# Using Differential Mutation Analysis to Compare and Improve Static Analysis Tools


## ABSTRACT

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



**ACM Reference Format:**

. 2020. Using Differential Mutation Analysis to Compare and Improve Static Analysis Tools. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 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 tradeoffs) they might want to imitate. Unfortunately, comparing tools is hard, and would seem to require vast manual effort to inspect findings and determine ground truth, to obtain any statistical validity.

Differential testing [15, 19] 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 [6].

Mutation testing [2, 7] (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 [10] paper and a 2018 journal extension [11] of that paper, Groce et al. proposed examining individual mutants that survive a formal verification or automated test generation process to detect and correct weaknesses in a specification or test generator. The approach was able to expose bugs in a heavily-tested module of the Linux kernel [1] and improve a heavily used test generator for the `pyfakefs` file system. Recently, mutation analysis has been adopted in industrial settings, though not for actual examination of all surviving mutants [17, 20], 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 both to users wishing to select a good tool or set of tools, and to developers hoping to improve their own tools.

### 1.1 Differential Mutation Analysis

We can say that a static analysis tool kills a mutant when the number of (non-informational or optimization related) warnings or errors produced with respect to the code *increases* for the mutated version, compared to the original code. This difference is most informative and easily interpreted when the original code produces no warnings or errors (it is "clean"); for non-clean code, a tool conceivably could detect the mutant, but only change a previously




*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

generated warning, not add an additional warning, leading to an underestimate of effectiveness. However, even for non-clean code, most detected mutants will produce a new warning.

The value of the differential comparison lies in a few key points. First, this is a measure that does not reward a tool that produces too many false positives. The tool cannot simply flag all code as having a problem or it will perform poorly at the task of *distinguishing* the mutated code from non-mutated (and presumably at least *more* correct) code. Based on verification and testing uses of mutation, it is safe to say that at least 50, and likely 60-70% or more, of mutants that are not semantically equivalent to the original code are actually fault-inducing, so the task presented to a static analysis tool is the generalization of the task we ideally expect static analysis to perform: to identify faulty code, without executing it, and, most critically, *to distinguish faulty from correct code*. Obviously, many faults cannot be identified statically without a complete specification, or without unreasonable analysis cost and precision, but the measure of performance here is meant to be mostly *relative* to other tools applied to the same code; this is primarily a *differential* approach. 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 [6]. 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 contracts while still detecting *more* mutants. Other than having better precision and recall, how else could a tool effectively distinguish mutated from un-mutated code?

We can quantitatively express this relationship between detecting mutants and findings for the original program. Since we are comparing over the same program or set of programs, we simply

