A Smarter Smart Contract Testing Tester

Vincent Yang , Sahana Mundewadi, Joseph Kotlarek, Doug Sherman

ABSTRACT

As the drive for decentralized frameworks based on blockchains such as Bitcoin and Ethereum continues to grow, the need for reliable and accurate Smart Contracts is increasingly important. To this end, testing suites such as Mythril and Oyente have been produced that employ static analysis techniques such as symbolic and chain execution. These methods have been shown to vary wildly in which bugs they can detect, and studies have been made to demonstrate the number of bugs each can detect. However, there are minimal studies discussing the relevance of the bugs detected; which is vital when bugs such as the DAO bug can lead to over 60 million in damages. Thus we introduce a novel system for testing smart contracts with both Mythril and Oyente through a user-driven interface. This tool allows the user to specify what parts of the code are most important and emphasize which tests provide the most coverage over these regions. We used these tools to study the DAO Solidity scripts as well as 40 other scripts including `Honeypot.sol` and `BrokenToken.sol`. We found that Oyente consistently finds more bugs than Mythril, but Mythril provides further insight into the source of these bugs. Moreover, the trend between code complexity and bugs is a bimodal trend where the smallest and largest scripts contain the most problems. Overall, we have shown this tool allows users to explore their smart contracts with more control using metrics truly relevant to their search for mistakes.

1. INTRODUCTION

Since Bitcoin was first introduced in 2009, decentralized transaction systems have surged in popularity in both the academic and private sectors [1]. To maintain decentralization, these systems rely on a consensus ledger maintained solely by the users, and thus no institution is required to maintain it. Blockchains were created as a way to handle the intricacies of a decentralized ledger system for transactions; however, today blockchains are used for much more. Miller et al. demonstrated a way to preserve data using blockchains by incentivizing users to store large volumes of data rather than provide large amount of computing power [2], and use cases have been proposed to built entire operating systems from the blockchain structure [3, 4]. However, with no central force governing these tools, new problems arise to handle secure and trusted transactions, data transfers, and events.

Thus, smart contracts were developed to provide a secure way to handle events in decentralized systems. A smart contract follows the success of Bitcoin's consensus ledger by providing a "consensus protocol" that a blockchain decides upon to guarantee any event between two entities is handled without third party intervention and results in costly penalties for any party found in breach of the contract [5–7]. These contracts can be used for transactions between two people, allow for digital identities, and facilitate data sharing. An example of a smart contract is shown in figure 1. In this example we have a basic script that sends money only if the condition `defined_condition.is_met()` succeeds. This condition can be any collection of jointly agreed upon requirements, and thus a wide array of contracts can be decentralized using smart contracts.

Unfortunately, the current state of smart contract code quality is remarkably dismal. Solidity, the most popular smart contract programming language, was created to be similar to JavaScript and attract as many users as possible [8, 9]. Many developers on Solidity are drawn in by the sensationalized articles of blockchain and "Hello World" applications are everywhere to be found. Many developers do not know about the specifics of Solidity and the many pitfalls. One of the biggest pitfalls is that many functions can fail [9]. Most notably, the send function – the function that sends money from one account to another, can fail. When this happens, it is currently the developer's responsibility to roll back what has happened appropriately [6]. In fact, this has been the cause for many significant attacks on the Ethereum network.

By nature of the blockchain, developing decentralized applications is quite different from that of normal development. Since the blockchain is created to be immutable, patches are issued as new versions of the same program, and everyone is encouraged to switch over. However, the old version of an application or contract is still accessible on the global database, the ledger. Additionally, it takes time for an application to spread throughout the network [6]. This inability to roll back increases the importance of getting code right on the first try [6].

Luckily, many researchers have worked to mitigate these attacks by providing code analysis tools. These analysis tools cover a wide variety of bugs and coding styles to promote clean, readable code[10]. Unfortunately, not all of these tools are easy to use and many are poorly documented. This is where our tool comes in play.

