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 without executing it. Understanding the strengths and weaknesses of tools, and performing direct comparisons of their effectiveness is difficult; 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 effective new detectors as an open source contribution 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, 11 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, and when security is a concern, applying effective static analysis tools is essential. Static analysis can find problems that are extremely hard to detect by testing, e.g. 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 like AFL days to find may be immediately identified by a good static analysis tool.

Users of static analysis tools often wonder which of multiple tools available for a programming language are most effective, and, when using more than one tool is an option (e.g. with free tools), how much tools overlap in their results. Given the human effort required to read static analysis results, the latter can be an important question. If two tools find substantially different (non-false-positive) bugs, it is wise to use both if finding all bugs is important. On the other hand, if two tools are very similar in what they detect, the effort of wading through duplicate results may not be a good use of time. Developers of static analysis tools also want to be able to compare their tools to others targeting the same domain, in order to see what detection methods (or tweaks to precision/soundness trade-offs) they might want to imitate. Unfortunately, comparing static analysis tools 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 [24, 36] 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 [12].

Mutation testing [6, 13, 30] (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 [19] paper and a 2018 journal extension [20] 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, allowing it to detect multiple new bugs. Recently, mutation analysis has been adopted in industrial settings, though not for actual examination of all surviving mutants [29, 39], a practice that is hard to scale to large code bodies.

Combining a differential approach (not differential *testing*, precisely, in that individual differences are not always worth inspecting) and mutation analysis, however, offers a novel way to compare static analysis tools, one useful to users wishing to select a good tool or set of tools, researchers interested in, e.g., the impact of precision/soundness trade-offs or different intermediate languages, and developers of static analysis tools hoping to improve their tools by systematic examination of code changes that are likely fault-inducing but not detected.

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

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

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 usually at minimum 40%, often 50-60%, and frequently up to 80%+ [1, 20, 34], of mutants are not semantically equivalent to the original code, and are actually fault-inducing and detectable by some test or property [26, 37, 43], 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. In other words, a key notion is that while most mutants cannot be detected statically, the ones that are tend to be *true positives*: if they were real code changes, they would be faults.

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.

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; naïve methods simply will not work [12]. 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, 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 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. While it is not proof of this claim, it is hard to construct another plausible explanation for reporting *fewer* findings on un-mutated code while

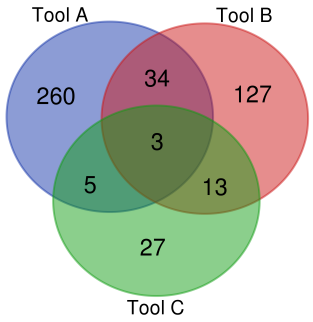

Figure 1: Mutants killed by three static analysis tools.

still detecting *more* mutants. Other than having better precision and recall, how else could a tool effectively distinguish mutated from un-mutated code?

We can quantitatively express the relationship between detecting mutants and findings for the original program. Since we are comparing over the same program or set of programs, we simply divide the mean *mutation score* ($\frac{|killed|}{|mutants|}$, the ratio of killed mutants to all mutants) by the mean number of findings. This *mutant ratio* tells us about the ability of a tool to produce findings *for mutants*, relative to its tendency to produce findings in general (per line of code, per source file, etc.). If a tool has a tendency to produce large numbers of findings (compared to other tools), and this is paired with a tendency to detect most mutants, 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 ratio score) 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 the worst case in our results for real tools.

Finally, even when tools have similar quantitative results, including similar ratios, 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. Moreover, simply looking at how much two tools agree on mutants can answer the question of a user of static analysis tools: given that I am using tool A, would adding tool B be likely to add enough new, interesting results to make it worth my time to examine its output? When (rarely) one tool subsumes another in terms of mutants, it can be very clear that the tool whose killed mutants are all killed by another tool is likely strictly inferior. 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. Assume that tools A, B, and C all produce very similar numbers of findings for un-mutated code, and have similar execution times. 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 probably a very good

idea, assuming, e.g., it is not a very expensive commercial tool. 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; recall that it produces a similar number of findings to A and B on the un-mutated code, so it is notably bad at detecting faults (at least ones that look like mutants). 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.

