Evaluating and Improving Static Analysis Tools Via Differential Mutation Analysis

Alex Groce  
School of Informatics, Computing and Cyber Systems  
Northern Arizona University  
Flagstaff, AZ, USA  
agroce@gmail.com

Iftekar Ahmed  
Department of Informatics  
University of California, Irvine  
Irvine, CA, USA  
ifteka@uci.edu

Josselin Feist  
Trail of Bits  
New York, NY, USA  
josselin.feist@trailofbits.com

Gustavo Grieco  
Trail of Bits  
New York, NY, USA  
gustavo.grieco@trailofbits.com

Jiri Gesi  
Department of Informatics  
University of California, Irvine  
Irvine, CA, USA  
ffiriges@uci.edu

Mehran Meidani  
Department of Informatics  
University of California, Irvine  
Irvine, CA, USA  
mmeidani@uci.edu

Qihong Chen  
Department of Informatics  
University of California, Irvine  
Irvine, CA, USA  
chenqh@uci.edu

Abstract—Static analysis tools attempt to detect faults in code without executing it. Understanding the strengths and weaknesses of such tools, and performing direct comparisons of their effectiveness, is difficult, involving either manual examination of differing warnings on real code, or the bias-prone construction of artificial test cases. This paper proposes a novel automated approach to comparing static analysis tools, based on producing mutants of real code, and comparing detection rates over these mutants. 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 smart contract static analysis tools, and show how our approach allowed us to add three effective new detectors to the best of these. We also evaluate popular Java and Python static analysis tools and discuss their strengths and weaknesses.

Index Terms—mutation testing, static analysis, smart contracts

I. INTRODUCTION

Static analysis of code is one of the most effective ways to avoid defects in software, and, when security is a concern, is essential. Static analysis can find problems that are extremely hard to detect by testing, when the inputs triggering a bug are hard to find. Static analysis is also often more efficient than testing; a bug that takes a fuzzer days to find may be immediately identified. Users of static analysis tools often wonder which of multiple tools available for a language are most effective, and how much tools overlap in their results. Tools often find substantially different bugs, making it important to use multiple tools [32]. However, given the high cost of examining results, if a tool provides only marginal novelty, it may not be worth using, especially if it has a high false-positive rate. Developers of static analysis tools also want to be able to compare their tools to other tools, in order to see what detection patterns or precision/soundness trade-offs they might want to imitate. Unfortunately, comparing static analysis tools in these ways is hard, and would seem to require vast manual effort to inspect findings and determine ground truth on a scale that would provide statistical confidence.

Differential testing [41], [29], [56] is a popular approach to comparing multiple software systems offering similar functionality, but the wide divergence of possible trade-offs, analysis focuses, and the prevalence of false positives in almost all analysis results makes naïve differential testing not applicable to static analysis tools [14]. Mutation analysis[34], [15], [7] 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, such changes will often introduce a fault. For the most part, mutation analysis has been used to evaluate test suites by computing a mutation score, the fraction of mutants the suite detects, or “kills”. Groce et al. [24], [25] proposed examining individual mutants that survive a rigorous testing and verification effort to detect and correct weaknesses in testing, and found bugs in a heavily-tested module of the Linux kernel [2] and a widely used Python file system. Recently, mutation analysis has been adopted in industrial settings, though not for actual examination of all surviving mutants [45], [33].

Combining a differential approach and mutation analysis offers a novel way to compare static analysis tools, one useful to users wishing to select a good tool or set of tools, to researchers interested in the impact of precision/soundness trade-offs or different intermediate languages, and to developers of static analysis tools hoping to improve their tools.

We can say that a static analysis tool kills a mutant when the number of warnings or errors, which we call findings, increases with mutation. In order to make this definition useful, we ignore informational or optimization related warnings (e.g., if a mutant is merely stylistically suboptimal this is not “finding a fault”), That is, a mutant is killed when a tool “finds more (unique) bugs” for the mutated code than for the unmutated code. This difference may be most easily interpreted when the original code produces no findings; we call such
code clean (by analogy with Chekam et al.’s notion [10]). For non-clean code, a tool conceivably could detect the mutant, but only change a previously generated finding, not add an additional finding. However, even for non-clean code, most detected mutants should produce a new warning. We count findings, rather than consider their location or type, because some mutants cause a fault at a far-removed location. Forcing tools to produce an additional warning is a conservative and automatable estimate of mutant detection.

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 usually at minimum 40%, often 50-60%, and frequently up to 80%+ [2], [25], [39], of mutants are not semantically equivalent to the original code [42], [31], [49], so the task presented to a static analysis tool is simply the core functionality of static analysis: to distinguish faulty from correct code without execution. 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 relative to other tools applied to the same code; this is primarily a differential approach. While many mutants cannot be detected statically, the ones that are tend to be true positives; if they were real code changes, they would be faults. We manually confirmed that for a large portion of the detected mutants in our experiments, the changes were indeed ones that would be real faults if present in the 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 requiring human effort to classify results as real bugs or false positives. It is not clear that any other fully automated method is meaningfully 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 already in use were able to detect and fix. Rather than the hundreds [35] or at most few thousand of faults [53] in benchmark defect sets, our approach enables the use of many tens of thousands of hypothetical faults.

It is the combination of differential comparison and mutation that is key. Differential comparison of tools, as noted above, is not really meaningful, without additional effort; naive methods simply will not work [14]. Consider a comparison of the number of findings 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 findings, it is impossible to be sure. 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 findings 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, since it is hard to construct another plausible explanation for reporting fewer findings on un-mutated code while still detecting more mutants. Our method (see the mutant ratio defined below) provides a quantitative measure of this insight.

