Verifying Security Vulnerabilities for Blockchain-based Smart Contracts

Nedas Matulevicius
University of Manchester, UK
nedas.matulevicius@postgrad.manchester.ac.uk

Lucas C. Cordeiro
University of Manchester, UK
lucas.cordeiro@manchester.ac.uk

***Abstract*—In a modern world, aspects of cybersecurity become more of a requirement to software, systems, applications than just a feature implemented by programmers in their spare time. On the one hand, blockchain remains a pastime for people interested in digital currencies or decentralized, anonymous environments such as auctions or voting. On the other hand, cyberattacks are also not an exception to the blockchain community. Most of those attacks were made through smart contracts - pieces of code through which blockchain users interact with the actual blockchain. This paper analyses the background of blockchain technology, the implementation of smart contracts, and the cybersecurity aspect in the blockchain field. We describe an in-depth analysis of five static analysis tools (or code verifiers), their capabilities and drawbacks. These are tested with smart contracts with vulnerabilities deliberately included in their source code. The vulnerabilities are tailored so that they fit into the cybersecurity properties. After the implementation process, analysis is presented. We have found out which state-of-the-art static analysis tool is the most appropriate to secure the smart contract code from future cyberattacks on the blockchain.**

***Index Terms*—Blockchain, Cybersecurity, Static Analysis, Software Verification.**

I. Introduction

Blockchain technology nowadays tends to become more and more popular [1], with more people finding various interesting approaches and applications of blockchain, starting from decentralized forms of cryptocurrencies, such as Bitcoin [2] or Ethereum [3], ending with secure sensitive patient data transfer in healthcare institutions, Internet of Things (IoT) device management systems, and even voting systems [4]. To implement these ideas, one has to create a smart contract, which is a piece of code, written in some of the programming languages such as Solidity [5], also called a "block", which is then appended to the end of the whole system of other blocks chained together; thus, the name "blockchain"[1] [6], [7]. These blocks may contain any code written by a programmer, and anyone who has access to the blockchain can execute the code in the block. Furthermore, as the blocks cannot be altered or deleted from the blockchain under normal circumstances, the blocks act as a ledger for users to track interactions with blocks [8], named *transactions*. As a result, this feature provides some important security aspects to the technology, such as *integrity* and *transparency*. However, blockchain technology is not immune to cyberattacks. The main concern is the block's code verification before they are appended to the blockchain so adversaries cannot exploit it with malicious intent. Most of the attacks performed on blockchain were caused by abusing simple things that a programmer might have forgotten to implement, such as logical errors, uncaught exceptions, or even the classical buffer overflow problem [9], [10], [11]. However, smart contract programmers are not alone, as several verifiers are created to tackle the problem.

This paper aims to perform an overview of existing smart contract verifiers written in Solidity language and used for Ethereum smart contracts, finding out the most efficient and accurate static analysis tool for Ethereum smart contracts. The objectives for this paper are as follows. (1) *Writing various smart contracts as tests for verifiers to check their accuracy and efficiency.* This is the central aspect of the technical implementation part of this paper, as the analysis and statistics would be derived later from the smart contracts which deliberately have vulnerabilities in them. It has to be noted that many different vulnerabilities exist in smart contracts. However, only the ones that can cause a real cybersecurity threat to the user and/or system are analyzed here. (2) *Finding, using and adapting tests where applicable to various existing smart contract verifiers.* To perform analysis, several verifiers have to be used and tested out. It could be the case that a uniform smart contract vulnerability test might not fit all verifiers; therefore, the tests have to be tweaked for the verifiers to work while preserving the properties of the test. (3) *Performing benchmarking tests on verifiers.* In this paper, not only the accuracy of the verifier matters but also its performance, such as verification time, memory and CPU consumption rates and other parameters, which are essential for users. Lastly, (4)*performing analysis and statistics given benchmarks and verifier accuracy to derive conclusions.* Here, our goal is to conclude the most appropriate static analysis tool to verify security vulnerabilities in smart contracts.

We describe state-of-the-art static analysis tools, which could or are already being used in the industry as efficient and trustworthy tools for verifying smart contract code [12], [13], [14], [15], [16]. By performing an in-depth analysis of these tools, it is possible to derive some conclusions about their efficiency, accuracy, and reliability. Therefore, one can objectively choose one static analysis tool over another to detect some exact or all possible vulnerabilities in the code before deploying the blockchain. Moreover, as cybersecurity becomes more and more important in today's industry, the analysis performed here and its results would greatly help other people in determining what currently available tools are the best for particular vulnerabilities as well as deciding

[1]The original bitcoin white paper creator, Satoshi Nakamoto, mentions "chains of blocks", but not exactly "blockchain" [2].

where and what improvements have to be made in the field of verifying smart contract code.

Contributions. The main original contribution of this paper is the creation of Solidity smart contract tests containing security vulnerabilities and their verification with various static analysis tools designed for Solidity smart contracts. It is crucial to point out that the tests are written with cybersecurity properties in mind. Thus, the vulnerabilities and risks of tests were assessed and prioritized with established cybersecurity risk assessment methods [17], [18]. However, these methods are created for other programming languages, systems, programs, and frameworks. The general concepts inside the methods can be transferred to any programming language, program, or system, thus leading to one of our contributions.

Outline. This paper is divided into five distinct sections. The first part is the introductory one, explaining the aims and objectives of this paper. The second part summarizes the background theory needed for the paper, while the third part delves briefly into the technical part of our work. The fourth part discusses the data gathered from the research and shows the analysis of the results. Lastly, we summarize our work with notes about achievements and possible future work.

