University of Manchester

Department of Computer Science

Verifying Information Flow Security for Blockchain-based Smart Contracts

A dissertation submitted to The University of Manchester
for the degree of Master of Science
in the Faculty of Science and Engineering

Nedas Matulevicius

10145602

2021

Contents

List of Figures

List of source codes

List of Tables

Abstract

In a modern world, aspects of cybersecurity become more of a requirement to software, systems, applications etc. than just a nice feature implemented by programmers in their spare time. Blockchain, on the other hand, still stays as a pastime to people interested in digital currencies or decentralised, anonymous environments such as auctions or voting. However, cyberattacks are also not an exception to the blockchain community, and most of those attacks were made through smart contracts - pieces of code through which blockchain users interact with the actual blockchain. This project analyses the background of blockchain technology, implementation of smart contracts and the cybersecurity aspect in the field of blockchain. The project presents an in-depth analysis of five static analysis tools (simply put - code verifiers), their capabilities and drawbacks and these are tested out with test 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, and it is found out which static analysis tool is the best in order to secure the smart contract code from future cyberattacks on the blockchain.

Declaration

No portion of the work referred to in the dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to thank my family and friends who supported me throughout these exceptional times when the project was undertaken. A special thanks go to the project supervisor, Dr Lucas Cordeiro, whose in-depth academical knowledge helped greatly in completing this project successfully. I would also like to personally thank Kunjian Song, who provided interesting insights about smart contracts and blockchain technology.

Chapter 1

Introduction

Blockchain technology nowadays tends to become more and more popular [4], with more people finding various interesting approaches and applications of blockchain, starting from decentralised forms of cryptocurrencies, such as Bitcoin [5] or Ethereum [6], ending with secure sensitive patient data transfer in healthcare institutions, Internet of Things (IoT) device management systems, and even voting systems [7]. 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 [8], 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] [9, 10]. 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. 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 [11], named transactions. This provides some important security aspects to the technology, such as integrity and transparency, but blockchain technology is not immune to all kinds of cyberattacks. The main concern is the verification of the code in the blocks before they are appended to the blockchain so that they cannot be exploited with malicious intent by adversaries, and 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 and buffer overflow [12][13][14]. However, smart contract programmers are not left alone, as there are several verifiers created to tackle the problem.

1.1 Aims and objectives

The main aim of this project is to perform an overview of existing smart contract verifiers written in Solidity language and used for Ethereum smart contracts, finding out the most efficient as well as accurate static analysis tool for Ethereum smart contracts. The objectives for this project are as follows:

- *Writing various smart contracts as tests for verifiers to check accuracy and efficiency.* This is the main aspect of the technical project imple-

[1]The original bitcoin white paper creator, Satoshi Nakamoto, mentions "chains of blocks", but not exactly "blockchain" [5].

mentation part, 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 there are lots of different vulnerabilities existing in smart contracts, but here only the ones which can cause a real cybersecurity threat to the user and/or system are analysed.

- *Finding, using and adapting tests where applicable to various existing smart contract verifiers.* In order 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 a little bit for the verifiers to work while preserving the properties of the test.

- *Performing benchmarking tests on verifiers.* In this project, not only the accuracy of the verifier matters but also its technical performance, such as verification speed, memory and CPU consumption rates and other parameters which are important for users.

- *Performing analysis and statistics given benchmarks and verifier accuracy to derive conclusions.*

1.2 Motivation

This project was designed and undertaken with several outcomes in mind. Firstly, the project allows learning about the 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. By performing an in-depth analysis of the 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 in order to detect some exact or all possible vulnerabilities in the code before the deployment on the blockchain. Moreover, as cybersecurity becomes more and more important in today's industry, the analysis performed in this project and its results would greatly help other people in determining what currently available tools are the best for particular vulnerabilities as well as deciding where and what improvements have to be made in the field of verifying smart contract code.

1.3 Contributions

The main contribution of this project is the creation of Solidity smart contract tests containing 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 in tests and their risk were assessed and prioritised with established cybersecurity risk assessment methods. However, these methods are created for other programming languages or systems, programs, frameworks etc. but the general concepts inside the methods can be transferred to any programming language or a program or a system - this was done in this project.