Finally, even when 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 helpful context: the difference between the un-mutated code and mutated code will always be small and simple. Moreover, simply looking at how much two tools agree on mutants can answer the question: given that I am using tool A, would adding tool B be likely be worthwhile? Interested users, e.g. security analysts, can inspect the differences to get an idea of the particular cases when a tool might be most effective, but a more typical user can simply look at a Venn diagram of kills like that shown in Figure 1. Consider hypothetical tools A, B, and C. A and B produce similar numbers of findings on the code in question, while tool C produces an order of magnitude more findings. Tool A is likely the most important tool to make use of; it detected more mutants than any other tool, and more than twice as many mutants were killed by A alone than by B alone. However, also running tool B is well justified. B does not do as well as A, but it is the only tool that detects a large number of mutants, and most mutants it detects are unique to it. Finally, tool C may not be worth running, since its poor performance on mutants but high finding rate suggests it may be prone to missing bugs and to false positives. It might be a good idea to just look at the 27 mutants detected by C alone: if they represent an important class of potential problems (perhaps C specialized in detecting potentially non-terminating loops), then C might be useful, but if the first few mutants inspected are false positives, then C is likely not useful.

More concretely, consider the code in Figure 2 [1]. The Universal Mutator tool [27], [28], which has been extensively tuned for Solidity’s grammar (though not to target any particular vulnerabilities), 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 (by Trivial Compiler

![Fig. 1: Mutants killed by three static analysis tools.](image-url)
contract SimpleStorage {
    uint storedData;
    function set(uint x) public {
        storedData = x;
    }
    function get() public view returns (uint) {
        return storedData;
    }
}

Fig. 2: A simple example Solidity smart contract

Equivalence [42]) mutants for it. Both the public version of Slither [19] and SmartCheck [51] (two popular smart contract analysis tools) produce a small number (three and two, respectively) of low-severity, informational, warnings for this code. Both tools also detect four of the seven mutants (here the number of warnings increases, and the additional warnings are clearly driven by the mutation change). 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, deleting it. 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 analysis to findings 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 destroy itself. Given that both tools, ignoring informational results, detect no problems with the original code, and only Slither detects any problems with the mutants, we can say that Slither performs better for this contract.

Comparing mutant results also leads to the idea of improving static analysis tools by examining mutants detected by another tool, and thus known to be in-principle detectable. Improving tools by adding detectors is useful because, even if all tools had the same set of detectors, they would not all report the same bugs; different choices in intermediate language and tradeoffs made to avoid false positives may make the use of multiple tools with similar detectors essential for thorough analysis. And if one tool simply has a superior engine, it is beneficial to users that the “best” tool incorporate all detection rules. However, as with efforts to improve test suites, manually searching through all mutants can be an onerous task, especially for large-scale evaluations. We therefore introduce the idea of prioritizing mutants to make it easier to inspect different weaknesses in tools.

A general objection to our approach is that mutants may differ substantially from “real” faults, in some way. This is certainly true, in a sense [23], but for static analysis purposes we believe it does not matter. The real risk is that some mutation operators align with patterns a particular tool identifies, biasing the evaluation in favor of that tool. Such faults may be dis-proportionately present in mutants vs. real code. However, we consider this unlikely. The vast majority of applied mutation operations for all of our experiments were highly generic, and do not plausibly represent a pattern in which some tool might specialize. Code deletions, the most common kind of mutation by far, leave no “trace” for a tool to match against, but only an omission, so cannot be subject to this concern. Changing arithmetic and comparison operators and numeric constants (incrementing, decrementing, or changing to 0 or 1) account for most of the non-statement-deletion mutants, and it is difficult to imagine how any tool could unfairly identify these.

This paper offers the following contributions:

• We propose a differential approach to comparing static analysis tools based on the insight that program mutants are easy-to-understand, likely-faulty, program changes.
• We propose a definition of mutant killing for static analysis.
• We introduce a simple scheme for prioritizing mutants that helps users understand and use the results of analysis.
• 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 also provide results for popular Java and Python static analysis tools, further demonstrating our approach and showing strengths and weaknesses of these tools.

While there are limitations to using differential mutation analysis to compare/improve static analysis tools, it scales to basing comparisons on many real software source files and very many “fauxts,” but still offers some of the advantages of having humans establish ground truth.

II. DIFFERENTIAL MUTATION ANALYSIS

The proposed approach is simple in outline:

1) Run each tool on the unmutated source code target(s), and determine the baseline: the number of (non-informational/stylistic) findings produced.
2) Generate mutants of the source code and run each tool on each mutant. Consider a mutant killed if the number of findings for the mutated code is greater than the number for the baseline, un-mutated code.
3) Compute, for each tool, the mutant ratio: the mutation score (killed mutants) divided by (mean) baseline. If it is zero, use a baseline equal to either one or the lowest non-zero baseline for any tool in the comparison set2.
4) (Optional): Discard all mutants not killed by at least one tool and all mutants killed by all tools. What remains allows differential analysis. Examine the remaining mutants in the difference in prioritized order.

The most important step here is the computation of the mutant ratio, which tells us about the ability of a tool to produce findings for mutants, relative to its tendency to produce findings in general. If a tool has a tendency to produce large numbers of findings compared to other tools, and this is

1Slither’s “missing return” detector was only available in the private version of Slither, at the time we performed these experiments.

2This problem seldom arises in practice.
paired with a tendency to detect more mutants as well, then the tool will not be penalized for producing many findings. Assuming that real faults are relatively rare in the original, un-mutated code, the best result and best (highest) mutant ratio will be for a tool that produces comparatively few findings for un-mutated code, but detects a larger portion of mutants than other tools; the worst result will be a tool that produces lots of findings, but detects few mutants. We will actually see some examples of this worst case.

A. Prioritizing Mutants

One goal of our approach is to make it easy for tool developers to examine cases where one tool kills a mutant and another fails to, in order to identify patterns for new detectors or analysis algorithm problems. Dedicated developers may also simply want to scan all mutants their tool does not kill, for the same purpose, analogous to what Groce et al. have proposed for automated verification and testing [24], [25]. Security analysts and other expert users who are not developers may also wish to do this, to better understand tool strengths and weaknesses.

Unfortunately the full list of un-killled mutants, or differentially un-killled mutants, is likely to be both large and highly redundant. In our results below, only one of 9 tools we examined killed fewer than 1,000 mutants it alone detected. Any cross-tool comparison is thus likely to involve hundreds or thousands of mutants.