II. Verifying Security Vulnerabilities for Blockchain-based Smart Contracts

A. Background

A blockchain is a write-only list of data structures, called “blocks”, chained together [19]. On the blockchain, smart contracts are deployed so that the users can interact with the blockchain. Smart contracts, in this paper, are written with the Solidity programming language. In order to have a viable and fully functional product, say a system or a program, it needs to be bug-free, robust, and available at all times. In order to achieve this, various manual and automated testing methods are used; one of them is static analysis, e.g., lexical and dataflow analysis, symbolic execution (can be shortened to *symex*), and model checking [20]. The concepts from the cybersecurity field needed in this paper are the CIA triad [18] and the SEI CERT C Coding Standard’s risk assessment methodology [17].

B. Testing approach

The overall testing strategy and research methodology follow a *cyclic pattern* seen in Figure 1, as it involves testing each of the tests, described more in detail later in Section II-F. In order to describe one cycle of this pattern, the following methodology was used. (1) Find out a vulnerability that can be exploited through the Solidity smart contract code. (2) Write an example test containing the previously mentioned vulnerability. (3) Test the example with various static analysis tools and write the results under normal conditions. Finally, write down the outcome of the tool (vulnerability found/not found). (4) Repeat the test by reducing the number of threads or programs running on the system, write down the results under ideal conditions. (5) Repeat the test by introducing stress tests: 100% CPU consumption, 77% memory consumption, 90% memory consumption, and a full test: 100% CPU consumption and 90% memory consumption. For each of the stress tests, write down the results under each of the stress conditions.

This cycle is repeated for each of the vulnerabilities found and each of the static analysis tools used. Fig. 1 visually summarizes the overall testing strategy and research methodology of this paper.

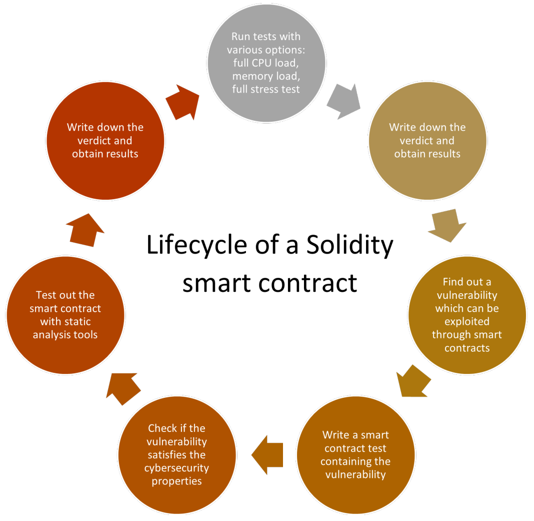
Lifecycle of a Solidity smart contract
Run tests with various options: full CPU load, memory load, full stress test
Write down the verdict and obtain results
Find out a vulnerability which can be exploited through smart contracts
Write a smart contract test containing the vulnerability
Check if the vulnerability satisfies the cybersecurity properties
Test out the smart contract with static analysis tools
Write down the verdict and obtain results

Figure 1: The lifecycle of a Solidity smart contract vulnerability.

Our research hypothesis is as follows: H1: *The more complex and technically powerful system or tool or program is slower and consumes more resources available to the system, i.e., CPU power, memory and/or disk capacity, etc. at the expense of system accuracy*. This hypothesis will be checked against in the analysis part of this paper, where it will be possible to derive if the hypothesis holds given the data gathered from the testing.

C. Static analysis tools

Here, various static analysis tools were considered for security testing purposes. However, we faced some issues – some tools were not maintained for a long time [21], [22], others did not work with newer Solidity compiler versions [23][2]. Some tools require a complex setup process, which may not be viable in commercial and industrial environments. Therefore, they were discarded from the testing process as well. Another important factor was tool availability. Some tools are not publicly available, and one has to buy a license to use the tool. Although this can be acceptable in the industry, these types of Solidity static analysis tools were not used for research purposes in this paper. Therefore, we are left with five Solidity static analysis tools, which satisfy *availability*, *usability*, and preferably *maintainability* requirements. The tools are as follows: Remix IDE static analysis plug-in [24]; Slither [13]; Oyente [14]; Mythril [15]; and SmartCheck [16]. These tools are written in different programming languages with different verification approaches – most of them employ

[2]A new Solidity major compiler version is a “breaking” one – e.g. a smart contract compiled with compiler version 0.6.0 might not work with compiler version 0.7.0 and vice versa.

already existing Satisfiability Modulo Theories (SMT) solvers, such as the Z3 SMT solver [25].

D. Tools and equipment

One laptop was used with technical parameters to simulate average working conditions and tools available in a commercial/industrial setting. The laptop has an Intel i7-3667U type CPU with four cores running at a clock speed of 2 GHz. The laptop has 8 GB of total available memory. However, 7.32 GB of RAM can be used for any purpose – the system consumes the remaining part. The laptop has Linux operating system (OS) with Ubuntu distribution, version 20.04. The disk capacity of the laptop is 180 GB. There was another PC with Windows OS in consideration for usage in this paper, but, as it turned out, most static analysis tools and other testing or benchmarking tools are not very friendly with Windows. Therefore, the idea of having several computers had to be scrapped.