1.4 Structure and organisation

This project is divided into five distinct parts. The first part is the introductory one, explaining the aims and objectives of this project. The second part covers the background theory needed for the project, while the third part delves into the technical part of this project - how the project was implemented. The fourth part discusses the data gathered from the research and shows the analysis of results. Finally, the last part summarises the project with notes about achievements, reflection and possible future work.

Chapter 2

Background

In this chapter, the core background theory is covered in several sections in order to obtain a better understanding of the concepts used throughout the project. The main concepts include blockchain technology, its usage, advantages and flaws, smart contracts, code verification via static analysis and the cybersecurity aspect of this project.

2.1 Blockchain technology

A blockchain is a write-only list of data structures, called “blocks”, chained together [15]. Effectively, blocks can contain any information inside as the technology is not bound to specific applications, for example, cryptocurrency. Each block contains cryptographic hashing functions and a timestamp, which provide the user required Confidentiality, Integrity and Availability (CIA) [16] properties needed to ensure that a block has not been altered [17]. Therefore, one can be certain that the interaction with the blocks in the blockchain is secure in terms of data management. In order to submit a block to a blockchain, it has to be approved first by the (majority, most of the time) blockchain community, and this concept is called a *consensus mechanism* [18]. As Tim Swanson explains in his report on distributed ledger systems, “it is a set of rules and procedures that allows maintaining coherent set of facts between multiple participating nodes” [19]. This shows that block appending to the blockchain is not a fully automatic process - it needs to be approved first.

There are two popular and widely used blockchain platforms in the public - Bitcoin and Ethereum. While both employ similar technological solutions [20], we are mainly going to discuss Ethereum, its smart contracts and verification methods. Ethereum blockchain contains:

1. states, which are mappings between addresses and account states [21], which in turn can be either externally owned or contract accounts [20],

2. transactions, which are cryptographically signed instructions, containing a nonce (a value, usually a number, which ensures the user that the received signature is fresh), gas price and value (discussed later in this chapter), the address of the instruction recipient and the value to be sent in Wei [21], which is a unit of value measurement - a good analogy would be a

real-life currency: an equivalent of pounds and pence would be Ether and Wei,

3. blocks, which contain encrypted information about the parent block, the current block, timestamp acquired when the block was created and other required information. The information about other blocks stored in the current block is encrypted with the 256-bit Keccak hash [21].

Each transaction performed requires some computational power, and it does not come for free. Therefore, the concept of *gas* and gas consumption is introduced in the blockchain technology. Each computational operation in Ethereum blockchain consumes gas, which is generally 1 gas per 1 computational operation [20], although prices vary and transactors can define their gas price per transaction [21]. The concept of gas is useful for miners, who verify the transactions, as they receive monetary incentives depending on computational power [22]. Also, gas consumption partially prevents Denial-of-Service (DoS) attacks [22]. However, gas can run out during the computation process, and improper handling of these exceptions can lead to vulnerabilities, which can be exploited by adversaries [23].

Ethereum blockchain, like Bitcoin, is well-suited for building digital economies [24], but one can build any systems they like. These systems are called *decentralised applications* (DApps). Most of the DApps are built as mobile applications, as in this example [25], but they can be web-based as well.

2.2 Smart contracts

A smart contract is a closely related term to the technology of blockchain, as the transactions occurring in the blockchain need to be formalised so that people can establish trust not only in the system but also between parties participating in the system. Therefore, a smart contract is a set of rules and protocols, which are deployed in the blockchain to verify and validate the transactions between users [26]. The users do not interact directly with the blockchain and the blocks inside it, but with the smart contracts, which are deployed and verified[1]. However, the code in the smart contracts needs to be verified, so that users may not be able to exploit the contracts and, in turn, cause financial and/or other damage to users of the blockchain. Unfortunately, this is not strictly enforced by the blockchain, as it does not have any implementations which would protect the blockchain from adding malicious smart contracts. One survey showed that through exploiting vulnerabilities in the smart contracts, adversaries managed to incur large financial losses, with one DoS managing to create 280 million US Dollars of financial damage [27]. The famous DAO attack was based on a logical error of implicit fallback function calling [28] created by the programmer, which could have been easily avoided by a code verifier - a static analysis tool.