The problem of identifying unique “faults” (tool weaknesses) in this situation is very similar to the fuzzer taming problem in software testing, as defined by Chen et al. [11]: 

“Given a potentially large collection of test cases, each of which triggers a bug, rank them in such a way that test cases triggering distinct bugs are early in the list.” [11]. Their solution was to use Gonzalez’ Furthest-Point-First [21] (FFP) algorithm to rank test cases so that users can examine very different test cases as possible. An FFP ranking requires a distance metric $d$, and ranks items so that dissimilar items appear earlier. The hypothesis of Chen et al. was that dissimilar tests, by a well-chosen metric, will also fail due to different faults. FFP is a greedy algorithm that proceeds by repeatedly adding the item with the maximum minimum distance to all previously ranked items. Given an initial seed item $r_0$, a set $S$ of items to rank, and a distance metric $d$, FFP computes $r_i$ as $s \in S : \forall s' \in S : min_{j<i}(d(s, r_j)) \geq min_{j<i}(d(s', r_j))$. The condition on $s$ is obviously true when $s = s'$, or when $s' = r_j$ for some $j < i$; the other cases for $s'$ force selection of some max-min-distance $s$.

In order to apply FFP ranking to examining mutants, we implemented a simple, somewhat ad hoc distance metric and FFP ranker. Our metric $d$ is the sum of a set of measurements. First, it adds a similarity ratio based on Levenshtein distance [40] for (1) the changes (Levenshtein edits) from the original source code elements to the two mutants, (2) the two original source code elements changed (in general, lines), and (3) the actual output mutant code. These are weighted with multipliers of 5.0, 0.1, and 0.1, respectively; the type of change (mutation operator, roughly) dominates this part of the distance, because it best describes “what the mutant did”; however, because many mutants will have the same change (e.g., changing + to −), the other ratios also often matter. Our metric also incorporates a measure of the distance in the source code between the locations of two mutants. If the mutants are to different files, this adds 0.5; it also adds 0.025 times the number of source lines separating the two mutants if they are in the same file, but caps the amount added at 0.25.

We do not claim this is an optimal, or even tuned, metric. Devising a better metric is left as future work, we only wish to show that even a hastily-devised and somewhat arbitrary metric provides considerable advantage over wading through an un-ordered list of mutants, and introduce the idea of using FFP for mutants, not just for tests: FFP is useful for failures in general, however discovered.

III. EXPERIMENTAL RESULTS

Our primary experimental results are a set of comparisons of tools using our method, for three languages: Solidity (the most popular language for smart contracts), Java, and Python. We used the Universal Mutator tool for all experiments; for Solidity and Python, we believe the Universal Mutator is simply the best available tool. For Java, PIT [12] is more popular, but does not produce source-level mutants, needed for PMD and for manual inspection of results. Universal Mutator includes a large set of mutation operators, some unconventional (e.g., swapping order of function arguments) but based on real-world bugs; the complete set is described by regular expressions at https://github.com/agrose/universalmutator/tree/master/universalmutator/static. However, most mutants that were detected came from a small set of commonly-used operators [3], particularly 1) code deletion and 2) operator, conditional, and constant replacements.

We used our results to answer a set of research questions:

• **RQ1:** Does mutation analysis of static analysis tools produce actionable results? That is, do raw mutation kills serve to distinguish tools from each other, or are all tools similar?

• **RQ2:** Does our approach provide additional information beyond simply counting findings for the original, un-mutated analyzed code? Do ratios differ between tools?

• **RQ3:** Do the rankings that raw kills and ratios establish agree with other sources of information about the effectiveness of the evaluated tools?

• **RQ4:** Do tools detect more mutants in programs for which they produce no warnings, initially?

• **RQ5:** Are mutants distinguishing tools usually flagged due to real faults, where the finding is related to the introduced fault; that is, are our results usually meaningful?

• **RQ6:** Do individual mutants, prioritized for ease of examination, allow us to identify classes of faults that different tools are good at/bad at, and use this information to improve tools? How does this compare to using mutants that have not been prioritized?

In particular, we consider **RQ2** to be of critical importance; if the mutant ratios for tools differ, then this is clear evidence...
that our hypothesis that the tendency of mutants to be faults, and to expect that mutated code will, by a more precise and accurate tool, be flagged as problematic more often than non-mutated code, holds. This expectation that (some subset of the) mutants can serve as proxies for real, detectable faults is the core concept of our approach. **RQ4** addresses a concern briefly mentioned in the introduction: it is possible that warnings for the original code interfere with our definition of detection. The ideal case for our approach is when a tool reports no findings for un-mutated code, and reports a finding when the mutant is introduced. Chekam et al. showed that the “clean program assumption” for testing is a threat to the validity of investigations of the relationship between coverage and fault detection [10], but we show that this is unlikely to be the case for our approach. Our answer to **RQ5** is somewhat inherently qualitative and incomplete; we cannot analyze all mutants on which results are based manually, and understanding the mutants and tool warnings completely would require deep understanding of all the subject programs. However, in many cases, the impact of a mutant is clear, and the reason for warnings is obvious. This was often enough the case that, as we discuss below, we are confident mutants that distinguish tools are meaningful (missed) opportunities for static analysis tools. For **RQ6** we have only a preliminary answer.

### A. Solidity Smart Contract Tools

1) **Smart Contracts and Smart Contract Static Analysis:** Smart contracts are autonomous code instruments, usually operating on a blockchain, that often have critical responsibilities such as facilitating and verifying (large) financial services transactions, tracking high-value physical goods or intellectual property, or even controlling “decentralized organizations” with multifarious aspects. Security and correctness are thus critical in the smart contract domain, and static analysis is a key way to ensure allocation of high-value resources is not compromised. The most popular smart contract platform, by far, is the Ethereum blockchain, and the Solidity smart contract language [8], [55]; the Ethereum cryptocurrency has a market capitalization as we write of over $100 billion dollars, largely fueled by interest in the smart contract functionality. Ethereum contracts have been the targets of widely publicized attacks, with large financial consequences [50], [46]. A recent paper examining results from 23 professional security audits of Solidity contracts argues that effective static analysis is a major key to avoiding such disasters in the future [26].