Our tool works by allowing people to run their code on a variety of testing tools. By making this process as simple as selecting a set of testing suites, the users can easily see what errors are generated by what vulnerabilities and from what testing suite. Additionally, we are working to allow the user to specify code snippets of higher priority so we can rank errors in those regions higher. Another neat benefit is that many of these testing suites are still in active development, so new vulnerabilities would still be caught without any further action from the developer. In the grand scheme, we want to promote code quality and make improving code quality on

smart contracts a trivial task. Hopefully this improves the state of smart contracts for the community as a whole.

```
function approve(address _spender, int _value)
{
  if (defined_condition.is_met()) {
    // Contract succeeds so send money
    returns bool(success) {
      allowance[msg.send][_spender] = _value;
      return true;
    }
  }else{
    // Fails, so money is returned to spender
    returns bool(failure) {
      allowance[_spender][_spender] = _value;
      return true;
    }
  }

}
```

Figure 1: An example of a smart contract that sends money if a certain condition is met; allowing for many types of contracts.

2. RELATED WORK

Recently, research into removing bugs from smart contracts has surged. This has lead to an array of tools designed to ensure the stability and accuracy of smart contracts. This has lead to three distinct schools of thought in bug detection. One method is Static analysis, where tools rely on techniques that do not require running the code, such as determining any paths that are unreachable [11]. Code coverage, which is focusing solely on testing as much of a block of code as possible, and linters, a technique designed to preemptively catch bugs during development, account for the remaining tool categories [12]. Based off studies from researchers like Kochhar et al., code coverage is known to have little to no correlation with actual bugs [13, 14], and linters need to be employed before development; providing no support for the extensive list of smart contracts already in use. Thus, our study focuses primarily on comparing static analysis tools through a wide array of smart contracts.

The current smart contract static analysis tools employ a wide variety of techniques to study the code. Mythril, Oyente, and Porosity. Each of these techniques travel branches within the solidity script, studying possible points of failure along the way. Oyente employs symbolic execution, a technique that uses symbols to determine which values traverse certain paths in the code [15]. Mythril provides a comprehensive blockchain exploration tool that has finally allowed chain exploration to succeed in smart contract testing [16]. Porosity translates EVM bytecode into Solidity; allowing for static and dynamic analysis of the contracts [17].

Fontein studied each of these static analysis techniques in detail. In this study, he found that Oyente detects on average 75% of the errors, but severely lacks in missed timestamp dependency bugs. Moreover, since Oyente relies on symbolic execution, a missed opportunity to provide an export of the script's control flow graph (CFG) could drastically improve it's effectiveness. Porosity's decompilation tools were shown to be very limited, and this tool only detects re-entrancy bugs. However, many of the re-entrancy bugs it detects are incorrect. Fontein defined Mythril as the clear winner as it detected the most bugs without false positives. However, Fontein did not mention the quality of the bugs, such as fixing the infamous DAO bug that cost almost 60 million in lost funds [18].

Even with this surge in tools, smart contracts filled with bugs are still be created. This is largely due to the lack of usability within each tool. With the huge opportunity for smart contracts in the financial sectors, many non-technical developers are creating contracts of their own [4]. However, these tools require knowledge of concepts such as EVM bytecode or symbolic execution. Thus, systems have been developed to provide user friendly interfaces to these tools. SmartCheck automatically checks contracts with code highlighting of errors. This tool allows users to run tools on their code with ease, but the analysis is limited by the tools in the backend of the code. Thus, user-driven studies are limited, and focusing analysis on important aspects is impossible [19].

Thus, we explored these smart contract testing solutions, and validate these tools using novel metrics that consider the bugs unique to blockchain platforms and the security problems presented by these decentralized transactional events. Moreover, we developed a tool for comparing these testing suites on a specific codebase through user-driven studies. This tool combines visualizations of the bugs discovered by the testing suites, and explores developer driven metrics allowing specific blocks of code to be tagged with more importance. Thus, allowing developers to compare available testing options on their own codebase with metrics of far more meaning than generic code coverage.

3. METHODS

3.1 Testing Suites

Oyente

Oyente works by finding the gap between contract programmers' assumptions about the system and the actual nature of the network. This gap is the core reason for thousands of bugs on the Ethereum network – the most popular smart contract platform today. OYENTE uses symbolic execution to analyze Ethereum smart contracts [6].