Comparing mutant results also 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 [17] algorithm and distance metric inspired by previous work on helping developers sort through failing test cases to find distinct faults [10, 28], to help static analysis tool developers find interesting patterns without wading through numerous uninteresting mutants not identified for similar reasons.

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 [18], but for static analysis purposes 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. Statement deletions leave no "trace" for a tool to match against, but only an omission. 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. We cannot see how any tool could have a (useful) detection rule that looks for these kinds of changes (a tool obviously would not flag all instances of 0 and 1 constants, or of addition, and remain a useful tool). Finally, many mutants (though many fewer than in these categories) add `break` or `continue` statements. Again, we do not see how a tool could take advantage of this without either genuinely understanding likely-bad control flow patterns (in which case, we think it "deserves" to kill more mutants) or flagging so many legitimate uses of `break` and `continue` in un-mutated code that it would have a bad mutant ratio, due to producing more warnings than other tools.

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

1.2 A Simple Example

Consider the code in Figure 2, from the Solidity 0.4.24 "Introduction to Smart Contracts". The Universal Mutator tool [22, 23], 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 Equivalence [37]) mutants for this trivial example code. Both the public version of Slither [16] and SmartCheck [45] 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 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 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 mutants saves little effort), so we will show the utility of that idea in our full Solidity results.

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 simply compare, 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.

- 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.

The Solidity, Java, and Python case studies also serve to answer a set of research questions investigating the utility of our approach, and provide basic evidence that it provides actionable, non-obvious information that is otherwise difficult to produce. While there are certainly limitations to using differential mutation analysis to compare (and extend) static analysis tools, the method scales to basing comparisons on large numbers of real software source files, but has some of the advantages of humans establishing ground truth for tool findings.

2 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. Dedicated developers may also simple 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 [19, 20]. Security analysts and other expert users who are not developers may also wish to do this, to better understand details of tool strengths and weaknesses.

Unfortunately the full list of unkilled mutants, or differentially unkilled 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. [10]: "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." [10]. Their solution was to use Gonzalez' Furthest-Point-First [17] (FPF) algorithm to *rank* test cases so that users can examine very different test cases as quickly as possible. An FPF ranking requires a distance metric d, and ranks items so that dissimilar ones appear earlier. The hypothesis of Chen et al. was that dissimilar tests, by a well-chosen metric, will also fail due to different faults. FPF 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, FPF 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 FPF ranking to examining mutants, we implemented a simple, somewhat *ad hoc* distance metric and FPF ranker in the Universal Mutator [22] tool. Our metric d is the sum of a set of measurements. First, it adds a similarity ratio based on Levenshtein distance [35] 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. The Python `Levenshtein` library's similarity ratio is used, as it is based on true minimal string edits; it reports similarity ratios between 0.0 and 1.0.

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 to the distance; it also adds 0.25 times the number of source lines separating the two mutants if they are in the same file, divided by 10, but caps the amount added at 0.25. The full metric, therefore is:

$$\begin{gathered} 5.0 \times r(edit_1, edit_2) + \\ 0.1 \times r(source_1, source_2) + \\ 0.1 \times r(mutant_1, mutant_2) + \\ 0.5 \times different\_files + \\ max(0.25, \frac{line\_distance(mutant_1, mutant_2)}{10}) \end{gathered}$$

Where r is a Levenshtein-based string similarity ratio, and *line_distance* is the distance in a source file between two locations, in lines (and zero if the locations are in different source files).