For benchmarking, publicly available tools were used. To track resource consumption rates better, *htop* was used [26]. It is similar to the in-built *top* Linux command and is easily installable. While *top* already shows how much CPU or memory each thread or program uses, *htop* also produces graphs and gives better visual feedback to the user. For benchmarking and average script running time tracking, *hyperfine* command-line benchmarking tool was installed in the laptop and used throughout the paper [27]. The main advantage of *hyperfine* is that it allows benchmarking commands and running several benchmarks at the same time, saving time running a static analysis tool on each of the tests manually.

E. Performance metrics

In order to gather data and derive results, it was necessary to establish what qualities of the verification process would be measured. It is essential to find the most efficient and accurate static analysis tool. Therefore, the following performance metrics were chosen. (1) *Accuracy*: one of the most crucial requirements for any static analysis tool is its ability to accurately report the number of vulnerabilities in the Solidity smart contract code. A poor accuracy score means that either the static analysis tool is very simple and not fit for real-life use or it does not have to detect some types of vulnerabilities, as there are static analysis tools that specialize in some types of vulnerabilities, e.g., out-of-gas-vulnerabilities [28][29]. A good accuracy score shows that the static analysis tool is competent in reporting different bugs and can be used in business environments. (2) *Speed*: a fast static analysis tool increases the overall cybersecurity level of the blockchain, as people would be more compelled to use tools that take only a fraction of their time. (3) *CPU consumption*: efficient tools are essential so that their operations would not block other processes happening at the same time. A user would likely run a static analysis tool while browsing the Internet and having several programs and/or applications open or running in the background. (4) *Memory consumption*: The same argument mentioned about the CPU consumption applies here, although a high memory consumption rate for a prolonged period can have unwanted consequences, e.g., other programs or threads (or even the static analyzer itself) might be killed by the kernel due to insufficient memory, and in the extreme cases, the computer might crash. Therefore, it is desired that the static analysis tools would have low memory consumption rates to prevent errors and crashes.

F. Smart contract tests

There were thirteen different Solidity smart contracts created and used for our experiments. Each test was evaluated in terms of several cybersecurity concepts: whether the particular vulnerability violates one or more of CIA properties - *confidentiality*, *integrity*, and *availability*. Also, each smart contract is given a priority level according to the SEI CERT C Coding Standard risk assessment [17]. The tests are prioritized in severity levels: source code containing Level 1 (L1) security vulnerabilities are the most severe and require fixes and repairs as soon as possible, while Level 3 (L3) bugs can wait for their patching, and they have a low severity rating.

The following is the list of tests with vulnerabilities in them: Test 1 - canary test, no vulnerabilities; Test 2 - re-entrancy vulnerability; Test 3 - dead code vulnerability; Test 4 - weak PRNG and timestamp dependency vulnerabilities; Test 5 - lost contracts and unprotected self-destruction vulnerabilities; Test 6 - out-of-gas vulnerability; Test 7 - gas griefing vulnerability; Test 8 - usage of tx.origin for validation; Test 9 - unchecked value transfer and contract locking vulnerabilities; Test 10 - double constructor vulnerability; Test 10.1 - constructor-like functions; Test 11 - inline assembly code usage vulnerability; Test 12 - code size test, no vulnerabilities.

III. Evaluation and analysis

A. Description of benchmarks

Overall, all five static analysis tools were measured with all 13 tests[3] [4]. Each test was run 10 times with each static analysis tool; the average running times can be observed in Fig. 2. Thus, each static analysis tool was run 130 times - 650 test runs in total were done to obtain the data represented in Tables I to IX[5]. The number of runs for each test was not chosen for some specific reason – if one performs more runs for a single test, one can derive more accurate data with lesser error margins. Time and resource constraints for this project were decisive in choosing the amount of test runs; therefore, the optimal number of 10 runs for each test was chosen.

Fig. 2 illustrates the average running times for each test and each static analysis tool. The graph is logarithmic to improve the readability and understanding of outlier values produced by Oyente [14] and Mythril [15].

The benchmarks were divided into several parts. (1) Normal conditions (time and accuracy test) - all 13 tests were run under the ordinary operation of the laptop without any extra processor or memory load. The primary objective was to obtain average running times of code verification of each of the smart contract analyzers and to check the outcomes of static analysis tools. (2) Normal conditions (resource management

[3]Two tests are marked Test 10 and Test 10.1, indicating the similar properties of the source code, but are counted as different tests.

[4]The Github repository for these tests can be found here: https://github.com/nedasma/solidity-vuln-tests

[5]The detailed experimental results are available at https://drive.google.com/file/d/19DlLyRPr7zekhqR8C5Cgv5E-ezBGqlaM/view?usp=sharing