As proof of OYENTE's effectiveness, the creators tested 19,366 smart contracts and found that 8,833 contracts have potential bugs. The total value in these contracts was equal to around 30 million USD when they created the tool. Ethereum's value has risen dramatically since. Over the last few years, several vulnerabilities have made headlines worldwide. The most significant of these is TheDAO bug, which had a 60 million loss in USD. If the creators had used OYENTE, they would have caught this bug early on.

Some notable bugs Oyente searches for are mishandled exceptions, deliberately hitting the stack limit, timestamp-independent calls, and functions that depend on transaction ordering. The first two are significant because the developer must explicitly implement what happens when a call fails. For instance, in the `send` function, where people send Ether to someone else, it is fully possible for this function to fail. In response, the developer must revert all that has happened thus far, such as un-withdrawing money from an account. Next, timestamp-independence and transaction ordering are important because miners are allowed to choose what contracts to run. Often times, developers use timestamps as seeds for randomization, but this is easily attacked when miners can determine the seed. This logic applies similarly for transaction or-

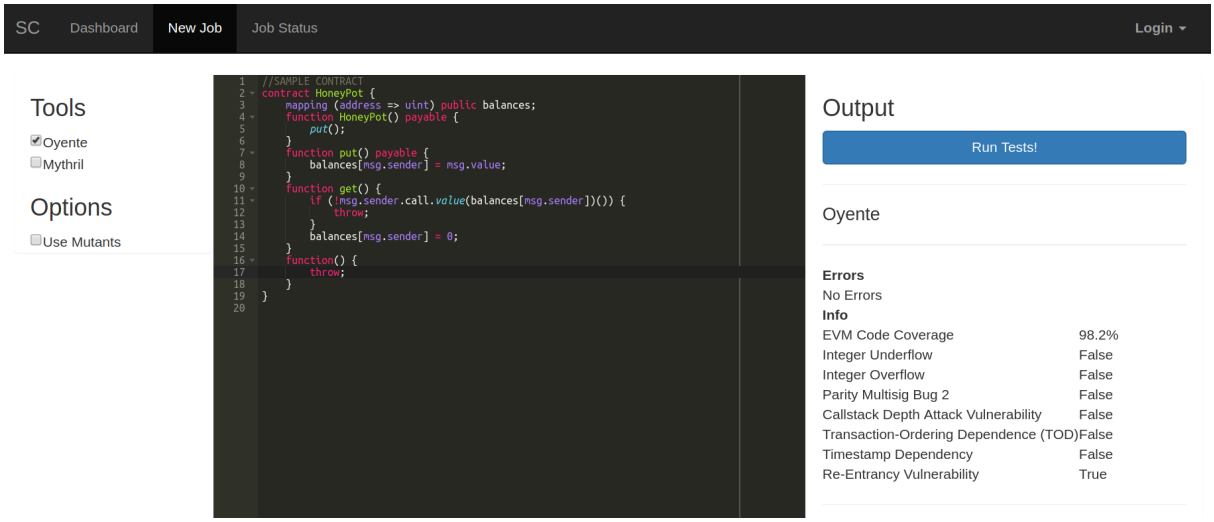

Figure 2: The Smarter Smart Contract Testing Tester User Interface

dering. When the result of a contract is dependent on another contract's running before or after it, then miners have influence over the effect of the contract. Thus, the contract is not secure.

Creators of Oyente chose to use symbolic execution, which uses symbols to represent what conditions some combination of variables might satisfy. This was preferable to dynamic testing, which would try much more inputs. This would be even worse than normal in Ethereum, because there could be far more variables on a decentralized network. For instance, in order to check timestamp-ordering independence, the program would have to check every possible variation of race condition caused by multiple programs interacting with each other.