We do not claim this is an optimal, or even tuned, metric; in the long run, it even may be completely replaced. One element that will likely remain is the focus on source code, rather than bytecode; static analysis tools generally analyze source, and some even work for code that does not compile. The metric we use provided useful results for a different problem, ranking mutants in an effort similar to that of Groce et al. [19, 20] to examine unkilled mutants for a smart contract library tested using the Echidna fuzzer [47], and we simply adopted it for our differential mutation analysis. 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 FPF for mutants, not just for tests: FPF is useful for failures in general, however discovered.

3 EXPERIMENTAL RESULTS: COMPARISON AND IMPROVEMENT OF STATIC ANALYSIS TOOLS

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 also use these results to answer a set of research questions that consider the utility of our method:

- **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 in terms of mutation detection capability?
- **RQ2:** Does our approach provide additional information beyond simply counting findings for the original, un-mutated analyzed code? Do *ratios* differ between programs, or does

the number of mutants killed simply reflect the "verbosity" of each tool?
- **RQ3:** Do the rankings that raw kills and ratios establish agree with other sources of information about the effectiveness of the evaluated tools? Can we confirm results from other evaluation methods, or informal opinion?
- **RQ4:** Do mutation scores for clean programs differ from those for programs for which tools report findings before mutation?
- **RQ5:** Do individual mutants, prioritized for ease of examination, allow us to identify classes of faults that different tools are good at/bad at? Can we extend this information to improve tools by addressing identified weaknesses?
- **RQ6:** How easy is it identify low-hanging fruit for improving tools without prioritization to remove conceptually redundant mutants?

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 sound and precise 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 report no findings for un-mutated code, and either reports a finding (kills the mutant) or not (does not kill it) when the mutant is introduced. Chekam et al. showed that the "clean program assumption" for testing, the idea that coverage will be similar for faulty and fixed versions of a program, is a threat to the validity of investigations of the relationship between coverage and fault detection [9], and we want to establish that this is unlikely to be the case for our approach. **RQ5** and **RQ6** receive only preliminary answers, for the smart contract tools.

3.1 Solidity Smart Contract Tools

3.1.1 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 [7, 49]; the Ethereum cryptocurrency has a market capitalization as we write of over $15 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 [40, 44]. 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 [21].

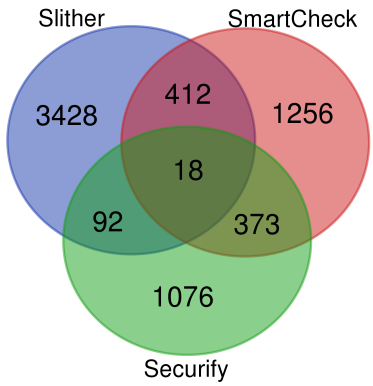

Figure 3: Mutants killed by Solidity static analysis tools.

Tool	Findings Mean	Findings Median	Mutation Score Mean	Mutation Score Median	Mutant Ratio
Slither	2.37	1.0	0.09	0.09	0.038
SmartCheck	1.89	1.0	0.05	0.05	0.026
Securify	24.65	17.0	0.03	0.02	0.001

Table 1: Solidity tool results over all contracts.

3.1.2 Static Analysis Tools Compared. We analyzed three well-known tools for static analysis of Solidity smart contracts: Slither [16], SmartCheck [45], and Securify [48]. Slither, based on an SSA-based intermediate language (SlithIR [16]) is an open-source tool from Trail of Bits. SmartCheck, 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 predefined problems.

3.1.3 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.

3.1.4 Analysis Results. Figure 3 shows the mutants killed by the Solidity analysis tools. Tables 1 and 2 provide numeric details of

Tool	# Clean Contracts	Mutation Score Mean	Mutation Score Median	Clean For All (3) Mutation Score Mean	Clean For All (3) Mutation Score Median
Slither	39	0.11	0.11	0.09	0.09
SmartCheck	27	0.03	0.01	0.03	0.00
Securify	5	0.00	0.00	0.00	0.00

Table 2: Solidity tool clean contract results.

the results, including the *ratio* for each tool, adjusting its mutation scores by its' general tendency to produce findings. First, 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. Second, a user might suspect that the large number of findings produced, and smaller number of mutants killed, for Securify, mean that including Securify in the static analysis tool stable 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.

