code makes mutation analysis more attractive.

- Smart contract code is worth spending human and computational efforts, including mutation analysis efforts, in testing and verifying. Because contracts may control considerably economic value, they are inherently high-criticality targets. This also means that examining surviving mutants should be easier than in some other settings, since there is a strong motivation to produce effective test suites.

- As noted below, it is likely that static methods can detect more bugs in smart contracts than in other code; the same pressures that generally keep LOC low keep contracts focused on core functionality, not irrelevant (and untested) bells and whistles. Most code is doing something that can be checked, and in many cases the specification may be implicit in the almost-DSL like concentration of blockchain code; even if Solidity is Turing-complete, the range of things implemented in Solidity may be more restricted than for general-purpose programming languages.

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 meant to be mostly *relative* to other tools applied to the same code.

Second, and critically, this is an *automatable* method that can provide an evaluation of static analysis tools over a large number of target source code files, without needing human effort to classify results as real bugs or false positives. It is not clear that any other fully automatic method is competitively meaningful; it is possible that methods based on code changes from version control provide some of the same benefits, but these require classification of changes into bug-fixes and non-bug-fixes, and of course require version control history. Also, history-based methods will be biased towards precisely those faults humans (or tools) were able to detect and fix.

It is the combination of differential comparison and mutation that is key. Differential comparison of tools, alone, is not meaningful. Consider a comparison of the number of issues detected between two tools over a single program, or over a large set of programs. If one tool finds more problems than another, it may mean that the tool is more effective at finding bugs; but it may also mean that it has a higher false positive rate. Without human examination of the individual issues, it is impossible to be sure, or even (in cases where the tools are reasonably comparable) to make an informed guess. Adding mutants, however, provides a foreground to compare to this background. In particular, for a large set of programs, the most informative result will be when 1) tool A reports fewer issues on average than tool B over the un-mutated programs but 2) tool A also detects more mutants. This is strong evidence that A is simply better all-around than B; it likely has a lower false positive rate *and* a lower false negative rate. While it is not proof of this claim, it is hard to construct another plausible explanation for reporting *fewer* issues on un-mutated contracts while still detecting *more* mutants. Other than simply being both more precise and more accurate, how else could a tool clearly distinguish mutated from un-mutated code? This is the pattern we see with mutation analysis of smart contracts: Slither reports fewer issues on the original code and detects more mutants.

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.

2.1 Application to Solidity Static Analysis Tools

We used the universalmutator tool [9, 10] to compare three Solidity static analysis tools using this differential approach: Trail of Bits' Slither [5], Securify [15], and SmartCheck [14].

2.1.1 Solidity Compiler Version 0.5.4 Results

Note that these results for Securify are likely invalid, due to ignoring `Warning` level messages and only considering `Violation` messages.

We ran an initial comparison of the static analysis tools over a set of contracts that compile with Solidity version 0.5.4, the latest at the time of writing. These contracts were taken from the **Solidity by Example** webpage on the official Ethereum site, the **Mastering Ethereum** book, and the (small) set of contracts in the etherscan results that compile with 0.5.4, for a total of 30 contracts, some of which are essentially the null contract. These contracts produced 2,623 valid mutants to use in comparing the tools. Slither performed better than the other tools for this set of contracts, with a mean mutation score of 0.11 over all contracts with any valid mutants, and 0.06 over the five contracts for which none of the tools found any issues. By comparison, SmartCheck had means of 0.04 and 0.0 respectively for these sets, and Securify scores of 0.01 and 0.0. The mutation score differences for all contracts (but not for clean contracts only) are significant by Wilcoxon score. In contrast, while Slither detected more mutants, it reported considerably fewer issues on the original contracts, with a mean issue count over all contracts with valid mutants of only 0.29 issues reported; SmartCheck reported a mean of 0.52 issues, and Securify (even with only **Violation** level issues counted) reported a mean of 1.52 issues per contract.

In terms of cross-comparison on individual mutants, Slither also performed well. It killed 265 mutants not detected by SmartCheck, and 294 mutants not killed by Securify. In contrast, SmartCheck only detected 140 mutants not killed by Slither, and Securify only detected 40 mutants not detected by Slither. Only 13 mutants were detected by both SmartCheck and Securify, but not by Slither, all of them involving a change of `msg.sender` to `tx.origin`.

2.1.2 Solidity Compiler Version 0.4.24 Results

An analysis of etherscan contracts that compile with `solc` version 0.4.24 is in progress. At present the 46,769 valid mutants generated by 100 randomly selected contracts are being analyzed. This analysis is using a better definition of detection for Securify. So far, Slither results are complete, with a mean score of 0.09 over all contracts and 0.11 over contracts that are clean with respect to Slither alone. Analysis of 35 contracts with SmartCheck yields a mean score

of 0.05 for all contracts, and 0.02 for contracts that are clean with respect to SmartCheck alone.

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. As noted above, 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

Finally, the practical Trail of Bits/Crytic application of all this (other than improving/evaluating Slither) is the development of a framework for automatic mutation analysis of Solidity contracts, where the tool:

1. Generates mutants of the contract being analyzed.
2. Runs Slither (and any other static tools) to throw out mutants easily detect statically.
3. Runs Echidna / Manticore to find surviving mutants.
4. Ranks those mutants so that a user can quickly look at very different mutants the analysis cannot detect, and mark them as interesting/uninteresting, which provides feedback for presentation of further mutants.

To our knowledge, previous mutation analysis has never ignored mutants that are easily detected by static tools. This has some serious benefits: some faults are hard to detect dynamically, but easy to detect statically, or have impact in terms of best-practices or code maintainability but not in dynamic behavior, and these are not the mutants users need to see, when looking to improve their tests or test generation. Because even in this setting, examining all mutants may be impractical, we plan to develop methods, based on similarity and heuristic estimates of interestingness (e.g., amount of bytecode change), to rank mutants for presentation. One route to such methods is to use the FPF [6] method used in "fuzzer taming" previously [3, 11].

References

[1] Iftekhar Ahmed, Carlos Jensen, Alex Groce, and Paul E. McKenney. Applying mutation analysis on kernel test suites: an experience report. In *International Workshop on Mutation Analysis*, pages 110–115, March 2017.

[2] Timothy Alan Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–233. ACM, 1980.

[3] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–208, 2013.

[4] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[5] Josselin Feist, Gustavo Greico, and Alex Groce. Slither: a static analyzer for solidity. In *International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019.

[6] Teofilo F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[7] 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.

[8] 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.

[9] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. Regexp based tool for mutating generic source code across numerous languages. `https://github.com/agroce/universalmutator`.

[10] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 25–28, New York, NY, USA, 2018. ACM.

[11] Josie Holmes and Alex Groce. Causal distance-metric-based assistance for debugging after compiler fuzzing. In *IEEE International Symposium on Software Reliability Engineering*, 2018.

[12] Goran Petrović Marko Ivanković, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *Proceedings of the International Workshop on Mutation Analysis (Mutation). IEEE Press, Piscataway, NJ, USA*, pages 47–53, 2018.

[13] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, pages 163–171, New York, NY, USA, 2018. ACM.

[14] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16, 2018.

[15] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, 2018.