Using Differential Mutation Analysis to Compare and Improve Static Analysis Tools

ABSTRACT

Many programming languages offer multiple static analysis tools that offer to detect faults in code witthout executing it. Understanding the strengths and weaknesses of tools, and performing direct comparisons of their effectiveness is diffcult; it usually involves either manual examination of differing warnings on real code, or the bias-prone construction of artificial test cases. In practice, comparisons tend to be limited to superficial, anecdotal discussions in the informal literature (e.g., blog posts by software developers), or purely research-community-oriented evaluations made by the authors of new tools seeking to publish their results. This paper proposes a novel automated approach to comparing static analysis tools, based on producing *mutants* of real code, and comparing mutation detection rates for tools to their warning rates on the original code. In addition to making tool differences quantitatively observable without extensive manual effort, this approach offers a new way to detect and fix omissions in a static analysis tool's set of detectors. We present an extensive comparison of three well-known Solidity smart contract static analysis tools, and show how using an automatic prioritization of our results allowed us to add three new detectors to the best of the tools. We also evaluate popular Java and Python static analysis tools and discuss their strengths and weaknesses.

ACM Reference Format:

. 2020. Using Differential Mutation Analysis to Compare and Improve Static Analysis Tools. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Static analysis of code is one of the most effective ways to avoid defects in software. Static analysis can find problems that are extremely hard to detect by testing, e.g. when the inputs triggering a bug are

Mutation testing [2, 6] (or mutation analysis, the term we will use in this work, for reasons that will become clear) uses small syntactic 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 ratio 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 [9] paper and a 2018 journal extension [10] 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 [14, 16], a practice that is hard to scale to large code bodies.

1.1 Differential Mutation Analysis

We can say that a static analysis tool kills a mutant when the number of (non-informational or optimization related) 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"); for non-clean code, a tool conceivably could detect the mutant, but only change a previously generated warning, not add an additional warning, leading to an underestimate of effectiveness. However, even for non-clean code, most detected mutants will produce a new warning.

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 mutation, 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 without unreasonable analysis cost and precision, but the measure of performance here is meant to be mostly *relative* to other tools applied to the same code; this is primarily a *differential* approach.

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 requiring 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.

Conference'17, July 2017, Washington, DC, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

https://doi.org/10.1145/nnnnnnn.nnnnnnn

It is the combination of differential comparison and mutation that is key. Differential comparison of tools, alone, is not meaningful, without additional effort; naïve approaches will not work [5]. 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 emits more warnings and errors 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. Using 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 having better precision and recall, how else could a tool effectively distinguish mutated from un-mutated code?

Finally, even when two tools have similar quantitative results, examining individual mutants killed by one tool but not by another allows us to understand strengths and weaknesses of the tools, in a context that makes comprehending the cause of the detection (or lack of it) easy: the difference between the un-mutated code and mutated code will always be small and relatively simple. This leads to the idea of improving static analysis tools by examining mutants detected by another tool (thus known to be in-principle detectable) but not by the tool to be improved. Of course, any faults in code, not just mutants, could serve this purpose. But, again, the automatic nature of mutation generation, and the presumption that the mutation is indeed a fault, is useful. Moreover, because mutants follow syntactic patterns, searching for similar mutants/faults the tool to be improved does not detect is much easier than with arbitrary faults, and can be partly automated. Of course, knowing which mutation patterns are of interest requires human effort. As with efforts to improve test suites, manually searching through all mutants can be an onerous task, especially for large-scale evaluations. We therefore also introduce the idea of prioritizing mutants, using a Furthest-Point-First [8] algorithm and distance metric inspired by previous work on helping developers sort through failing test cases and avoid duplicates [4], to help static analysis tool developers find interesting patterns without wading through numerous uninteresting duplicates.

1.2 A Simple Example

Consider the code in Figure 1, from the Solidity 0.4.24 "Introduction to Smart Contracts". The Universal Mutator tool [11, 12], which has been extensively tuned for Solidity mutation (and is the only smart contract mutation tool referenced in the Solidity documentation (https://solidity.readthedocs.io/en/v0.5.12/resources.html)) produces seven valid, non-redundant mutants for this trivial example code. Both the public version of Slither [7] and SmartCheck [17] produce a small number (three and two, respectively) of low-severity, informational, warnings for this code. Both tools also detect (by increasing the number of warnings produced) four of the seven mutants. However, only one of the mutants detected is common to both tools: both tools detect changing the `return` statement in the `get` function to a call to `selfdestruct` the smart contract. Slither, but not SmartCheck, also detects replacing the assignment of `storedData` in `set` with either a `selfdestruct` or `revert`, or simply removing it altogether. SmartCheck, on the other hand, detects removing the `return` in `get` or replacing it with a `revert`, or removing the `public` visibility modifier for `get` [1]. If we restrict our analyis to issues with a severity greater than informational, SmartCheck detects no mutants of the contract, while Slither still reports that some mutants *allow an arbitrary caller to cause the contract to self destruct.* This simple example shows how our approach works on a small scale. Using large numbers of larger, more realistic contracts makes it possible to extract the same kind of information, on a much larger scale. Prioritization of mutants is not very useful here (ranking three mutant saves little effort), so we will show the utility of that approach in our full Solidity results.

```
pragma solidity ^0.4.0;
contract SimpleStorage {
    uint storedData;
    function set(uint x) public {
        storedData = x;
    }
    function get() public view returns (uint) {
        return storedData;
    }
  }
```

Figure 1: A simple example Solidity smart contract from https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html.