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 usually a bad idea, can lead to incorrect behavior, and may not be relied on to be meaningful at all in future versions of Ethereum (https://ethereum.stackexchange.com/questions/196/how-do-i-make-my-dapp-serenity-proof).

RQ2 is also answered in the affirmative. Counting findings for un-mutated code might suggest that Securify is the best tool, by a wide margin, but in the context of its near-zero mutant ratio, we must suspect 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.

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. [15]. Slither detected 17% of known vulnerabilities in their analysis, vs. 11% for SmartCheck and 9% for Securify. Slither and SmartCheck were also among the four (out of 9) tools that detected vulnerabilities in the most categories; Securify was not. The overall recommendation of Durieux et al. was to use a combination of Slither and Mythril [11] for contract analysis. Parizi et al. [38] also offer a ranking of tools, and determined that SmartCheck was the most effective, and far more so than Securify; unfortunately, they did not include Slither in their set of evaluated tools.

The Slither paper [16] 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 [4, 21] (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 Securify 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 Securify (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 week 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 generally simpler and easier to analyze.

3.1.5 Improving Slither **(RQ5 and RQ6)**. Based on the differential mutation analysis, we identified three low-hanging fruit to improve the performance of Slither. The process was simple. First, we produced a list of all mutants killed by either SmartCheck or Securify, but not killed by Slither. We then applied the prioritization method based on the FPF algorithm and the distance metric described in Section 2, and examined the mutants in rank order. Many of the mutants were difficult to identify as true or false positives, absent context. Some opportunities for enhancement were clear, but seemed likely to require considerable effort to implement without producing a large number of false positives. For example, Securify often detected when an ERC20 token contract's guard preventing making the special 0x0 address the owner of a contract was removed, and issued the error `Violation for MissingInputValidation`. Detecting such missing guards is probably useful, but formulating a way to do it without producing false positives is non-trivial. We wanted to show that mutants could identify *useful* but *easy to implement* missing detectors. Examining the first few mutants, we

```
0x598ab825d607ace3b00d8714c0a141c7ae2e6822_Vault.mutant.275.sol:
if (!p.recipient.send(p.amount)) {  // Make the payment
==>          if (true) {  // Make the payment
if (true) {  // Make the payment
```

Figure 4: Mutant showing Boolean constant misuse.

```
0x968815CD73647C3af02a740a2438D6f8219e7534_TTPresale.mutant.311.sol:
require(nextDiscountTTMTokenId6 >= 361 && nextDiscountTTMTokenId6 <= 391);
==>  ...361...==>...0...
require(nextDiscountTTMTokenId6 >= 0 && nextDiscountTTMTokenId6 <= 391);
```

Figure 5: Mutant showing Type-based tautologies.

```
0x534ccee849a688581d1b0c65e7ff317ed10c5ed3_NametagToken.mutant.480.sol:
byte char = byte(bytes32(uint(x) * 2 ** (8 * j)));
==>  ...*...==>.../...
byte char = byte(bytes32(uint(x) * 2 ** (8 / j)));
```

Figure 6: Mutant showing Loss of precision.

identified three such, based on mutants killed by either SmartCheck or Securify, or both:

(1) **Boolean constant misuse:** This detector flags code like `if (true)` or `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 notes that code such as `if (x == true)`, while semantically harmless, is difficult to read. This problem was easily identified from cases such as the mutant in Figure 4, killed by SmartCheck but not Slither:

(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/smartdec/smartcheck/blob/master/rule_descriptions/SOLIDITY_UINT_CANT_BE_NEGATIVE/, modified to actually compute the ranges of types and identify more general instances of tautological comparisons, e.g. `y < 512` where `y`'s type is `int8`. Again, the problem was easily identified by mutants killed by SmartCheck but not Slither such as the one in Figure 5.