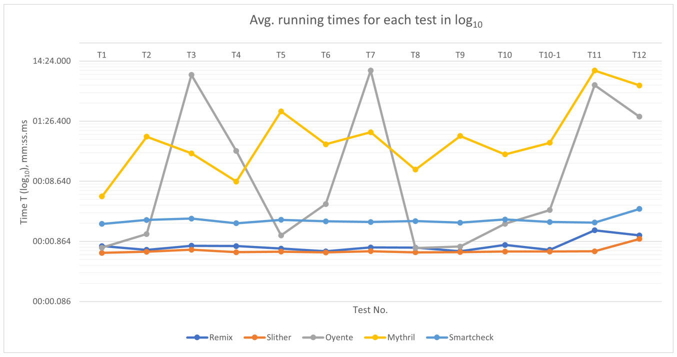

Figure 2: Avg. running times for each test in log-scale

test) - all 13 tests were run under the ordinary operation of the laptop without any extra processor or memory load. This time, the objective was to observe and document the CPU and memory consumption rates for each static analysis tool and each smart contract test. (3) Stress test 1 (maximum CPU load) - again, all 13 tests were run, but under the full load of the CPU cores. The objective was to observe and document the average running times for all smart contracts used for code verification. (4) Stress test 2 (77% memory load) - all tests were run under the 77% of RAM usage by executing several commands to increase the memory load artificially. The objective was similar to other tests - observe the behavior of analyzers and record running times. (5) Stress test 3 (90% memory load) - 13 tests were run under 90% of RAM usage. The objective is the same as for the stress test 2. (6) Stress test 4 (maximum CPU load and 90% memory load) - all tests were run under combined CPU and memory load, while still keeping the computer alive, as having 100% of memory load can cause crashes. The objective was to observe the static analysis tools, check for any abnormal behavior and get the average running times for each static analysis tool used with all source code of smart contracts verified.

B. *Summary of tests*

Overall, from the resource consumption perspective, Remix IDE plug-in [12] leads to the best CPU usage rates while Slither [13] manages memory usage; therefore, lower-end computers can handle both tools. Mythril [15] and SmartCheck [16] displayed the worst result in CPU rating since they consume the most memory on average. However, Oyente [14] has shown the most inconsistent results, meaning that it is prone to specific types of codebases, such as those containing lots of loops or contract sub-calls. In terms of robustness under stress conditions, Remix IDE plug-in [12] and Slither [13] showed the best results while having zero timeouts. The third-place goes to SmartCheck [16] because it had no timeouts, although stress conditions are not the ones SmartCheck [16] can deal with smoothly. Oyente [14] shows the worst performance with three timeouts and average longest-running times for CPU and memory stress tests.

C. *Setup*

The setup and execution process is very simple in order to run the tests. To start off, the equipment and tools used are explained in section II-D. To run a test with all static analysis tools, in the general case, one has to make four command statements in the Linux terminal and one statement online in the Remix IDE case. As for the example, we will take Test 2 as a placeholder value for static analyzers, and the source file of Test 2 code is called test2.sol. Note, that Oyente accepts only Solidity compiler versions 0.4.17 and lower, although it tends to work well with version 0.4.22 as well, while other tools and tests are written for mostly the newest compiler version (0.8.6 is the newest, but there are tests for compiler versions 0.7.0). Therefore, one has to change the Solidity compiler version in order to run Oyente properly. To do this, a tool called *solc-select* [30] is used to install and switch to other installed Solidity compiler versions. To switch to the required version supported by Oyente, run

```
solc-select use 0.4.17
```

where "0.4.17" is the Solidity compiler version. Also, the code has to be slightly modified in order to compile well with old Solidity versions, therefore, the test source code for Oyente are usually denoted with the name testName_cpy.sol, where *testName* is the name of the test, e.g. "test2".

The following list explains what commands should be run in order to verify the source code of Test 2 with each of the static analysis tools: (1) Remix IDE static analysis plug-in: click "compile" on test2.sol when in the "Solidity compiler" section, then click "run" after selecting the "Solidity static analysis" section. Uncheck the "autorun" property if it was selected in the first place to run the compiler and the analyser separately or keep it checked if both compilation and analysis are wanted to be done in a single "compile" click. (2) Slither: *slither test2.sol*. (3) Oyente: *oyente -s test2_cpy.sol*. (4) Mythril: *myth analyze test2.sol*. (5) SmartCheck: *smartcheck -p ./test2.sol*.

However, doing this is time-consuming, especially when there are 13 tests to be done. Therefore, *hyperfine* comes to help by automating the runs and performing time benchmarks on each of them. Unfortunately, some tests are written for specific compiler versions, so by changing the versions one might not be able to verify the tests correctly. Therefore, Table I shows the groupings of tests by Solidity compiler versions. Note that this table does not apply to Oyente in most cases because its tests were slightly adapted to comply with the newest acceptable version by Oyente. Neither the overall existence of the vulnerability nor the logic in the code changes in these modified files, which can be recognized by having a _cpy in the file name at the end. To run hyperfine in the Linux terminal, write:

Table I: Test groupings by Solidity compiler version

Solidity compiler version	Test numbers
0.8.6	1, 7, 10.1, 11
0.7.0	2, 3, 5, 6, 8, 9
0.6.0	4
0.4.22	10

```
hyperfine -L version 1,7,10.1,11 'slither test{
    version}.sol' -i
```

Here the tests would be run for the Solidity compiler version 0.8.6, the analysis of tests would be done by Slither, and any output by the analyzers is suppressed by the -i flag in order to have *hyperfine* working properly. To adjust the command, use different version numbers explained in Table I and other static analysis tools. For stress testing, the *stress* package is used, and it allows filling up the memory or consume a lot of processing power. To obtain 100% of CPU usage, run

```
stress --cpu 4 -t 60
```

This will keep the CPU at 100% of usage for 60 seconds on all CPU cores, which are four of them in the testing laptop. For obtaining around 77% of memory consumption, run

```
stress --vm-bytes $(awk '/MemAvailable/{printf ''\%d
    \n'', $2 * 0.99;} ' < /proc/meminfo)k --vm-keep
    -m 1 -t 60
```

This will keep the memory busy at around 77% of its capacity for 60 seconds. In order to get 90% or more of RAM usage, one can use the following command on the terminal:

```
sudo </dev/zero head -c 7000m | pv -L 500m | tail
```

This will use around 7 GB of memory[6], with each tick increasing the usage by 500 MB. The *pv* command gives a nice visualization to the programmer only. However, one must be aware that using very large amounts of memory might trigger the kernel's out-of-memory killer (OOM), therefore, in order to keep the stress test command alive and not killed by the kernel, use the following command:

```
sudo echo -1000 > /proc/procNo/oom_score_adj
```

where *procNo* is the process ID of the stress command. The ID with the benchmarks for resource management of the tools can be found in the *htop* interactive process viewer.