More specifically, Oyente has several key components. It simulates the Ethereum Virtual Machine with 4 tools: the CFG Builder, Explorer, CoreAnalysis and Validator. The CFG Builder creates a control flow graph to see which parts of code can jump to others. Some of these edges are generated with symbolic execution, as it may be difficult to determine all the jumps with only static analysis. Next, the Explorer runs through the graph created by the CFG Builder and executes symbolic instructions given a certain state. It runs symbolic instructions for every single state to figure out if branch conditions may be met. At this point, it also adds potential edges to the CFG. After the Explorer runs, we are left with symbolic traces that show what conditions must be satisfied for certain events to occur. To do the actual analysis, we have CoreAnalysis. This runs along the symbolic traces and searches for the various vulnerabilities listed previously. Finally, we have Validator. This looks for false positives. Given the potential execution trees, it checks if the symbolic variables contradict one another.

Mythril

Another tool similar to Oyente, is Mythril, a security analysis tool specifically for Ethereum smart contracts written in Solidity. It uses concolic execution, or a mix of symbolic and concrete execution, taint analysis to find which variables take user input (and thus make symbolic), and control flow checking to detect a variety of security vulnerabilities. These vulnerabilities include integer underflow, owner-overwrite-to-Ether-withdrawal, and many others. However, Mythril does not handle any business logic and is therefore not a formal verfication tool.

This tool is built on the Ethereum Virtual Machine, or EVM, using the Laser-Ethereum, which is a symbolic interpreter for Ethereum bytecode. The EVM is what allows Mythril to successfully use symbolic execution, as it is much simpler than a desktop or mobile operating system, and thus allows the symbolic execution to reach 100% code coverage. With the EVM, devlopers of Mythril did not have to worry about hard to model features of an Operating System such as filesystems, sockets, or multi-threading. Laser-Ethereum takes in at least one smart contract account as input and returns a set of abstact program states. Each state is made up of the set of values of all the variables in the EVM at a given point in execution.

Mythril has many detection modules, including unchecked suicide or ether_send, unchecked_retval, external calls to untrusted contracts, integer overflow/underflow, etc. The error of unchecked "suicide" or "self-destruct", which is the act of sending the remaining balance of a contract to a specified user, has been exploited as recently as November 6th, 2017, on the Parity multisig wallet library contracts, where 280 million dollars worth of Ether was made inaccessable. Mythril has proved to be very effective at detecting this dangerous vulnerability (as well as many other bugs) and will help prevent such attacks in the future by giving developers a chance to test their code for vulnerabilities before deploying their contract.

Operation	Mutations	Description
+	−	Occurs often when increasing the amount owed, or the amount paid. Can lead to errors associated with loops that end when a balance is paid.
>	<	Can cause limits to be ignored, such as with debate period, leading to errors caused by exceeding maxima.
/	*	Commonly used during increments of some delta time or delta value measure. Can lead to infinite loops when relying on total value to exceed some measure.
&	|	Allows checks that require multiple steps to succeed to move forward if only one condition is met.
==	!=	Hard equality checks, such as the 51% attack check in the example in figure 3.3, will work reverse from intended.

Table 1: The mutations applied to each script. In this table any instance of the Operation column will be replaced by the Mutations column and vice versa. The descriptions section describes circumstances that would cause more bugs to arise from this mutation.

3.2 User Interface for Analysis

To take advantage of the Mythril and Oyente testing suites, we present our user interface that allows running several testing applications and viewing the results interactively in a browser. Users can upload and modify or write a new Solidity Smart Contract in the syntax-highlighted editor, and choose to run Oyente, Mythril or both to detect security vulnerabilities in the code. The output is displayed as a verbose list of errors and warnings with exact line numbers identifying the cause of the vulnerability.

The application is a web based front-end that can run in any browser, supported by a Python back-end that automatically schedules tasks and prepares environments for each tool individually, reducing the amount of set-up time and knowledge required on behalf of the user to effectively nothing beyond loading a web-page. Additionally, the back-end is able to maintain the environment using docker containers to drastically speed up future runs of the test suites, reducing the overhead time and computational costs of each run after the first from the order of minutes to mere seconds.

Portability and accessibility are achieved through Docker containers that are managed by the API. Each container has the necessary environment and dependencies, automatically fetching them from a trusted repository, so that even running locally, the tool requires minimal set-up. While it can be run locally, the application as a whole is intended to be used entirely web-based, uploading the Smart Contract to a remote server for testing and returning the results within a browser window.