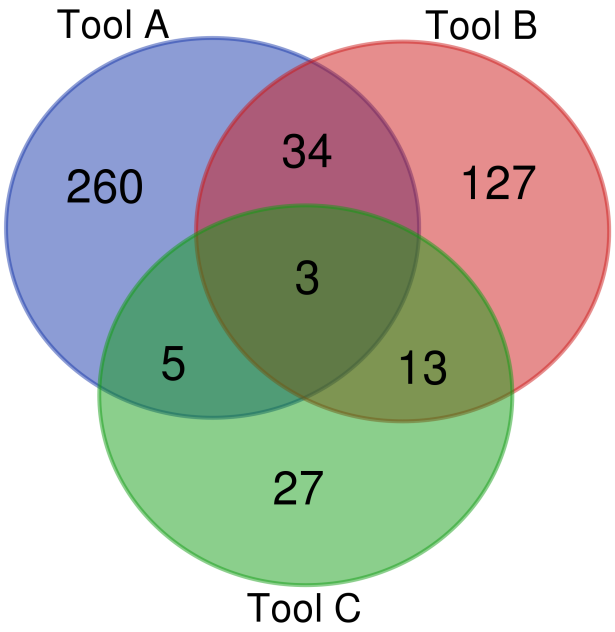


**Figure 1: Mutants killed by three static analysis tools.**

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

```
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.**

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 [9] algorithm and distance metric inspired by previous work on helping developers sort through failing test cases and avoid duplicates [5], to help static analysis tool developers find interesting patterns without wading through numerous uninteresting duplicates.

## 1.2 A Simple Example

Consider the code in Figure 2, from the Solidity 0.4.24 "Introduction to Smart Contracts". The Universal Mutator tool [13, 14], which has been extensively tuned for Solidity mutation (and is the only smart contract mutation tool referenced in the Solidity documentation (https://solidity.readthedocs.io/en/v0.5.12/resources.html)) produces seven valid, non-redundant mutants for this trivial example code. Both the public version of Slither [8] and SmartCheck [23] produce a small number (three and two, respectively) of low-severity, informational, warnings for this code. Both tools also detect (by increasing the number of warnings produced) four of the seven mutants. However, only one of the mutants detected is common to both tools: both tools detect changing the `return` statement in the `get` function to a call to `selfdestruct` the smart contract. Slither, but not SmartCheck, also detects replacing the assignment of `storedData` in `set` with either a `selfdestruct` or `revert`, or simply removing it altogether. SmartCheck, on the other hand, detects removing the `return` in `get` or replacing it with a `revert`, or removing the `public` visibility modifier for `get` [1]. If we restrict our analyis to 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 mutant saves little effort), so we will show the utility of that approach 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.
- 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.

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

## 2 PRIORITIZING MUTANTS

## 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 our approach 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 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?

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.

### 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 [3, 25]; 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 [21, 22]. 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 [12].

We analyzed three well-known tools for static analysis of Solidity smart contracts: Slither [8], SmartCheck [23], and Securify [24].

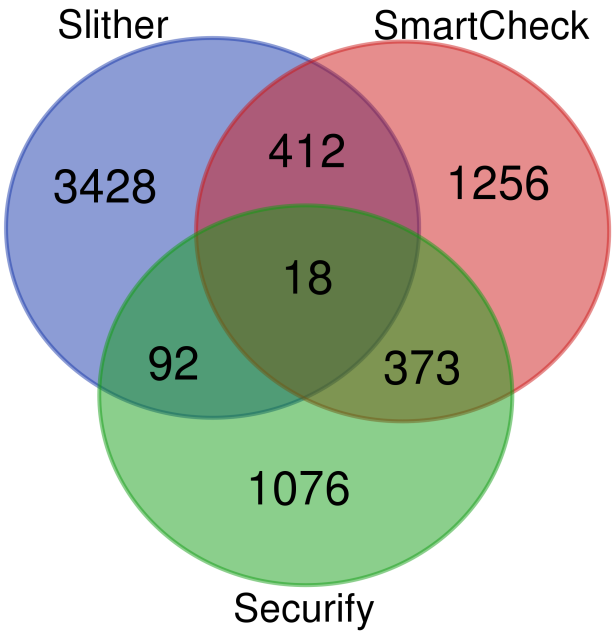


**Figure 3: Mutants killed by Solidity static analysis tools.**

| Tool | Findings | | Mutation Score | | Mutant |
|---|---|---|---|---|---|
| | Mean | Median | Mean | Median | 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 tools: number of findings and mutation scores over all contracts.**

*3.1.2 Original Program 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.

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

| 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 tools: clean contract counts and mutation scores.**

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 8 days sequential compute time for Slither and 3 days for SmartCheck), 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.

We first compare the two tools that performed best. Slither detected 2.37 mean issues over the 100 contracts, and found 39 contracts free of all issues. Slither had a mean mutation score of 0.09 over all contracts, and 0.11 over clean (according to Slither) contracts. SmartCheck detected 1.89 mean issues over the 100 contracts, but only found 27 contracts free of all issues. The mean issue counts for the tools were not statistically significantly different ($p = 0.34$). SmartCheck had a mean mutation score of 0.05 over all contracts, and 0.03 over clean (according to SmartCheck) contracts. The ratio between mean issues (Slither / SmartCheck) is 1.25, and the ratio of numbers of clean contracts is similar (1.44). The mutation score ratio, however, is 1.8, suggesting that Slither is not outscoring SmartCheck *only* because it produces more false positives. Slither detects 3,520 mutants not detected by SmartCheck, while SmartCheck only detects 1,629 mutants not detected by Slither (of these, 1,256 were also not detected by Securify) . Over the 7 contracts that both tools mark as clean, Slither has a mean mutation score of 0.08, and SmartCheck has a mean mutation score of 0.03; the difference, even with only 7 contracts, is statistically significant by paired Wilcoxon test, at $p = 0.03$.

Securify was an outlier, compared to the other tools. First, while running Slither and SmartCheck on all 46,769 valid mutants was relatively quick, Securify often required hours to analyze a contract; the full analysis required over a month. Second, Securify identified a mean of 24.65 issues on the un-mutated contracts, with a median of 17 issues found; this was a significant difference in issue counts from either other tool, with $p < 1.0^{-15}$ in both cases. Despite this huge number of non-informational warnings for the original contracts, the mean mutation score was even worse than for SmartCheck, 0.03. For contracts where Securify found no issues in the original contract, it *never* detected any mutants. Securify, however, did detect 1,449 mutants not detected by Slither, 1,076 of which were not detected by SmartCheck. On the other hand, given the ratio of issue counts compared to both tools is over 10.0, it is fairly likely many of these *are* false positives, despite the probability mutants are actual faults.

Our recommendation, based on mutation analysis results, is that smart contract analysis should at least use both Slither and SmartCheck, perhaps applying only Slither until it reports no interesting issues. Securify does offer a substantial non-overlapping set of issues reported, however, but produces a very large number of false positives. Available human resources and total number of issues produced (as well as high runtime) might justify not using Securify in some cases.

*3.1.4 Improving Slither* **(RQ 4).** 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 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 this mutant, killed by SmartCheck but not Slither:

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

(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/description_en.html, 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 a mutants killed by SmartCheck but not Slither such as:

```
x968815CD73647C3af02a740a2438D6f8219e7534_TTPresale.
      require(nextDiscountTTMTokenId6 >= 361 && nextDi
 ==>  ...361...==>...0...
      require(nextDiscountTTMTokenId6 >= 0 && nextDisc
```

(3) **Loss of precision due to divide before multiply:** 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/description_en.html, which enabled it to detect mutants such as:

```
x534ccee849a688581d1b0c65e7ff317ed10c5ed3_NametagTok
      byte char = byte(bytes32(uint(x) * 2 ** (8 * j)))
 ==>  ...*...==>.../...
      byte char = byte(bytes32(uint(x) * 2 ** (8 / j)));
```

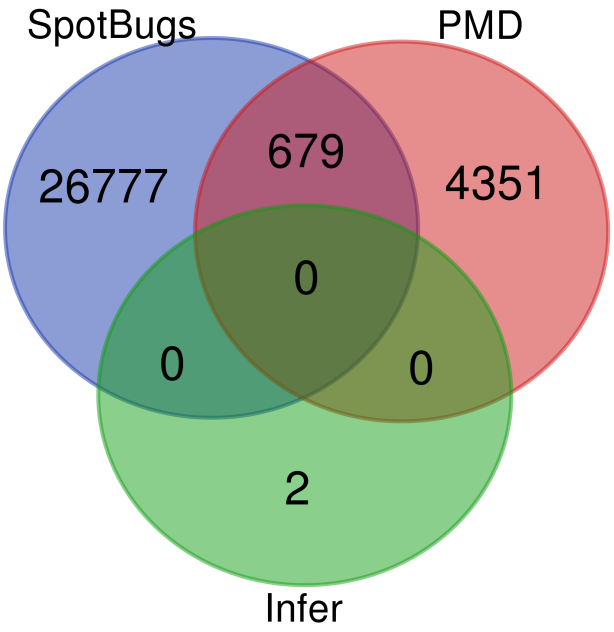


**Figure 4: Mutants killed by Java static analysis tools.**

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.

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).

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. The majority of cases involved either the missing return problem or replacing `msg.sender` with `tx.origin`; Slither has a detector for misuses of `tx.origin`, and we believe (but are not sure) that almost all of these are due to intentional behavior: SmartCheck and Securify tend to identify most uses of `tx.origin` as incorrect, while Slither has a more selective rule intended to avoid false positives.

| Tool | Findings Mean | Findings Median | Mutation Score Mean | Mutation Score Median | Mutant Ratio |
|---|---|---|---|---|---|
| SpotBugs | 0.26 | 0.00 | 0.30 | 0.07 | 1.132 |
| PMD | 0.48 | 0.00 | 0.09 | 0.00 | 0.183 |
| Infer | 0.00 | 0.00 | 0.00 | 0.00 | 0.006 |

**Table 3: Java tools: number of findings and mutation scores over all files.**

| Tool | # Clean Files | Mutation Score Mean | Mutation Score Median | Clean For All (1,185) Mutation Score Mean | Clean For All (1,185) Mutation Score Median |
|---|---|---|---|---|---|
| SpotBugs | 1,426 | 0.32 | 0.05 | 0.37 | 0.06 |
| PMD | 1,359 | 0.08 | 0.00 | 0.08 | 0.00 |
| Infer | 1,598 | 0.00 | 0.00 | 0.00 | 0.00 |

**Table 4: Java tools: clean file counts and mutation scores.**

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 using mutant prioritization supports a limited affirmative answer to **RQ4**.

## 3.2 Java Tools

*3.2.1 Original Program Selection.*

*3.2.2 Analysis Results.* Figure 4 shows the mutants killed by the Java analysis tools.

### 3.3 Python Tools

### 3.4 Threats to Validity

## 4 RELATED WORK

The goal of "analysing the program analyser" [4] 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 [16]; most tools will not produce "the right answer" all the time, by their very nature. This is a result of both algorithmic constraints and basic engineering trade-offs.

Cuoq et al. [6] proposed the generation of random programs (*á la* Csmith [26]) 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 [18]. Their approach is in some ways similar to ours, in that it takes as input a set of seed programs, and compares results across new versions generated from that seed. The primary differences are that their seed programs must be warning-free (which greatly limits the set of input programs available) and their tool must parse and understand the programs, and that the new versions are based on adding new assertions, not "breaking" the original code. We allow arbitarily buggy seed programs (thus many more real programs can be used), and can, due to the any-language nature of the mutation generator we use, operate even in new languages without further development effort. Further, their approach only identifies problems when tools are outliers compared to numerous other tools in either detecting a bug (precision) or not detecting it (soundness), and so requires comparing multiple tools. Our approach has some utility for even a single tool (you can just examine prioritized un-detected mutants). On the other hand, their approach can identify precision issues, while we offer no real help with false positives (in theory, you could apply their majority-vote method to mutants only a few tools flag, but mutants *are* usually faults. in contrast to their introduction of checks that may be guaranteed to pass, so this is probably not very helpful).

## 5 CONCLUSIONS AND FUTURE WORK

## REFERENCES


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

[2] Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. pages 220–233. ACM, 1980.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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).

[22] 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).

[23] 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.

[24] 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.

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

[26] 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.