2) **Static Analysis Tools Compared:** We analyzed three well-known tools for static analysis of Solidity smart contracts: Slither [19], SmartCheck [51], and Securify [54]. Slither, based on an SSA-based intermediate language (SolidityIR [19]) is an open-source tool from Trail of Bits. SmartDec, developed by SmartDec, translates Solidity source directly to an XML-based representation, then uses XPath patterns to define problems. Securify, from SRI Systems Lab at ETH Zurich, works at the bytecode level, first parsing and decompiling contracts, then translating to semantic facts in order to look for problems.

3) **Smart Contract Selection:** We could have used a set of high-transaction contracts, or known-important contracts to validate our approach. However, we knew that one of our goals in the Solidity experiments was to actually improve a mutation analysis tool, and the developers of the static analysis tools use exactly such benchmarks to validate their tools. Basing our improvements on mutants of the contracts used for evaluation of proposed detectors would introduce a serious bias in our favor: we would be more likely to produce detectors that would have true positives and few false positives on the benchmark contracts. We therefore instead selected 100 random contracts for which EtherScan (https://etherscan.io/) has source code, and used this (quite arbitrary) set of contracts from the actual blockchain to compare tools and identify opportunities for improvement. The collected contracts had a total of 15,980 non-comment source lines, as measured by cloc, with a mean size of 159.8 LOC and a median size of 108 LOC. The largest single contract had 1,127 lines of code. The Universal Mutator generated 46,769 valid mutants for these 100 contracts.

4) **Analysis Results:** Figure 3 shows the mutants killed by the Solidity analysis tools. Table I provides numeric details of the results, including the ratio for each tool, adjusting its mutation scores over mutation cases, the impact of a mutant is clear, and the reason for warnings is obvious. This was often enough the case that, as we discuss below, we are confident mutants that distinguish tools are meaningful (missed) opportunities for static analysis tools. For **RQ6** we have only a preliminary answer.

**A. Solidity Smart Contract Tools**

1) **Smart Contracts and Smart Contract Static Analysis:** Smart contracts are autonomous code instruments, usually operating on a blockchain, that often have critical responsibilities such as facilitating and verifying (large) financial services transactions, tracking high-value physical goods or intellectual property, or even controlling “decentralized organizations” with multifarious aspects. Security and correctness are thus critical in the smart contract domain, and static analysis is a key way to ensure allocation of high-value resources is not compromised. The most popular smart contract platform, by far, is the Ethereum blockchain, and the Solidity smart contract language [8], [55]; the Ethereum cryptocurrency has a market capitalization as we write of over $100 billion dollars, largely fueled by interest in the smart contract functionality. Ethereum contracts have been the targets of widely publicized attacks, with large financial consequences [50], [46]. A recent paper examining results from 23 professional security audits of Solidity contracts argues that effective static analysis is a major key to avoiding such disasters in the future [26].

2) **Static Analysis Tools Compared:** We analyzed three well-known tools for static analysis of Solidity smart contracts: Slither [19], SmartCheck [51], and Securify [54]. Slither, based on an SSA-based intermediate language (SolidityIR [19]) is an open-source tool from Trail of Bits. SmartDec, developed by SmartDec, translates Solidity source directly to an XML-based representation, then uses XPath patterns to define problems. Securify, from SRI Systems Lab at ETH Zurich, works at the bytecode level, first parsing and decompiling contracts, then translating to semantic facts in order to look for problems.

Fig. 3: Mutants killed by Solidity static analysis tools.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Findings</th>
<th>Mutation Score</th>
<th>Mutant Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Median</td>
<td>Median</td>
</tr>
<tr>
<td>Slither</td>
<td>2.37</td>
<td>1.0</td>
<td>0.09</td>
</tr>
<tr>
<td>Clean (39)</td>
<td>-</td>
<td>-</td>
<td>0.11</td>
</tr>
<tr>
<td>SmartCheck</td>
<td>1.89</td>
<td>1.0</td>
<td>0.05</td>
</tr>
<tr>
<td>Clean (27)</td>
<td>-</td>
<td>-</td>
<td>0.03</td>
</tr>
<tr>
<td>Security</td>
<td>24.65</td>
<td>17.0</td>
<td>0.03</td>
</tr>
<tr>
<td>Clean (5)</td>
<td>-</td>
<td>-</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**TABLE I: Solidity tool results over all contracts.**

We could have used a set of high-transaction contracts, or known-important contracts to validate our approach. However, we knew that one of our goals in the Solidity experiments was to actually improve a mutation analysis tool, and the developers of the static analysis tools use exactly such benchmarks to validate their tools. Basing our improvements on mutants of the contracts used for evaluation of proposed detectors would introduce a serious bias in our favor: we would be more likely to produce detectors that would have true positives and few false positives on the benchmark contracts. We therefore instead selected 100 random contracts for which EtherScan has source code, and used this (quite arbitrary) set of contracts from the actual blockchain to compare tools and identify opportunities for improvement. The collected contracts had a total of 15,980 non-comment source lines, as measured by cloc, with a mean size of 159.8 LOC and a median size of 108 LOC. The largest single contract had 1,127 lines of code. The Universal Mutator generated 46,769 valid mutants for these 100 contracts.

3) **Smart Contract Selection:** We could have used a set of high-transaction contracts, or known-important contracts to validate our approach. However, we knew that one of our goals in the Solidity experiments was to actually improve a mutation analysis tool, and the developers of the static analysis tools use exactly such benchmarks to validate their tools. Basing our improvements on mutants of the contracts used for evaluation of proposed detectors would introduce a serious bias in our favor: we would be more likely to produce detectors that would have true positives and few false positives on the benchmark contracts. We therefore instead selected 100 random contracts for which EtherScan has source code, and used this (quite arbitrary) set of contracts from the actual blockchain to compare tools and identify opportunities for improvement. The collected contracts had a total of 15,980 non-comment source lines, as measured by cloc, with a mean size of 159.8 LOC and a median size of 108 LOC. The largest single contract had 1,127 lines of code. The Universal Mutator generated 46,769 valid mutants for these 100 contracts.