1.3 Contributions

This paper offers the following contributions:

- We propose a differential approach to comparing static analysis tools based on the insight that program mutants provide an automated source of simple, easy-to-understand program changes that are likely faults.
- We propose a definition of mutant killing that works well in a static analysis context.
- We introduce a simple scheme for prioritizing mutants that helps users understand the results of analysis and guide efforts to improve, rather than simple compare, tools.
- We apply our method to an extensive, in-depth comparison of three Solidity smart contract analysis tools, and show how prioritization allowed us to easily identify (and build) three new detectors for the most effective of these tools.
- We further provide results for comparisons of popular Java and Python static analysis tools, showing the general usefulness of our methods, and giving a new picture of the comparative effectiveness of these tools.

[1]Slither's "missing return" detector is only available in the private version of slither, or through the `crytic` service provided by Trail of Bits.

2 COMPARISONS OF STATIC ANALYSIS TOOLS

2.1 Solidity Smart Contract Tools

2.2 Java Tools

2.3 Python Tools

2.4 Threats to Validity

3 RELATED WORK

The goal of "analysing the program analyser" [3] and applying better automated methods to evaluate and improve analysis tools has become recently more popular and, we suspect, more possible. The irony of using mostly ad-hoc, manual methods to test and understand static analysis tools is apparent; however, the fundamentally incomplete and heuristic nature of effective analysis tools makes this a challenge similar to testing machine learning algorithms [13]; most tools will not produce "the right answer" all the time, by their very nature. This is a result of both algorithmic constraints and basic engineering trade-offs.

Cuoq et al. [5] proposed the generation of random programs (*á la* Csmith [18]) to test analysis tools aiming for soundness, in limited circumstances, but noted that naïve differential testing of analysis tools was not possible. This paper proposes a non-naïve differential comparison (not, exactly, differential testing, however, in that only aggregate results are possible to interpret without human intelligence), based on the observation that the ability to detect program mutants offers an automatable way to tell which of two tools is better (for a given universe of examples, at least) at telling faulty from non-faulty code. Klinger et al. propose a different approach to differential testing of analysis tools [15]. Their approach is in some ways similar to ours, in that it takes as input a set of seed programs, and compares results across new versions generated from that seed. The primary differences are that their seed programs must be warning-free (which greatly limits the set of input programs available) and their tool must parse and understand the programs, and that the new versions are based on adding new assertions, not "breaking" the original code. We allow arbitarily buggy seed programs (thus many more real programs can be used), and can, due to the any-language nature of the mutation generator we use, operate even in new languages without further development effort. Further, their approach only identifies problems when tools are outliers compared to numerous other tools in either detecting a bug (precision) or not detecting it (soundness), and so requires comparing multiple tools. Our approach has some utility for even a single tool (you can just examine prioritized un-detected mutants). On the other hand, their approach can identify precision issues, while we offer no real help with false positives (in theory, you could apply their majority-vote method to mutants only a few tools flag, but mutants *are* usually faults. in contrast to their introduction of checks that may be guaranteed to pass, so this is probably not very helpful).

4 CONCLUSIONS AND FUTURE WORK

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 A 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. pages 220–233. ACM, 1980.

[3] Cristian Cadar and Alastair F Donaldson. Analysing the program analyser. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 765–768, 2016.

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

[5] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *NASA Formal Methods Symposium*, pages 120–125. Springer, 2012.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *International Conference on Software Engineering: Companion Proceeedings*, pages 25–28, 2018.

[13] Alex Groce, Todd Kulesza, Chaoqiang Zhang, Shalini Shamasunder, Margaret Burnett, Weng-Keen Wong, Simone Stumpf, Shubhomoy Das, Amber Shinsel, Forrest Bice, and Kevin McIntosh. You are the only possible oracle: Effective test selection for end users of interactive machine learning systems. *IEEE Transactions on Software Engineering*, 40(3):307–323, March 2014.

[14] 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.

[15] Christian Klinger, Maria Christakis, and Valentin Wüstholz. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 239–250, 2019.

[16] 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.

[17] 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.

[18] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation*, pages 283–294, 2011.