D. Objectives

The objectives of this testing are as follows. (1) Test out all five static analysis tools in question - namely: Remix IDE static analysis plug-in [12], Slither [13], Oyente [14], Mythril [15], and SmartCheck [16] and see if they work correctly. (2) If point one is satisfied, run all designed tests with vulnerabilities and find out the accuracy rating of each static analysis tool. (3) Find out if there are any tests whose vulnerabilities cannot be caught by any of the static analysis tools. (4) Observe the performance of the analyzers under normal conditions and stress conditions.

From these main objectives described, it is possible to derive which static analysis tool is the best in terms of accuracy and performance when finding security bugs in Solidity smart contracts. The summary section of this section in section III-G gives an overview of the results and achieves the main goals laid out in section I.

E. Results

The analysis of the results presented in Section III-A can be seen in three different perspectives: from the accuracy point of view, from the resource usage point of view and combined. However, remembering the hypothesis given in Section II-B requires us to look at a combined approach – whether there is a possibility to find a correlation between accuracy, complexity and efficiency of static analysis tools. First of all, it is crucial to look at the accuracy Table III displayed in the separate document[7] and compare it to the results obtained in Fig. 3 of this document.

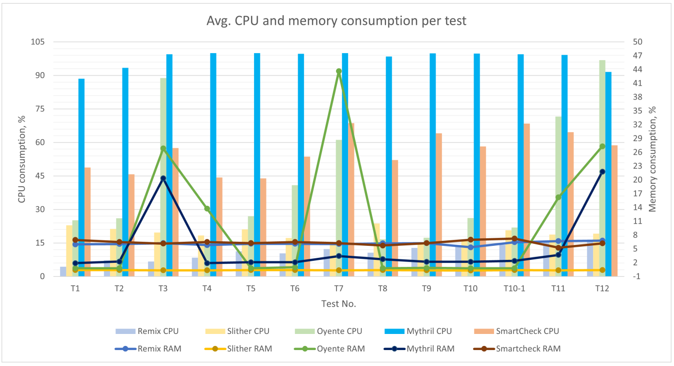

Figure 3: Avg. CPU and RAM usage per test

As an example, consider Remix IDE graph part in Fig. 3 of this paper, colored in lighter blue, showing how CPU and memory consumption as well as running time changes from Test 1 to Test 12 for Remix IDE static analysis plug-in. It can be seen that the upper bound for CPU usage increases almost linearly for each test, while the lower bounds stay pretty much the same at 3-5% of CPU consumption. It is also shown that the memory usage for all tests stays quite consistent, although the upper bound for Test 10 is an outlier. Comparing that to the accuracy table, we cannot determine any patterns between accuracy and consumption, as Test 3 requires less computing power than, say, Test 10, but both are marked that the analyzer could not find any vulnerabilities in both of them. The reverse argument is also valid by comparing Tests 4 and 5. What can be determined, though, is the relation between inline assembly code in Test 11 - the CPU consumption rises to 28%. At the same time, the upper bound for test verification time illustrated in Fig. 2 has also increased significantly compared to other tests.

As for the accuracy rating, Remix IDE static analysis plug-in finds 8 out of 12 security vulnerabilities in Solidity smart contracts correctly, therefore getting the accuracy rating of 66.67%. The low memory usage, processing power consumption, and fast verification process put the plug-in in a high place for the competition to find the best static analysis tool for Solidity smart contracts.

As a slight jump back to the performance analysis, we established that Test 11 was verified incorrectly by the Remix IDE static analysis plug-in; therefore, it would be interesting to look at Slither, which correctly identified inline assembly code in the source code of the test in question. Compared to Remix IDE plug-in's CPU consumption patterns, Slither seems to have a completely different graph, as illustrated in Fig. 3 of this document. Here, the largest upper bound was observed in Test 3 with the CPU usage nearing 30%. On the other hand, the memory usage, seen in the same graph, is consistent, similarly to Remix plug-in. Fig. 2 illustrates

[6]6.83 GB to be more exact in the binary base.

[7]The corresponding details can be found in the footnote of Section III-A.

that Slither runs consistently regardless of the test – the only exception being Test 12, which is effectively a code size test. However, this shows that inline assembly does not affect resource consumption or speed of the static analysis tool because, for Slither, Test 11 is verified as consistently in terms of metrics discussed as other tests. It is also worth noting that Slither correctly identified inline assembly usage in the source code of Test 11. Overall, Slither tends to consume less memory than Remix IDE static analysis plug-in, while the CPU usage is quite similar. However, Slither is way more consistent in terms of average running times, as illustrated in Fig. 2.

Slither's accuracy rating is the best of all static analysis tools tested in this paper, with a rating of 75% achieved by the analyzer. Combined with robust performance, constant running times and average resource management, it aims to be one of the best static analysis tools to date.