Ethereum smart contracts written in Solidity programming language can be analysed and verified with static analysis tools, just like other programming languages, such as C/C++ [29] or Java [30]. Several static analysis tools exist for Ethereum smart contracts with varying degrees of capability and scope [31]. Some are general-purpose static analysis tools, such as Vandal [32], Slither [33]

[1]See figure 2.1.

and Zeus [34], while others are more specific, for example, MadMax [23] and GASPER [35] for out-of-gas vulnerabilities, Ethainter for composite information flow vulnerabilities [36] or VerX for satisfying functional specifications [37]. Some of these tools will be used in the project to test their capabilities as well as test their speed and resource consumption rates.

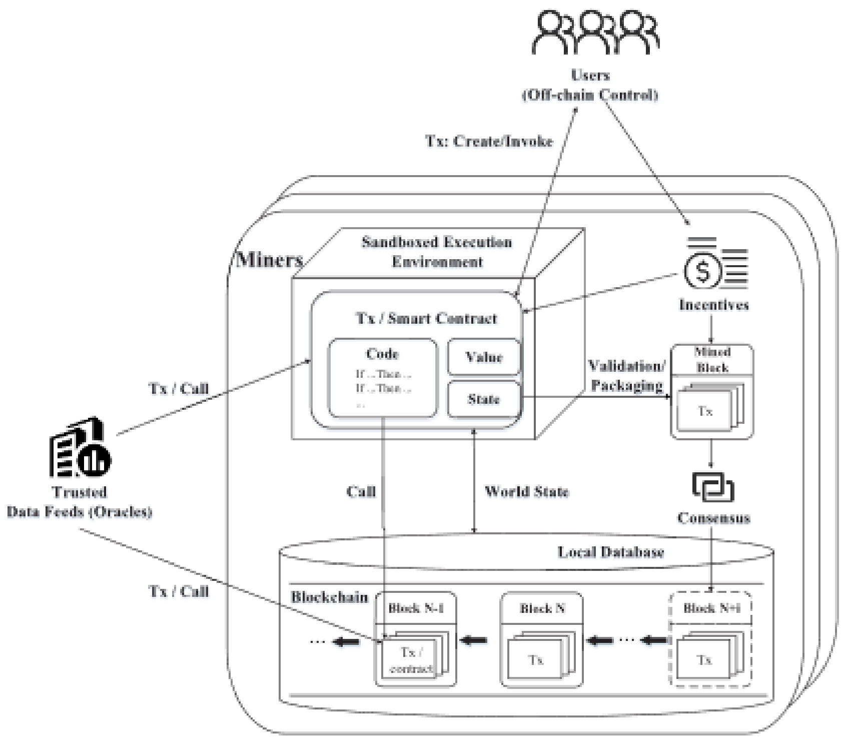


Figure 2.1: General operation of the smart contract in a blockchain[26]

2.3 Static analysis and verification of programs

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. To achieve this, various manual and automated testing methods are used:

- *manual methods*, also called informal methods, such as desk checking, walkthrough, code reviews, inspections and audits [38];
- *automated methods*, which could be further categorised into the following subcategories:
  - *Static analysis* - e.g. lexical and dataflow analysis, symbolic execution (can be shortened to symex) and model checking [39];

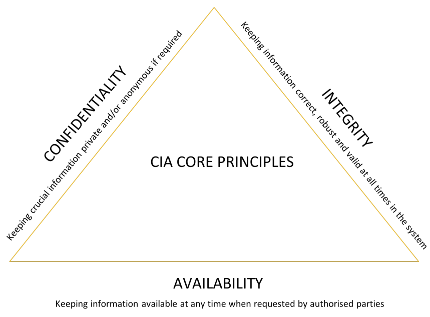
– *Dynamic analysis* - e.g. top-down and bottom-up testing strategies, white-box and black-box (mutation-based, grammar-based or random) fuzzing techniques, automatic debugging, stress testing [38].