4) **Analysis Results:** Figure 3 shows the mutants killed by the Solidity analysis tools. Table I provides numeric details of the results, including the ratio for each tool, adjusting its mutation scores over its’ general tendency to produce findings. The second row for each tool shows the number of contracts for which it reported no findings, and the mutation scores over those contracts, only. A user examining these results would suspect that Slither and SmartCheck are both useful tools, and should likely both be applied in a high-risk security-sensitive context like smart contract development. A user might also suspect that the large number of findings produced, and smaller number of mutants killed, for Securify, means that whether to apply Securify is a more difficult decision. On the one hand, Securify does detect nearly as many mutants it alone can identify as SmartCheck. The large number of findings, and very bad mutant ratio, however, lead us to suspect that many of these “detected” mutants are false positives (or, at
least, that the problem is not the one Securify identifies. Extracting the signal from Securify’s noise will be difficult. We also note that while running Slither and SmartCheck on all 46,769 valid mutants was relatively quick (it took about 6 days sequential compute time for Slither and 3 days for SmartCheck, i.e., about 5-15 seconds per mutant for both tools), Securify often required many hours to analyze a mutant, and frequently required a few days to analyze a mutant; the full analysis required over three months of compute time. However, similar statistical results would almost certainly be produced by running as few as 1,000 mutants, randomly selected, due to the implications of Tchebysheff’s inequality [22], a technique that should improve scalability for all such analyses.

If we consider results at the individual contract level, the overall picture is in some ways even clearer. Slither detected more mutants than Securify for 84 of the contracts, and more than SmartCheck for 85 of the contracts. SmartCheck detected more mutants than Securify for 83 of the contracts. Comparing ratios instead, Slither was better than Security for 97 contracts, but better than SmartCheck for only 75 of the contracts. SmartCheck’s ratio was better than Security’s for 95 contracts. The standard deviation of contract raw scores was 0.05 for Slither, 0.03 for SmartCheck, and 0.04 for Security.

For our research questions, RQ1 is clearly answered in the affirmative. Figure 3 shows that the tools address quite different problems, with all tools reporting far more uniquely detected mutants than mutants in common with other tools. There are only 18 mutants detected by all tools, all of them involving replacement of msg.sender (the caller of a smart contract, which may be another smart contract) with tx.origin (the original initiator of a sequence of blockchain calls, a “human” account). Use of tx.origin is often (though not always) a bad idea, and can lead to incorrect behavior, so it is not surprising tools all recognize some misuses of it.

RQ2 is also answered in the affirmative. Counting findings for un-mutated code might suggest that Security is the best tool, by a wide margin, but in the context of its near-zero mutant ratio, we must suspect (and we partially manually confirmed) that many of the warnings are false positives. Slither has the best mutant ratio, but the margin between it and SmartCheck confirms that both tools likely provide value; we note there is a 25% chance that SmartCheck has a better ratio than Slither for an individual contract.

For RQ3, there are only a few tool comparisons in the literature; this is probably due to the fast-moving nature of the blockchain analysis world; the oldest of these tools’ publication dates is 2018. The most extensive is that of Durieux et al. [18], though it unfortunately was unable to provide anything other than an implicit look at false positives, somewhat limiting its practicality. Slither detected 17% of known vulnerabilities in their analysis, vs. 11% for SmartCheck and 9% for Security. Slither and SmartCheck were also among the four (out of 9) tools that detected vulnerabilities in the most categories; Security was not. The overall recommendation of Durieux et al. was to use a combination of Slither and Mythril [13] for contract analysis. Parizi et al. [43] also offer a ranking of tools, and determined that SmartCheck was the most effective, and far more so than Security; unfortunately, they did not include Slither in their set of evaluated tools.

The Slither paper [19] also provides an evaluation of all three tools. Their findings counts differ from ours because of different choices (we threw out merely informational results), but these are unrelated to mutation analysis, in any case. The evaluation only considered reentrancy faults [5], [26] (which are sometimes, but only rarely, introduced by mutants). For reentrancy, Slither performed best on two real-world large contracts, finding subtle bugs in both, SmartCheck detected the problem in one of the two, and Security detected neither. For a set of 1,000 contracts, SmartCheck had a high false positive rate (over 70%) but detected more actual reentrancies (209) than Slither (99) or Security (6). On the other hand, Slither’s low false positive rate of 11% makes its results possibly more useful in practice.

For RQ4, on the changes seen when restricting analysis to clean contracts, Slither did slightly better at detecting mutants when the original contract was clean for Slither, and the other two tools did somewhat worse on contracts for which they reported no findings. For the three contracts clean for all tools, Slither performed almost exactly as it did over contracts in general, and the other tools performed worse, by about the same margin as they did for their own clean contracts. For our approach, we only need a weak version of the “clean program assumption”: the threat is that kills may be under-reported for non-clean programs, due to interference with findings for the original code. It is not a problem if mutation scores are worse for programs where a tool reports no findings for the un-mutated code. We therefore, for smart contracts, find no threat to our approach arising from the presence of findings on un-mutated code. We speculate that “clean” results for some tools result from contracts where the tool has trouble with the contract code, but does not actually crash; Slither may do better on clean code because it has fewer such failures, and clean contracts are probably somewhat easier to analyze.

Following the method proposed in Section II, for RQ5 we focused on examining mutants detected by at least one tool, but not detected by all tools, the only ones that actually influence the comparison of tools. The vast majority of the mutants in this set were meaningful semantic changes a static analysis tool could be expected to detect, and the findings produced by tools were relevant to the nature of the fault. We do not believe that all mutants represent definite faults; some are harmless but unusual code changes. Many cases where use of tx.origin in place of msg.sender was flagged seem to us to be strange, but not necessarily incorrect, code. On the other hand, it is not at all unreasonable for tools to report such notably strange code. Our estimate is that, ignoring tx.origin cases, at least 70% of the mutants detected by one, but not all, tools, represent realistic bugs, and failure to detect is roughly equally due to missing detectors and imprecise analysis.