Let us look at Oyente. As we can see from the CPU and memory consumption graph in Fig. 3 of this document, colored in green, the problematic tests (Test 3, 7 and 11) stand out in the graph as having the upper bound at 100% CPU usage. As one of these tests (Test 7) did not finish successfully and it timed out after exceeding the threshold limit of 10 minutes, it can be assumed that longer running times can correlate with higher CPU usage for Oyente static analysis tool. For comparison, the code of Test 3 was verified for about 8 minutes, while the code for Test 11 took about 5 minutes and 30 seconds. This can be seen both in Fig. 2, showing the visual "spikes" for Tests 3, 7 and 11. The memory consumption also indicates the problematic tests as visual "spikes" in Fig. 3 of this document. Therefore, in Oyente's case, one can conclude that a longer verification process leads to abnormal resource usage by the static analysis tool. If we let it run for a prolonged period, the kernel would likely kill the analyzer's process, making the verification process useless. It also needs to be emphasized that Oyente is an old static analysis tool, with its first incarnation released to the public over 5 years ago [14], where blockchain technology and Solidity programming language were still in their infancy. Therefore, it is only natural that currently, there are better optimized, faster, and more accurate static analysis tools than Oyente. Oyente's accuracy rating is the lowest of all 5 static analysis tools tested – only 3 out of 12 security vulnerabilities were identified correctly by Oyente, giving the accuracy percentage of 25%.

Mythril is a bit newer static analysis tool than Oyente. Thus, it is expected that it will outperform Oyente on several metrics, and this statement is correct regarding running time. However, as we can see in Fig. 3 of this document, colored in cyan, Mythril takes a greedy approach to resources, especially the computing power of the processor. Mythril verifies the tests almost in all cases by using 100% of CPU power, which is not the best outcome if, e.g., a user has other CPU-intensive processes running. In that case, the computer would slow down and all processes, including Mythril's static analysis. However, memory usage is sustainable, although both Slither and Remix IDE plug-in use less memory than Mythril counterpart. It is also essential to note that Mythril, on average, takes more time to verify Solidity smart contracts, a perk not seen in other static analysis tools, including SmartCheck, which results will be explained later in this section. For example, Test 5, concerned with unprotected self-destructs, and arbitrary data sends. The running time bounds, which can be seen in Fig. 2, takes about two-and-a-half minutes for Mythril to verify the source code, while all other tools take a maximum of 2 seconds to complete the analysis. Furthermore, Mythril does not handle inline assembly code in Solidity smart contracts – the static analysis of Test 11, a test related to the security vulnerability in question, timed out after running for 10 minutes.

Mythril's accuracy is below average, with a 41.67% accuracy rating (5 out of 12 vulnerabilities identified correctly). The tool tries to compete with Remix IDE plug-in and Slither but is underperforming both in resource management and accuracy. The largest drawback is probably speed - on average it runs the longest compared to all other tested static analysis tools.

The performance of SmartCheck, in terms of resource management, is satisfactory considering that the analyzer is written in Java, which means that code compilation, translation processes and garbage collection take more memory resources in general. However, SmartCheck handles the memory consumption very well - fig. 3 of this document (colored in light brown) shows that the static analysis tool does not reach 10% of available RAM for use. On the other hand, the same figure shows large averages between CPU consumption rates, sometimes the range reaches 60 per cent. This tells us that during the execution process the CPU usage increases at a very large rate until it reaches 100% of consumed processing power. The running times for SmartCheck, seen in fig. 2, are quite consistent and the graph resembles Slither's part in the same figure. Unfortunately, the tool is not very accurate - the accuracy of SmartCheck is only 25 per cent, and it does not show any signs of being different to other tools in terms of vulnerability catching process, that is, if other analyzers do not catch a particular vulnerability, it is very likely that SmartCheck will not catch it as well, therefore the tool is not very suitable for cross-checking the source code for additional vulnerabilities.

F. Threats to validity

The experiments carried out with the static analysis tools for Solidity smart contracts are accurate and done consistently to prevent bias in the results gathered. However, the experiments and evaluations are not perfect, and there is always room for improvement. First, there are 13 tests written in total; 11 of them have security vulnerabilities. While the aim was to create the tests that contain the most common security vulnerabilities w.r.t. cybersecurity properties, the list of vulnerabilities is by no means exhaustive. For example, the SWC registry contains 36 different vulnerabilities. However, not all of those weaknesses can cause cyberattacks or attacks, which can cause actual damage - one of the examples would be the floating pragma (Solidity compiler version). Also, there is the case of "known unknowns" – we know that security vulnerabilities exist, but we do not know how many of them exist on the live blockchain. Therefore, the limit of tests to be included in the experimentation is not bound to a specific number. If there are any new vulnerabilities discovered, likely, the static analysis tools will not catch them. Moreover, one should not discard the possibility of false positives as well.

Continuing the narrative of the tests used for the experiment, the accuracy rating can change if more or fewer tests are included in the testing phase. It can be also be perceived that the inclusion of more tests can drop the accuracy rating for all analyzers. However, the rating will be more correct as more cases would be included in the evaluation process. The reverse action also has consequences - having fewer tests might distort the accuracy rating, disproportionally inflating or reducing the rating in question. The improvement here can be twofold: (1) Include more tests with different vulnerabilities, especially if the tests have code that deviates from the standards given by the documentation of static analyzers. This will probably reduce the overall accuracy rating, but better conclusions can be derived on which tool is more versatile and can catch different types of security bugs. (2) Reduce the number of tests but make them more specialized. For example, suppose one is particularly interested in re-entrancy vulnerabilities. In that case, one can create ten different tests having re-entrancy vulnerabilities with different code snippets and various code obfuscation methods applied. Including canary tests (the tests without vulnerabilities) with code snippets, which can look like the ones having the re-entrancy bug, but there are no vulnerabilities there that would improve the testing strategy. The specialized tests would allow determining whether the tool is truly capable of detecting the security bug in question and checking for any false positives that might occur during the code verification process.