While manual testing methods can be used to detect vulnerabilities in code, automatic testing is faster and can cover more code, detect more vulnerabilities than a human doing this manually. Although static analysis tends to be less accurate (but faster) than dynamic analysis due to the fact that static analysis relies on the source code and abstractions [40][41], while dynamic analysis uses test suites to find vulnerabilities in programs [41], we will be primarily interested in the static analysis part in the area of code verification. This is mainly because of the time constraints - static analysis works faster and produces sufficiently good results without the actual need of running the source code, while dynamic analysis introduces runtime overheads [42], which is not a very suitable outcome in this project. It must be noted, however, that for the best results the hybrid approach of static and dynamic analysis can be considered as an option, as in this way it is tried to combine the advantages of both types of code analysis [43].

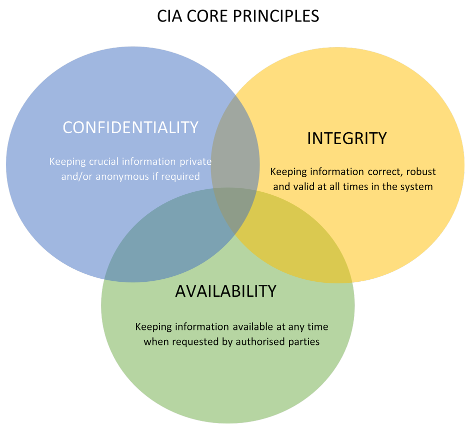
2.4 Cybersecurity

Cybersecurity is a broad term, which is quite abstract and encompasses plenty of areas not only related to computer science, but also engineering, politics, management and social sciences [44]. However, there is no exact established definition of "cybersecurity", as one can look from many different perspectives - for example, it can be looked as similar to information security [45], which is defined by the ISO/IEC 27000 standard, last updated in 2018 (as of 2021) [46] and the concept is defined as "preservation of confidentiality, integrity and availability of information" [47]. The International Telecommunication Union (ITU) defines cybersecurity as "the collection of tools, policies, security concepts, security safeguards, guidelines, risk management approaches, actions, training, best practices, assurance and technologies that can be used to protect the cyber environment and organization and user's assets." [48]. The United States Cybersecurity and Infrastructure Security Agency (CISA) is less abstract about cybersecurity - it says that "cybersecurity is the art of protecting networks, devices, and data from unauthorized access or criminal use and the practice of ensuring confidentiality, integrity, and availability of information" [49]. The National Institute of Standards and Technology Computer Security Resource Center (NIST CSRC) in the US gives a similar definition to cybersecurity: it is a "prevention of damage to, protection of, and restoration of computers, electronic communications systems, electronic communications services, wire communication, and electronic communication, including information contained therein, to ensure its availability, integrity, authentication, confidentiality, and nonrepudiation." [50]. As it can be seen from the citations, it can be derived - at least from the computer science point of view - that cybersecurity revolves mainly around the same idea: keeping electronic products secure by ensuring that they adhere to the CIA principle[2].

[2]See figures 2.2a and 2.2b for the visual representation of the CIA principle.


CONFIDENTIALITY
Keeping crucial information private and/or anonymous if required
INTEGRITY
Keeping information correct, robust and valid at all times in the system
CIA CORE PRINCIPLES
AVAILABILITY
Keeping information available at any time when requested by authorised parties

(a) The "CIA Triad" - Confidentiality, Integrity and Availability


CIA CORE PRINCIPLES
CONFIDENTIALITY
Keeping crucial information private and/or anonymous if required
INTEGRITY
Keeping information correct, robust and valid at all times in the system
AVAILABILITY
Keeping information available at any time when requested by authorised parties

(b) The "CIA Triad" as a Venn diagram

Figure 2.2: CIA Triad diagrams

Once we can establish the fact that one of the cores of cybersecurity is the CIA principle, it is possible to go further into analysing actual source code and determine where cybersecurity threats can occur if the adversaries would try to exploit the vulnerabilities existing in the code. Luckily, the most common vulnerabilities in any programming language or software, system, framework etc. are well-documented and can be easily accessed on the Internet. For example, the Common Weakness Enumeration (CWE) [51] list or the Common Vulnerability Enumeration (CVE) [52] list are good databases of documented cybersecurity threats of various levels, where programmers can find detailed descriptions about a specific vulnerability, such as type of threat, method(s) of reproducing the vulnerability, affected languages, systems, frameworks, programs etc. as well as CVSS score, which shows the level of severity of the threat [53]. In Solidity programming language, however, the documentation of vulnerabilities is scarce and not well standardised, leading to the problem where programmers may have heard about a specific vulnerability, but they tend to have a hard time finding the accurate description and possible ways of fixing the vulnerable code. Not everything is lost, though. The NCC Group published a Decentralized Application Security Project (DASP) Top 10 Solidity smart contract vulnerabilities in 2018 [54], where the most commonly occurring cybersecurity threats in smart contracts were documented and explained with plenty of examples. Also, the Smart Contract Weakness Classification Registry (SWC) was created as an attempt to standardise the documentation of vulnerabilities in Solidity. [55]. Currently, at the time of writing, there are thirty-six distinct vulnerabilities, each with their unique identifiers, description examples and relations to the CWE vulnerability list.