Because our random contracts’ quality might be low, we also checked our results on 30 contracts from the Solidity doc-
Examining only a few mutants, we identified three:

1) **Boolean constant misuse:** This detector flags code like

   ```solidity
   if (true) 
   ```

   or

   ```solidity
   g(b || true) 
   ```

   (where `g` is a function that takes a Boolean input). Constant-valued conditionals tend to indicate debugging efforts that have persisted into production code, or other faults; there are almost no circumstances where a conditional should not vary with state or input. This detector is actually split into two detectors, one for this serious issue, and an informational/stylistic detector that flags code such as `if (x == true)`, which is merely difficult to read.

2) **Type-based tautologies:** A type-based tautology is again a case where a Boolean expression has a constant value, but this is not due to misuse of a Boolean constant, but is instead due to the types in a comparison. For example, if `x` is an unsigned integer type, the comparison `x >= 0` is always true and `x < 0` is always false. This detector is a generalization of the SmartCheck detector https://github.com/crytic/slither/blob/master/rule\_descriptions/SOLIDITY\_UINT\_CANT\_BE\_NEGATIVE/, modified to actually compute the ranges of types and identify other cases such as `y < 512` where `y`’s type is `int8`.

3) **Loss of precision:** Solidity only supports integer types, so performing division before multiplication can introduce avoidable rounding. This is a fairly important problem, given Solidity code often performs critical financial calculations.

   ```solidity
   require(nextDiscountTTMTokenId6 >= 361 \&\& ...);
   ```

   or

   ```solidity
   require(nextDiscountTTMTokenId6 >= 0 \&\& ...);
   ```

   This detector has, to date, detected two real-world exploitable vulnerabilities: https://github.com/crytic/slither/blob/master/contracts/SOLIDITY\_DIV\_MUL/.

   All three of these detectors were submitted as PRs, vetted over an internal benchmark set of contracts used by the Slither developers to evaluate new detectors, and accepted for release in the public version of Slither. All three detectors produce some true positives (actual problems, though not always exploitable) in benchmark contracts, have acceptably low false positive rates, and were deemed valuable enough to include as non-informational (medium severity) detectors. Moreover, the loss of precision detector has, to date, detected two real-world security vulnerabilities: https://github.com/crytic/slither/blob/master/trophies.md — search for “Dangerous divide before multiply operations.”

**Fig. 4:** Examples of mutants leading to new detectors.

```solidity
if (true) 
```
to the tool. Our three submitted detectors were not present as private detectors, and only one (the type-based tautology detector) had even been identified, via a GitHub issue, as a potential improvement (and only in the private version of Slither). Combining statement deletion mutants with other mutants only moved the mutants we used down to 3rd, 11th, and 14th positions. By default we rank statement deletions separately, since such mutants are usually easier to understand and evaluate, and in testing (but not static analysis) they are likely to be the most critical faults not detected.

Examining the first 100 mutants in the unprioritized lists for SmartCheck and Securify, ordered by contract ID and mutant number (roughly source line mutated) we were unable to identify any obviously interesting mutants, suggesting that it is indeed hard to use mutation analysis results without prioritization. A large majority of the mutants we inspected involved either the missing return problem noted in the introduction, or replacing msg.sender with tx.origin; Slither has a detector for misuses of tx.origin. SmartCheck and Securify tend to identify most (though not all) uses of tx.origin as incorrect, while Slither has a more selective rule, intended to reduce false positives. It is hard to scale our efforts here to a larger experiment, since writing and submitting changes to static analysis tools is always going to be a fairly onerous task, but we believe that our successful addition of new detectors, and the ease of identifying candidate detectors using mutant prioritization supports a limited affirmative answer to RQ6.

B. Java Tools


2) Project Selection: For Java and Python, we did not have to worry about invalidating tool improvements by basing our results on benchmark code. We therefore aimed to use realistic, important source code. We selected top GitHub projects (defined by number of stars) for each language, and removed projects with fewer than 5 developers or less than six months of commit history (as well as projects that did not build). For Java, we analyzed the top 15 projects satisfying our criteria, with a maximum of 623,355 LOC and a minimum of 3,957 LOC, and a total size of 1.8 million LOC. Because the Universal Mutator does not "know" Java syntax, and Java is very verbose, the Java compiler rejected a large number of the generated mutants (e.g., deleting declarations). We still, due to the huge size of the source files and thus number of mutants (and time to compile full projects), restricted our analysis to files where Universal Mutator’s implementation of TCE [42] for Java was useful, i.e. individual files that could be compiled and the bytecode compared, leaving us with just over 70,000 mutants, ranging from 136 to 10,016 per project.

### TABLE II: Java tool results over all projects.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Findings Mean</th>
<th>Findings Median</th>
<th>Mutation Score Mean</th>
<th>Mutation Score Median</th>
<th>Mutant Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpotBugs Clean (3)</td>
<td>28.93</td>
<td>14.00</td>
<td>0.07</td>
<td>0.07</td>
<td>0.0002</td>
</tr>
<tr>
<td>PMD Clean (0)</td>
<td>53.73</td>
<td>32.00</td>
<td>0.07</td>
<td>0.07</td>
<td>0.0001</td>
</tr>
<tr>
<td>Infer Clean (6)</td>
<td>11.60</td>
<td>3.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

3) Analysis Results: Figure 5 shows the mutants killed by the Java analysis tools, and Table II provides numeric results for projects and clean projects, respectively. At the individual project level, Infer was never best; PMD had a better raw score than SpotBugs for 10/15 projects, but SpotBug had a better ratio for 11. There is likely a tradeoff between verbosity and precision. Standard deviation in project scores was 0.05 for SpotBugs, 0.04 for PMD, and 0.0 for Infer.