Our solution covers several use cases, primarily those of the Smart Contract writer, the second party to a contract, as well as reviewers of the implemented test-suites. The Smart Contract writer is able to interactively view vulnerabilities in her code and quickly make iterative changes with the built-in and fully syntax-aware code editor window. The second party, assumed to be ignorant of the finer points of Smart Contracts, including how to write them, is able to verify security and integrity without having to learn from scratch how to run any of these complicated test suites on his own. Finally, a reviewer can take contracts with known or unknown flaws and compare the outputs of multiple tools to test which produces the best or most accurate output.

This user interface is designed and used not only for our study to measure the ability of Oyente and Mythril to detect security flaws, but also for the public to use to have easily accessible testing capabilities for any Smart Contract written in Solidity. With this tool, testing a Solidity Smart Contract is easy and straightforward, since the entire process from fetching dependencies all the way to destroying the containers on completion is managed totally automatically. This end-to-end approach is intended to make it possible for a lay-person to engage in a Smart Contract knowing that it has been tested by the best open-source tools available, while still providing the important technical information that the contract writer needs to create a sound and error-free contract.

3.3 Analysis Techniques

Mutation Studies

To expand upon the testing suites, we added mutations to the Solidity scripts to compare which suites would recognize that these perturbations. To create these mutations we replaced standard operations with contrary logic while avoiding syntactical errors. With these changes, we ran the testing suites again on the altered scripts.

The mutations were applied as a search and replace of a single operation per test. Thus, a file containing 10 unique operations would produce 10 mutated files each with a single operation mutated. For the mutations, we chose operations that would reverse the logic of the of the given statement. For example, consider the block of code from the DAO smart contract in Figure 3.3.

```
1  /// @param _recipient Address of the recipient
2  /// @param _amount Amount of wei to be sent
3  /// @param _debatingPeriod Time to debate
4  /// @param _proposalDeposit The new deposit
5
6  if (!allowedRecipients[_recipient]
7      || _debatingPeriod < minProposalDebatePeriod
8      || _debatingPeriod > 8 weeks
9      || msg.value < proposalDeposit
10     || msg.sender == address(this)
11     )
12         throw;
```

Figure 3: Example script from the DAO smart contract codebase.

In this example, there is a list of checks that are made prior to scheduling a contract between a `sender` and `recipient`. Specifically, the receiving party is first checked against a list of allowed recipients, the debating period, or time frame to cancel the contract, is confirmed to be in a predefined range, that enough money has been deposited to cover this transaction, and importantly

		Cyclomatic Complexity	Dependencies	LOC	Coverage	# Errors	# Issues
Oyente	Original	16.07	0.302	81.613	18.167	2.442	10.535
	Mutated	44.249	0.444	81.613	10.792	57.026	0.9
Mythril	Original	16.07	0.302	81.613	18.167	0.977	5.767
	Mutated	44.249	0.444	81.613	10.792	1.077	3.209

Table 2: The statistical analysis of the different test suits for both the original and mutated scripts. This shows that Oyente detects many more bugs, and this is much more pronounced with the mutated scripts.

a check against a 51% attack, where a majority owner of the block chain can cheat the system, is protected against. A valid testing suite should recognize the insecurities presented if any one of this statements is not checked. Thus our mutations primarily change the logic in checks like the one above. A simple way to guarantee this is by reversing the logic on each operation. For example the check against a 51% attack relies on a equality constraint, thus if this changed to an inequality we can guarantee that check will fail. A complete list of the mutations made on each Solidity script is given in table 3.1.

Static Analysis

In addition to Mutation studies, we also performed static analysis on the solidity scripts to provide metrics to help explain why one suite performed better than another. These metrics include measuring the number of lines, complexity, and dependencies in a block of code. These metrics were chosen following the results of Kochnar et al.'s code coverage study [13].