(3) **Loss of precision:** Solidity only supports integer types. This means that performing division before multiplication can introduce rounding that is not present when the multiplication is performed first. This is a fairly important problem, given the frequency with which Solidity code performs financial calculations where maximum precision is desired. SmartCheck provides a detector for such precision losses https://github.com/smartdec/smartcheck/blob/master/rule_descriptions/SOLIDITY_DIV_MUL/, which enabled it to detect mutants such as the one shown in Figure6.

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. The first mutants in prioritized rank exhibiting the issues, shown above, were the 2nd, 9th, and 12th non-statement-deletion mutants ranked for SmartCheck, out of over 800 such mutants. Using our prioritization, it was possible to identify these issues by examining fewer than 20 unkilled mutants. Without prioritization, on average a developer would have to look at more than 200, 80, and 400 mutants, respectively, to find instances of these problems. Interestingly, the very fact that these instances are "needles in a haystack" among the mutants not killed by Slither means that the results in Figure 3 and Tables 1 and 2 are almost unaltered by our improvements to Slither: our analysis is fairly robust to modest tool improvements, unless added detectors account for a large number of mutants not detected by the tool. Adding such detectors will also only improve mutation ratio if they do not add many false positives. Substantial changes in results therefore require adding very effective (for mutants) detectors that seldom trigger for correct code (or at least trigger much less than for mutants). "Cheating" with respect to a mutation benchmark is thus, we hope, difficult without actual major tool improvements.

There were 92 separately ranked statement deletion mutants also. These, however, could all be ignored, as they were almost entirely duplicates related to the missing-return statement detector. If this detector were not already present as a private Slither detector, it would also be a good candidate for addition 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 either note as important or dismiss, 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 for **RQ6** 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, of course, has a detector for misuses of `tx.origin`, and we believe (but are not sure) that almost all of the differences with respect to such changes are due to intentional behavior. 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 good candidate detectors

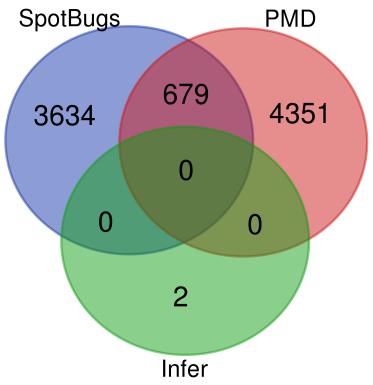

Figure 7: Mutants killed by Java static analysis tools.

Tool	Findings Mean	Findings Median	Mutation Score Mean	Mutation Score Median	Mutant Ratio
SpotBugs	28.93	14.00	0.07	0.07	0.002
PMD	53.73	32.00	0.07	0.07	0.001
Infer	11.60	3.00	0.00	0.00	0.000

Table 3: Java tool results over all projects.

Tool	# Clean Projects	Mutation Score Mean	Mutation Score Median
SpotBugs	3	0.05	0.06
PMD	0	N/A	N/A
Infer	6	0.00	0.00

Table 4: Java tool clean project results.

using mutant prioritization supports a limited affirmative answer to **RQ5** and **RQ6**.

3.1.6 Confirmation of Results. Because our random contracts' quality might be low, we also checked our results on 30 contracts from the Solidity documentation, the *Mastering Ethereum* book [2], and a handful of selected, recent, higher quality blockchain contracts. Slither had a mean mutation score of 0.11, vs. 0.04 for SmartCheck, and 0.01 for Securify. Associated mutant ratios were 0.38, 0.08, and 0.007. Mutant kill overlap was also similar; in fact, Figure 1 shows the results: Slither is A, SmartCheck is B, and Securify is C.

3.2 Java Tools

3.2.1 Static Analysis Tools Compared. For Java, we again compared three tools. SpotBugs (https://spotbugs.github.io/) is the "spiritual successor" of FindBugs [5, 41]. PMD (https://pmd.github.io/) [41] is an extensible cross-language static code analyzer. FaceBook's Infer (https://fbinfer.com/) [14] focuses on diff-related detection of serious errors (concurrency, memory safety, and information flow).