In terms of RQ1, the raw kills results suggest there is considerable value in running both SpotBugs and PMD. Both produce a large number of unique detections, though PMD produces about 20% more than SpotBugs. Infer on the other hand, is only able to detect two mutants, but these are unique; both were, however, spurious concurrency warnings. It may be that the diff sizes in our code were simply too small for Infer’s approach. RQ2 is also answered in the affirmative. While the raw kills for SpotBugs are not as good as for PMD, it had fewer findings, giving it a mutant ratio approximately twice that of PMD.

We note that SpotBugs crashed for many more Java programs than PMD and Infer (neither crashed for any original file in our experiments). SpotBugs “failed to detect” 23,000 of the mutants because it did not process the un-mutated file for 383 files, over 12 of the 15 projects—just over 23% of the 1,664 total files. Removing files for which SpotBugs failed, however, did not dramatically change results; SpotBugs’ mean mutation score rose to 0.09, and mutant ratio rose to 0.003, but PMD’s mean score rose to 0.10, and mutant ratio to 0.002.

For RQ3, to our knowledge there is no academic comparison of all three tools; one study dates from 2004 [47], used...
FindBugs, not SpotBugs, and reached no strong conclusions with respect to FindBugs vs. PMD; a more recent study found that SpotBugs outperformed Infer for Defects4J [35] bugs [32], but did not compare to PMD. However, the user postings listed in Figure 6, plus personal communications with security analysts who use these tools [20] supported some basic conclusions. SpotBugs is perhaps the best tool for finding bugs; PMD focuses more on stylistic issues and has a weaker semantic model. Running both is definitely recommended, as neither is extremely effective. Infer is closer to a model checker focusing on resource leaks than a truly general-purpose tool, arguably. In fact, we suspect Infer would perform much better if we used mutation operators targeting some important subtle Java bugs; Infer is probably not a good general-purpose static analysis tool for Java. Note that we did not use Infer’s experimental detectors. For RQ4 there were very few clean projects, but we see no evidence that non-clean code is a source of degradation in mutation detection.

For Java, again, the large majority (> 75%) of randomly chosen mutants in the tool difference set we inspected for RQ5 were definitely meaningful, essentially “real faults.” In particular, for Java, the large majority of mutants involved either deleted method calls or changes to conditionals (e.g., \(== \text{null}\) to \(!= \text{null}\)) that would clearly introduce potential null pointer exceptions (NPEs), and such a potential NPE was the produced finding. While the differences between Solidity tools were often due to different detectors, the Java differences seemed mostly rooted in analysis engine methods; all tools aim to warn about potential NPEs. Because the number of mutants we could examine and understand was smaller, we are less confident in making a probabilistic estimate than with Solidity, but it was clear the basis of the comparison was primarily realistic faults, which some tools detected and others did not.

C. Python Tools

1) Static Analysis Tools Compared: We compared three widely used and well-known Python tools: Pylint https://www.pylint.org/ (probably the most widely used of Python bug finding tools), pyflakes https://pypi.org/project/pyflakes/, designed to be faster, lighter-weight, and more focused on bugs (without configuration) than Pylint, and PyChecker http://pychecker.sourceforge.net/, an older, but still used tool.

2) Project Selection: For Python, we analyzed the top 25 GitHub projects by our criteria (see above), due to the smaller size of Python projects. These ranged in size from 137 LOC to 29,339 LOC, with a total size of about 75 KLOC, and a mean size of 3,185 LOC. We analyzed 158,418 valid, non-TCE redundant, mutants taken from these programs.

3) Analysis Results: Figure 7 shows the mutants killed by the Python analysis tools, and Table III provides numeric results for projects and clean projects, respectively. At the project level, Pylint was better than pyflakes and PyChecker for 21 and 24 projects, respectively, by raw score; pyflakes was better than Pylint and PyChecker for 3 and 24 projects, respectively; PyChecker was never better than another tool by raw score. Switching to ratio measures, Pylint was better than pyflakes and PyChecker for 11 and 23 projects, respectively; pyflakes was better than Pylint and PyChecker for 13 and 24 projects, respectively. PyChecker was better than Pylint for one project, by ratio. Standard deviations in mutation scores were sometimes high for Python: 0.17 for Pylint, 0.08 for pyflakes, and 0.01 for PyChecker.

For RQ1, there is a clear difference between tools. Pylint uniquely kills almost an order of magnitude more mutants than the next-best tool. There is a large overlap between Pylint and pyflakes, while PyChecker is both much less effective and doing something fairly different than the other tools. From the diagram, one might think that pyflakes acts, to some extent, as a less verbose “subset” of Pylint, in that most mutants detected by pyflakes are also detected by Pylint (however, the nearly 3,500 killed mutants unique to pyflakes suggest it is a useful tool, perhaps most useful after problems also reported by Pylint are fixed). PyChecker performs poorly in part because it crashed (due to changes in Python since the last update to the tool) for 19 of the 25 projects; however, it also performed poorly on programs where it worked.

The mutant ratios RQ2 show that pyflakes is more competitive than is obvious from raw kill comparisons. The mutant ratio is almost three times as good as for Pylint! That is, some of Pylint’s advantage may be due to general verbosity, even once stylistic warnings are turned off. Combining the results from raw kills and ratios, a strategy of using pyflakes as a quick check for problems, then using both pyflakes and Pylint for more in-depth analysis makes sense. Whether to use both pyflakes and Pylint in CI is a question of tolerance for handling false positives, but it is clear that the “price” of additional warnings from Pylint is (1) high but (2) not without ereturn on investment (in terms of additional real finds). PyChecker is probably too outdated, and sometimes too verbose when it works, to be useful.