The last issue is resource management monitoring. Even if the experiment is replicated identically to the one presented in this paper, there is a margin of varying running times or CPU or memory consumption. However, this problem should fall into the category of allowed deviations from the given results. The real issue can arise if the experiment is done on other computers with different technical parameters, OS types and/or versions. It should be thought that having a more powerful machine than the one used here will show better results for static analysis tools in the benchmarks and vice versa. However, the verdict given by a static analysis tool (verified code contains vulnerabilities or not) should be consistent regardless of the computer technical parameters.

G. Final remarks

First, let us get back to our research hypothesis established in Section II-B. According to Section II-C, we can conclude that while Slither is one of the most sophisticated tools tested. With the level of complexity and the capabilities of the analyzers, the only comparable ones are Oyente and Mythril, Slither does not consume significantly more resources or runs slower on average than others, e.g., more straightforward static analysis tools such as the plug-in for Remix IDE. The hypothesis, therefore, *does not hold*. However, it should be noted that if one discards Slither from the analysis and substitutes it with the other Solidity static analysis tool, the hypothesis could hold because Oyente and Mythril were by far the most resource-hungry and slowest static analysis tools. Thus, the statement that more sophisticated and more extensive tools are inefficient is incorrect because as the technology moves forward, better static analysis methods are created, achieving faster verification rates while retaining low resource consumption rates. If this paper was conducted several years ago, one could argue that the present day's technology has reached its limits. However, now, we can see that there are static analyzers that are sophisticated yet powerful, accurate and efficient. This moves to the paper's main aim - what is the best static analysis tool for Solidity smart contracts, which can find the most with cybersecurity-related vulnerabilities? The answer is pretty straightforward if we are looking at the data presented here.

Slither is by far the best of five tested tools, as its accuracy rating is the highest, and the resource management rating is also one of the best. Also, it is quick and robust - even under a comprehensive stress test, Slither managed not to exceed 4 seconds of verification time. The second place would go to Remix IDE static analysis plug-in, which can be unexpected since its implementation is relatively simple compared to other tools. However, it detects various cybersecurity-related vulnerabilities, which is a good sign for Solidity programmers using Remix IDE. Also, the resource management is in good standing, as well as the execution times. It is by far the fastest static analysis tool available out of all tools tested here, but it gets behind Slither only because it is less accurate.

The third place should go to Mythril, shared with SmartCheck - although being very greedy for resources, its accuracy rating is somewhat reasonable. However, it falls behind the Remix plug-in by 20% points. Of course, resource management can permanently be remedied by putting more resources into a machine, but this is a Turing-completeness problem - theoretically speaking, all programs will terminate if the computer would have infinite amounts of memory, which is physically impossible. Thus, while being bound to finite amounts of computing power and memory, it is essential to look at the resource management of a program, and Mythril, unfortunately, is not the best in this field. However, if the resources are abundant, the accuracy rating can be a reasonable choice for finding security bugs in Solidity smart contracts. Also, another drawback is timeouts – a property no one would enjoy having.

SmartCheck receives third place, sharing with Mythril due to its low accuracy rating – one of the main factors for a static analyzer to be considered a "good" tool. However, SmartCheck does not consume many resources and does not time out; this is why SmartCheck is on par with Mythril. Had the accuracy rating been better for SmartCheck, then it would have a guaranteed third place, moving Mythril to the solid fourth place. Overall, SmartCheck is a good choice for lightweight smart contract checking, but it must be noted that it is not a very strong tool and might need cross-verifying to catch all possible vulnerabilities in code.

The last place goes to Oyente due to several factors. First of all, the accuracy rating is one of the lowest, comparable to SmartCheck. Secondly, it is prone to code explosion, meaning that it is unsuitable for smart contracts, which have a large codebase or contain many loops. Lastly, because it is affected by the disadvantage mentioned previously, the resource management suffers, thus distorting the overall performance results. Suppose Oyente would be updated, and more efficient methods of traversing CFGs were implemented, from which logical statements are fed into Z3 SMT solver. In that case, Oyente can once again become a powerful static analysis tool for Solidity smart contracts as it was 5 or 6 years ago.