3.2.2 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 [37] for Java was useful, i.e. files that could be compiled and the bytecode compared without processing the full project, leaving us with just over 70,000 mutants, ranging from 136 to 10,016 per project.

https://stackoverflow.com/questions/4297014/what-are-the-differences-between-pmd-and-findbugs
https://www.sw-engineering-candies.com/blog-1/comparison-of-findbugs-pmd-and-checkstyle
https://www.reddit.com/r/java/comments/3i7w6n/checkstyle_vs_pmd_vs_findbugs_for_dummies_why/

Figure 8: Blog posts and forum threads discussing Java static analysis tools.

3.2.3 Analysis Results. Figure 7 shows the mutants killed by the Java analysis tools, and Tables 3 and 4 provide numeric results for projects and clean projects, respectively.

In terms of **RQ1**, the raw kills results suggest there is considereable 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. Unfortunately, these mutants only served to indicate that we were actually running Infer in such a way that it *could* detect problems: both were concurrency warnings, but the code change in both cases was simply removing a (semantically inoperative) `Override` annotation. 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 produced 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 recent academic comparison of these tools; the most relevant study dates from 2004 [41], used FindBugs, not SpotBugs, and reached no strong conclusions with respect to FindBugs vs. PMD. However, the user postings listed in Figure 8, plus personal communications with security analysts who use these tools (citation removed for now, due to potential

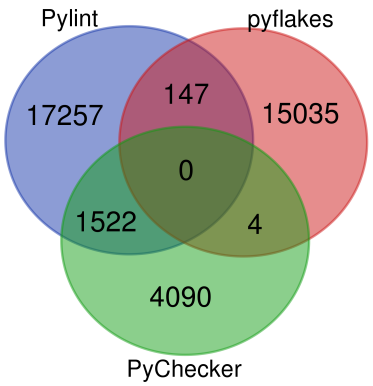

Figure 9: Mutants killed by Python static analysis tools.

Tool	Findings Mean	Findings Median	Mutation Score Mean	Mutation Score Median	Mutant Ratio
Pylint	2.24	1.00	0.22	0.20	0.097
pyflakes	2.4	1.00	0.03	0.00	0.012
PyChecker	0.56	0.00	0.10	0.00	0.183

Table 5: Python tool results over all projects.

to compromise blinding) supported some basic conclusions. SpotBugs/FindBugs is supposedly 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 crafted complex mutation operators targeting some important subtle Java bugs; it is still fair to say that Infer is probably not a good *general-purpose* static analysis tool for Java. Note that we did not use Infer's experimental, non-standard, detectors, however.

For **RQ4** there were very few clean projects, but the mutation scores for Infer were unchanged, and for SpotBugs they were worse. We see no evidence that non-clean code is a source of degradation in mutation detection.

3.3 Python Tools

3.3.1 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 (e.g., we found numerous `.travis.yml` files installing and running PyChecker), tool.

3.3.2 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 3,377 LOC to 34,671 LOC, with a total size of just under 100 KLOC. For Python, we were able to produce 167,511 valid mutants to analyze.

Tool	# Clean Projects	Mutation Score Mean	Mutation Score Median
Pylint	0	N/A	N/A
pyflakes	12	0.02	0.00
PyChecker	21	0.01	0.00

Table 6: Python tools clean project results.

https://stackoverflow.com/questions/1428872/pylint-pychecker-or-pyflakes
https://www.reddit.com/r/Python/comments/ii3gm/experience_with_pylint_pychecker_pyflakes/
https://www.slant.co/versus/12630/12631/~pylint_vs_pyflakes
https://news.ycombinator.com/item?id=12748885
https://doughellmann.com/blog/2008/03/01/static-code-analizers-for-python/

Figure 10: Blog posts and forum threads discussing Python static analysis tools.