For RQ3, there were again no academic comparisons we could find. However, opinions on the web were quite common (see Figure 8, which lists ones we examined). It is hard to sum-

<table>
<thead>
<tr>
<th>Tool</th>
<th>Findings Mean</th>
<th>Median</th>
<th>Mutation Score Mean</th>
<th>Median</th>
<th>Mutant Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pylint</td>
<td>10.63</td>
<td>5.50</td>
<td>0.31</td>
<td>0.34</td>
<td>0.03</td>
</tr>
<tr>
<td>Clean (4)</td>
<td>-</td>
<td></td>
<td>0.49</td>
<td>0.47</td>
<td>-</td>
</tr>
<tr>
<td>pyflakes</td>
<td>1.46</td>
<td>0.00</td>
<td>0.14</td>
<td>0.12</td>
<td>0.09</td>
</tr>
<tr>
<td>Clean (4)</td>
<td>-</td>
<td></td>
<td>0.14</td>
<td>0.12</td>
<td>-</td>
</tr>
<tr>
<td>PyChecker</td>
<td>11.6</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Clean (4)</td>
<td>-</td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>-</td>
</tr>
</tbody>
</table>

Fig. 7: Mutants killed by Python static analysis tools.
Fig. 8: Discussions of Python static analysis tools.

marize the overall opinion here, since it ranges considerably. There is probably general agreement that PyChecker is old and maybe less useful, but also terse and sometimes helpful. Pylint is the most recommended tool, and the general complaint that it is too picky was somewhat mitigated in our results by turning off warnings that are clearly purely stylistic. Pyflakes is also well liked, and is considered much less verbose than Pylint; this was definitely reflected in our results, where Pyflakes underperformed in raw kills, but had a much better mutant ratio than Pylint. For RQ4, Pylint performed significantly better on clean projects. Performance for the other two tools was essentially unchanged.

For Python, all but three of the mutants we examined for RQ 5 (a random sample of 100 mutants with a kill difference) involved code changes we agreed were definitely buggy, and would cause incorrect behavior if executed (the exceptions were deletions of code with no effect on state, e.g., strings as comments). The large majority involved statement deletions, detected via 1) unused variables/arguments, 2) instances lacking a member field, or 3) undefined variables. Interestingly, this seems to be an engine issue more than a detector issue, as pyflakes and Pylint both basically support all of these kinds of checks. In some cases the two tools both detected a problem (but PyChecker did not) but differed as to which variable was not used or defined, again suggesting an engine rather than detection rule difference. Pylint’s better performance was mostly, in the sample, due to detecting more of these issues, though it also was the only tool in the sample that detected arithmetic operation changes, due as far as we could determine to constant index changes. Pylint also detected a few mutants no other tool did, due to the presence of unreachable code. In one case, it also noticed a protected member access via a change in constant index, a surprisingly complex problem to detect assertions, rather than the much more common case of tools that identify bad code patterns.

There is no room in the paper to present the exact set of projects analyzed, but we have provided an (anonymized) github repository containing raw results for inspection by reviewers, or further analysis by other researchers (https://github.com/mutantsforstaticanalysis/rawdata).

IV. RELATED WORK

The goal of “analysing the program analyser” [9] is intuitively attractive. 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 such tools makes this a challenge similar to testing machine learning algorithms [30]; most tools will not produce “the right answer” all the time, as a result of both algorithmic constraints and basic engineering trade-offs. While comparisons of static analysis tools [47], [18], [43], [19], [44] have appeared in the literature for years, these generally involved large human effort and resulting smaller scale, did not make a strong effort to address false positives, or restricted analysis to, e.g., a known defects set [32], [17]. Defect sets are vulnerable to tools intentionally overfitting/gaming the benchmark; our approach makes it easy to compare tools on “fresh” code to avoid this risk. Compared to well-known studies of Java tools [32], [47] our approach used a larger set of subject programs (1.8 MLOC total vs. 170-350KLOC) and thousands of detected faults, vs. e.g., about 500 known defects [32]. Results not using defect sets are even more limited in that humans can only realistically examine a few dozens of each type of warning [47].

Cuq et al. [14] proposed the generation of random programs (à la Csmith [56]) 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 based on the observation that the ability to detect program mutants offers an automatable comparison basis. We essentially adopt the approach of the large body of work on using mutants in software testing [34], [15], [7], [24], [25], [45], [33], [3], [2], but re-define killing a mutation for a static analysis context.

Klinger et al. propose a different approach to differential testing of analysis tools [36]. 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 seeds. The primary differences are that their seed programs must be warning-free (which greatly limits the set of input programs available), and that the new versions are based on adding new assertions only. We allow arbitrarily buggy seed programs, and can, due to the any-language nature of the mutation generator we use, operate even in new languages. 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). Most importantly, however, their approach only applies to tools that check assertions, rather than the much more common case of tools that identify bad code patterns.
V. Conclusions and Future Work

In this paper, we showed that program mutants can be used as a proxy for real faults, to compare (and motivate improvements to) static analysis tools. Mutants are attractive in that a large body of work supports the claim that at least 60-70% of mutants are fault-inducing. This allows us to assume detected mutants are faulty, and escape the ground-truth/false positive problem that makes comparing static analysis tools so labor-intensive. We evaluated 9 popular static analysis tools, for Solidity smart contracts, Java, and Python, and offer advice to users of these tools. Our mutation results strongly confirm the wisdom of using multiple tools; with the exception of one Java tool, all tools we investigated uniquely detected over 1,000 mutants, and for Java and Python there were no mutants detected by all tools. For Solidity, academic research evaluations of the tools generally agreed strongly with our conclusions, but lacked the detail mutant analysis contributed.

We were also able to use our methods, plus a novel mutant prioritization scheme, to identify and implement three useful new detectors for the open source Slither smart contract analyzer, the best-performing of the tools.

As future work, we would like to further validate our approach and improve our admittedly ad hoc mutant distance metric. Allowing user feedback [37], [30], or applying metric learning methods [38] (particularly unsupervised learning [48], [52]) are the most obvious and interesting possibilities.

Acknowledgements: A portion of this work was funded by the National Science Foundation under CCF-2129446.

References


[26] Alex Groce, Josselin Feist, Gustavo Greco, and Michael Colburn. What are the actual flaws in important smart contracts (and how can we find them)? In International Conference on Financial Cryptography and Data Security, 2020. Accepted for publication.