IV. Conclusion

One of the main contributions of this paper is the compilation and evaluation of Solidity smart contract tests, which fit into the cybersecurity properties. As there are not many surveys or analyses carried out, at least to the authors' knowledge, where smart contracts with security vulnerabilities had risk assessments or comparisons to the CIA triad, this is a significant achievement unique to this paper. Furthermore, the experimental results we have obtained by running the custom smart contracts developed here give some critical insights about the capabilities of the static analyzer, including their accuracy ratings and their performance under various security conditions. From that data, we can derive which static analysis tools are more competent in specific types of security vulnerabilities and which ones are more universal. Also, the data obtained is valuable in determining the user's priorities wanting to use the analyzer, e.g., whether the accuracy or the speed is the primary requirement. As a result, we lay the groundwork for performing analysis on Solidity smart contracts in more specialized fields – cybersecurity. Some surveys discuss the cybersecurity behind the attacks on smart contract vulnerabilities [31], [32], [33], [34]. However, they are either quite abstract in terms of the cybersecurity properties provided, only the popular examples are given to the reader, or the vulnerabilities are discussed in detail, but they lack exact properties such as risk assessment or CIA triad property evaluation. For future work, we are extending the ESBMC tool [35] to perform in-depth security analysis for Solidity smart contracts from the cybersecurity perspective.[8]

References

[1] M. Risius and K. Spohrer, "A blockchain research framework," *BISE*, vol. 59, no. 6, pp. 385–409, 2017. [Online]. Available: https://link.springer.com/content/pdf/10.1007/s12599-017-0506-0.pdf

[2] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. bitcoin.org. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[3] V. Buterin. Ethereum whitepaper. ethereum.org. [Online]. Available: https://ethereum.org/en/whitepaper/

[4] S. Daley. 30 blockchain applications and real-world use cases disrupting the status quo. Built In. [Online]. Available: https://builtin.com/blockchain/blockchain-applications/

[5] About solidity. Solidity Team. [Online]. Available: https://soliditylang.org/about/

[6] J. Mattila, "The blockchain phenomenon – the disruptive potential of distributed consensus architectures," Helsinki, ETLA Working Papers 38, 2016. [Online]. Available: http://hdl.handle.net/10419/201253

[7] T. Caradonna, "Blockchain and society," *Informatik Spektrum*, vol. 43, pp. 40–52, 2020.

[8] D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain technology overview," *CoRR*, vol. abs/1906.11078, 2019.

[9] Most common smart contract vulnerabilities. Hacken OU. [Online]. Available: https://hacken.io/education/most-common-smart-contract-vulnerabilities/

[10] M. Pramesh. Most common smart contract bugs of 2020. A Medium Corporation. [Online]. Available: https://medium.com/solidified/most-common-smart-contract-bugs-of-2020-c1edfe9340ac

[11] M. Gogan. Smart contract security: What are the weak spots of ethereum, eos, and neo networks? TechNative. [Online]. Available: https://bit.ly/3jYIefw

[12] Remix analyzer. Remix Team. [Online]. Available: https://github.com/ethereum/remix-project/tree/master/libs/remix-analyzer#how-to-use

[13] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd Intl. WETSEB*, 2019, pp. 8–15.

[14] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. 2016 ACM SIGSAC CCS*, ser. CCS '16. New York, NY, USA: ACM, 2016, p. 254–269.

[15] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," 2018.

[16] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proc. 1st Intl. WETSEB '18*, ser. WETSEB '18. New York, NY, USA: ACM, 2018, p. 9–16.

[17] How this coding standard is organized - risk assessment. Carnegie Mellon University, Software Engineering Institute. [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c/How+this+Coding+Standard+is+Organized\#HowthisCodingStandardisOrganized-RiskAssessment

[18] S. Samonas and D. Coss, "The cia strikes back: Redefining confidentiality, integrity and availability in security." *JISSec*, vol. 10, no. 3, 2014. [Online]. Available: http://www.proso.com/dl/Samonas.pdf

[19] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck, "Blockchain," *BISE*, vol. 59, no. 3, pp. 183–187, 2017.

[20] P. Li and B. Cui, "A comparative study on software vulnerability static analysis techniques and tools," in *2010 IEEE ICITIS*, 2010, pp. 521–524. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=5689543

[21] I. Nikolic. Maian. [Online]. Available: https://github.com/ivicanikolicsg/MAIAN

[22] E. Rafaloff. Solanalyzer. [Online]. Available: https://github.com/EricR/solanalyzer

[23] C. Peng. Sif (solidity instrumentation framework) - issue no. 6: terminate called after throwing an instance of 'nlohmann::detail::out_of_range'. [Online]. Available: https://github.com/chao-peng/SIF/issues/6

[24] Remix - ethereum ide. remix-project.org. [Online]. Available: https://remix.ethereum.org/

[25] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[26] htop - an interactive process viewer. htop Team. [Online]. Available: https://htop.dev/

[27] hyperfine - a command-line benchmarking tool. The hyperfine developers. [Online]. Available: https://github.com/sharkdp/hyperfine

[28] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Gasol: Gas analysis and optimization for ethereum smart contracts," in *TACAS*, A. Biere and D. Parker, Eds., 2020, pp. 118–125.

[29] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, Oct. 2018.

[30] solc-select: A tool to quickly switch between solidity compiler versions. Trail Of Bits. [Online]. Available: https://github.com/crytic/solc-select

[31] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys*, vol. 53, no. 3, Jun. 2020.

[32] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust*, M. Maffei and M. Ryan, Eds., 2017, pp. 164–186.

[33] P. Praitheeshan, L. Pan, J. Yu, J. K. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities:A survey," *CoRR*, vol. abs/1908.08605, 2019.

[34] Z. Wang, H. Jin, W. Dai, and K.-K. R. Choo, "Ethereum smart contract security research: survey and future research opportunities," vol. 15, 2020.

[35] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "ESBMC 5.0: An industrial-strength C model checker," in *33^{rd} ACM/IEEE Int. Conf. on Automated Software Engineering (ASE'18)*. New York, NY, USA: ACM, 2018, pp. 888–891.

[8] https://github.com/kunjsong01/esbmc/tree/dev-solidity-support/src/solidity-frontend