3.3.3 Analysis Results. Figure 9 shows the mutants killed by the Python analysis tools, and Tables 5 and 6 provide numeric results for projects and clean projects, respectively.

For **RQ1**, Pylint and pyflakes both killed more than 15,000 mutants killed by no other tool; Pylint killed more mutants, but the relative difference was much smaller than in our Solidity or Java analysis. It is clear both tools are well worth using, assuming these are not false positives. Pyflakes' mutation score is lower than unique kills might suggest because it has almost no overlap with the other two tools. It is doing something different.

The mutant ratios **RQ2** suggest that PyChecker might be better than pyflakes, in fact, and better than Pylint, because while it killed fewer mutants, it also produced fewer findings by far, overall. PyChecker would be a good choice for quick and dirty analysis of code on the fly, perhaps, in that it had excellent mutation scores for some programs (hence a high mean) but is relatively "quiet" compared to the other tools. However, its very low median mutation score is troubling; it may simply not work for some code. A combination of pyflakes and Pylint seems safer, though adding the relatively low-findings PyChecker seems likely to have little cost. For Python, we see that all of our measures are required to get a good picture of the tools.

For **RQ3**, there were again no academic comparisons we could find. However, opinions on the web were quite common (see Figure 10, which lists ones we examined). It is hard to summarize 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 mitigated in our results by turning off clearly informative-only findings; users not configuring Pylint will probably see even more killed mutants, but likely a worse mutant ratio because mutants tend to introduce non-stylistic problems. Pyflakes is also well liked, and supposedly less verbose than Pylint; we did not see this in our results, but that may be because we configured Pylint, and pyflakes aims to focus on "actual bugs" so it is equally verbose for mutants, which are likely to be actual bugs.

For **RQ4** there were no clean projects for the known-to-be-aggressive Pylint. Performance for the other two tools was worse

on projects for which they were clean, again failing to support the possibility that our definition of killing a mutant is too restrictive.

3.4 Threats to Validity

The primary threat to validity in terms of generalization is that we only examined nine static analysis tools, and our analyses were restricted to 100 smart contracts, 15 Java projects, and 25 Python projects. Because it is hard to identify a ground truth to compare with (the motivation for our approach), we cannot be certain that our rankings of tools are correct even for these tools and this code. However, where there are existing discussions of the tools, our results seem to agree with these, but add substantial detail.

We used the Universal Mutator [22, 23], which aggressively produces large numbers of mutants, but does not target any particular software defect patterns, to generate all mutants. We will make available a repository containing our full raw results plus details on contracts and projects selected, for replication and further analysis, upon acceptance (to avoid breaking blinding).

4 RELATED WORK

The goal of "analysing the program analyser" [8] 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 [25]; 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. While comparisons of static analysis tools [15, 16, 38, 41] have appeared in the literature for years, these generally involved large human effort and resulting smaller scale, or did not make a strong effort to address false positives, or restricted analysis to, e.g., a known defects set [27].

Cuoq et al. [12] proposed the generation of random programs (*á la* Csmith [50]) 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 [31]. 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 arbitrarily 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). Most importantly, however, their approach only applies to tools that check assertions, rather than more general, popular static analysis tools that only identify bad code patterns.

Finally, the large body of work on using mutants in software testing [1, 6, 13, 19, 20, 29, 30, 39] is obviously relevant, in that we basically adopt its approach, but re-define killing a mutation for a static analysis context.