The statistical measures used to analysis the solidity script were Lines of Code (LOC), Cyclomatic Complexity, and the number of Dependencies. Lines of Code was directly computed from each script after removing whitespace and comments. One limitation of this measure, is that it does not account for varying coding styles, such as placing brackets in line with function signatures, but overall this provided a good metric for length of a script. Cyclomatic Complexity measure the dependence between certain paths within a block of code. This metric increases by one at each point where a new branch will need to be tested; such as an if statement or logical decision [20]. Dependencies measures the number of imports into a script, causing this file to require correct execution of the dependent file. For example a DAO solidity package consists of many separate scripts that each import a base DAO class. Thus each of these would have a dependency of at least 1. With these measures, we were able to balance the testing suite results with concrete information that can easily be pulled from a script; allowing for breadth in our study.

4. RESULTS & ANALYSIS

4.1 Testing Comparison

As part of our analysis, we created the SSCTT Web-based User Interface, and we use it drastically streamline the pipeline for running Solidity contract test suites. And while it still currently only supports Oyente and Mythril, we've already been able make full-fledged security vetting of smart contracts possible for any person regardless of skill level. While it is not in the scope of this paper, we intend to perform a user study and formal evaluation of our tool in the future. Even without this, however, the impact of moving these test suites from command-line tools to fully managed and contained web-apps is clear. Users are able to see prettified results in order of importance to them. Furthermore, the accessibility benefit of automating the set-up and tear-down of the existing tools is motivated by the need for uneducated parties to be able to verify their smart contracts' security. Thus, we consider the Web Interface a success in this regard.

4.2 Mutation Tests

Both suites provide very different results when trying to analyze a Solidity script. To understand the reason behind this, we employed a combination of mutation testing and static analysis methods. On average, each script had 9.27 operations that were mutated to produce a list of mutated scripts. Then each of the two suites were run on the original Solidity files and on each of the mutations.

Figure 3.2 shows a summary of the errors and issues found by each suite across our collection of solidity scripts. Here we see that Oyente found 83% more issues than Mythril on the original document, and 5300% more errors on the mutated scripts. However, the Mythril suite found significantly more issues on the mutated scripts. It is important to note that each of our solidity scripts consisted of only 81 lines of code on average. In general these contracts are very small, and define a very specific purpose of ensuring transactions are completed according to all parties.

4.3 Static Analysis

To study the suites in more detail, we plotted the number of bugs found by each suite (Figure 4 Left). If each suite were exactly the same, then each point would lie on the $y = x$ line. However, in this plot we see a consistent trend to the right of this line. This means that Oyente has found more bugs in every script. Moreover, on the mutated code, this trend is more pronounced on more of the scripts.

The plot to the right of Figure 4 shows the number of bugs as a function of complexity of the script. Here we see a trend where scripts that are smaller and scripts that are larger tend to have more bugs. It is expected that a larger script would have more bugs, but scripts with average complexity have much less bugs than low complexity scripts. A possible explanation for this is that smaller scripts tended to have significantly less error handling; thus bugs would be caught less.

5. CONCLUSION

We noticed that the correlation between complexity and errors is not as expected. In general, a larger file with more complexity should have more bugs, but we found that the the smaller files had almost as many bugs as the largest files. Moreover, files with average complexity had the least number of bugs. We attribute this to there being less code in smaller files, so less error checking. Thus, the medium sized files have similar logic to files of lower complexity, but spend more lines of code to catch errors.