5 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 changing a mostly correct program usually introduces a bug into it; this is the basis of mutation testing, after all, and a large body of work supports the claim that at least 60-70% of mutants are fault-inducing. This means we can assume mutants are faulty, and escape the ground-truth/false positive problem that makes comparing static analysis tools so labor-intensive. Our approach cannot identify precision problems, directly, but combined with finding counts for un-mutated code, our approach can identify tools that detect mutants well, adjusted for their general tendency to flag any code as faulty, and thus identify imprecise tools. 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 [25, 32], or applying metric learning methods [33] (particularly unsupervised learning [42, 46]) are the most obvious and interesting possibilities for a better metric. Finally, mutant prioritization should be applicable to improving software testing, as well [20]. We are also interested in combining static and traditional mutation analysis: it might be useful to consider a mutant killed if it is detected either by static or dynamic analysis, using static kills to lower the number of required costly testing/verification runs for mutants, and the combined result as a useful guide to the quality of overall *defense in depth* [3, 21] for high-importance code.

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] Andreas M Antonopoulos and Gavin Wood. *Mastering Ethereum: building smart contracts and DApps*. O'Reilly Media, 2018.

[3] Nuno Antunes and Marco Vieira. Defending against web application vulnerabilities. *Computer*, (2):66–72, 2012.

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts SoK. In *International Conference on Principles of Security and Trust*, pages 164–186, 2017.

[5] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, Sep. 2008.

[6] 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.

[7] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2013.

[8] 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.

[9] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 597–608, May 2017.

[10] 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.

[11] ConsenSys. Mythril: a security analysis tool for ethereum smart contracts. https://github.com/ConsenSys/mythril-classic, 2017.

[12] 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.

[13] 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.

[14] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70, July 2019.

[15] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *International Conference on Software Engineering*, 2020. Available as arXiv preprint at https://arxiv.org/abs/1910.10601.

[16] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analyzer for solidity. In *International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 8–15, 2019.

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

[18] Rahul Gopinath, Carlos Jensen, and Alex Groce. Mutations: How close are they to real faults? In *International Symposium on Software Reliability Engineering*, pages 189–200, 2014.

[19] 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.

[20] 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.

[21] Alex Groce, Josselin Feist, Gustavo Grieco, 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.

[22] 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.

[23] 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.

[24] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[25] 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.

[26] B. J. M. Grün, D. Schuler, and A. Zeller. The impact of equivalent mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 192–199, April 2009.

[27] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 317–328, New York, NY, USA, 2018. Association for Computing Machinery.

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

[29] 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.

[30] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[31] 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.

[32] Todd Kulesza, Margaret M. Burnett, Simone Stumpf, Weng-Keen Wong, Shubhomoy Das, Alex Groce, Amber Shinsel, Forrest Bice, and Kevin McIntosh. Where are my intelligent assistant's mistakes? A systematic testing approach. In *End-User Development - Third International Symposium, IS-EUD 2011, Torre Canne (BR), Italy, June 7-10, 2011. Proceedings*, pages 171–186, 2011.

[33] Brian Kulis. Metric learning: A survey. *Foundations & Trends in Machine Learning*, 5(4):287–364, 2012.

[34] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. MuCheck: An extensible tool for mutation testing of Haskell programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 429–432. ACM, 2014.

[35] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[36] William McKeeman. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation*, 10(1):100–107, 1998.

[37] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple fast and effective equivalent mutant detection technique. In *International Conference on Software Engineering*, 2015.

[38] Reza M. Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, page 103–113, USA, 2018. IBM Corp.

[39] 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.

[40] Phil Daian . Analysis of the dao exploit. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/, June 18, 2016 (acceded on Jan 10, 2019).

[41] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, page 245–256, USA, 2004. IEEE Computer Society.

[42] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.

[43] Ben H Smith and Laurie Williams. Should software testers use mutation analysis to augment a test set? *Journal of Systems and Software*, 82(11):1819–1832, 2009.

[44] SpankChain. We got spanked: What we know so far. https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe, Oct 8, 2018 (acceded on Jan 10, 2019).

[45] 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.

[46] Michael E Tipping and Christopher M Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(3):611–622, 1999.

[47] Trail of Bits. Echidna: Ethereum fuzz testing framework. https://github.com/trailofbits/echidna, 2018.

[48] 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.

[49] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf, 2014.

[50] 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.