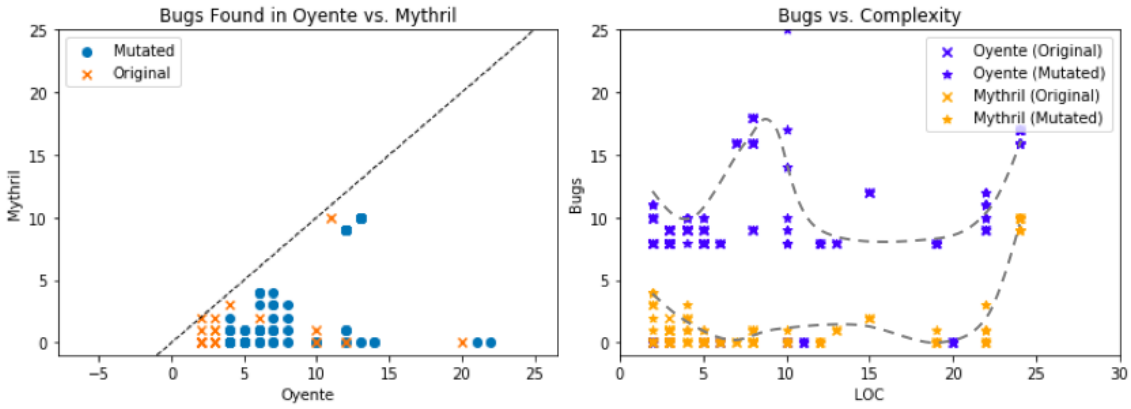

Figure 4: Plots of the number of bugs found with each suite (Left) and the number of bugs found as a function of the script's complexity (right). Here we see that Oyente consistently detects more bugs; more so with mutated scripts. Moreover, the number of bugs does not increase monotonically in complexity, but is bimodal where lowest and highest complexity lead to more bugs.

Moreover, after mutating the scripts, the number of bugs that Oyente found increased nearly 25 times, where Mythril found nearly the same number of bugs on average. This can likely be attributed to the types of bugs that Mythril searches for. With chain execution, Mythril misses bugs related to bad operations as the paths remain relatively unchanged. Interestingly, Mythril found many more issues in the mutated scripts, providing credence to the idea that Mythril treats mutations differently from Oyente, since Oyente found very few. In conclusion, we find that the Oyente testing suite significantly more bugs, but the Mythril tool provides significantly more details of the bugs.

6. FUTURE WORK

As it stands right now our tool makes use of Oyente and Mythril to analyze Solidity smart contracts then reports the bugs found in them. One of the goals of this project is to create a tool that not only allows Solidity developers to test their code in one, easy to use interface, but to also be adaptable to the user. For the tool to be adaptable, it must be simple for a developer to add more test-suites if they so wish. Therefore, to verify that this indeed can be easily performed, we hope to add at least one more test-suite to our tool, such as Manticore or Soligraph. Furthermore, we wish to expand on the experiments that we have already performed. Therefore, by the end of the quarter we will also extensively test the functionality of our tool with a larger number of more complicated Solidity contracts, with the intent of providing more in-depth feedback and comparison between the various tools and their effectiveness in finding bugs in smart contacts.

7. REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz, "Permacoin: Repurposing bitcoin work for data preservation," in *Security and Privacy (SP), 2014 IEEE Symposium on*, IEEE, 2014.

[3] "Use case for factom: The worldâĂŹs first blockchain operating system (bos).," 2015.

[4] D. Tapscott and A. Tapscott, *Blockchain revolution: how the technology behind bitcoin is changing money, business, and the world*. Penguin, 2016.

[5] N. Szabo, "The idea of smart contracts," *Nick SzaboâĂŹs Papers and Concise Tutorials*, vol. 6, 1997.

[6] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 254–269, ACM, 2016.

[7] P. Daian, "Analysis of the dao exploit," 2016.

[8] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: Platforms, applications, and design patterns," 03 2017.

[9] M. Alharby and A. van Moorsel, "Blockchain based smart contracts : A systematic mapping study," 08 2017.

[10] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," pp. 164–186, 2017.

[11] M. Everts and F. Muller, "Will that smart contract really do what you expect it to do?," tech. rep., TNO, 2018.

[12] "Manticore: Symbolic execution for humans," 2017.

[13] P. S. Kochhar, D. Lo, J. Lawall, and N. Nagappan, "Code coverage and postrelease defects: A large-scale study on open source projects," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1213–1228, 2017.

[14] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *New Technologies, Mobility and Security (NTMS), 2018 9th IFIP International Conference on*, pp. 1–5, IEEE, 2018.

[15] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[16] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 263–272, ACM, 2005.

[17] https://github.com/comaeio/porosity, "Porosity."

[18] R. Fontein, "Comparison of static analysis tooling for smart contracts on the evm," 2018.

[19] "Smartcheck automatically checks smart contracts for vulnerabilities and bad practices.."

[20] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.