To improve the evaluation of vulnerabilities, some standards provide risk assessments, so that in a large software system or framework programmers could prioritise their workload, starting with the most dangerous cybersecurity threats and leaving the low priority threats for later. The Software Engineering Institute at Carnegie Mellon University in the US developed a risk assessment framework for C programming language SEI CERT C Coding Standard, which shows what the consequences can be if the rules of the standard would be ignored [56]. As the methodology for determining the priority levels is quite universal and does not depend on a specific programming language (although this particular risk assessment method is developed for C), in this project, in chapter 3, the tests with vulnerabilities have the priority levels assigned according to the method developed by SEI CERT C Coding Standard. The levels are from level 1 (L1 - the most dangerous) to level 3 (L3 - the least dangerous), with priorities ranging from 1 (lowest priority) to 27 (highest priority), as shown in figure 2.3. The priority of the vulnerability is calculated by multiplying the values from each rule - severity, likelihood and remediation cost [56].

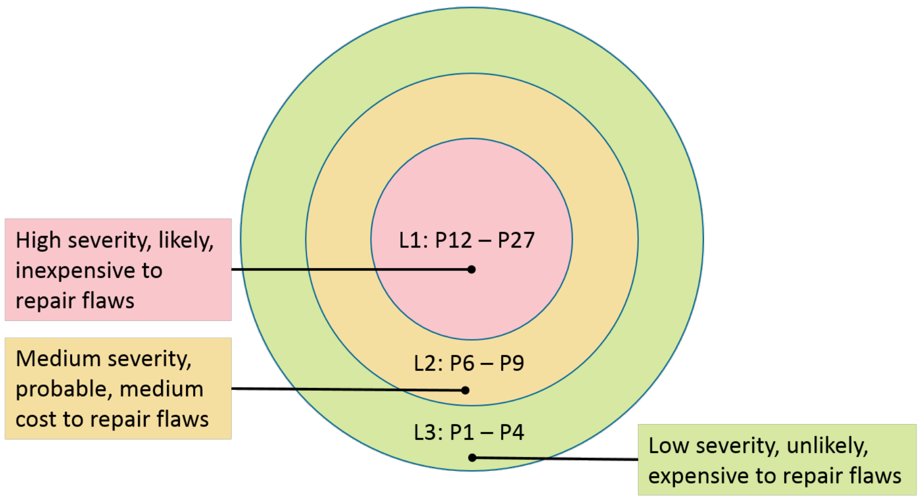

L1: P12 – P27
High severity, likely, inexpensive to repair flaws
L2: P6 – P9
Medium severity, probable, medium cost to repair flaws
L3: P1 – P4
Low severity, unlikely, expensive to repair flaws

Figure 2.3: Radial diagram of priorities and levels of the risk assessment developed for SEI CERT C Coding Standard [56]

2.5 Related work

The problem of vulnerabilities in smart contracts deployed on the blockchain have been observed and discussed by other members of the scientific community. *Atzei et al.* in 2017 performed a vulnerability survey on Ethereum smart contracts with indications of the most common types of vulnerabilities in the contracts [28]. *Chen et al.* in 2020 explored various types of vulnerabilities existing on the Ethereum blockchain, with some of the examples including re-entrancy, DoS and contract locking [27]. *Liu et al.* in the same year delved more into the specific types of vulnerabilities, in their case - out-of-gas vulnerabilities [35].

However, these articles mentioned tend to lack key cybersecurity properties evaluated in the surveys and they discuss more about the technicalities of vulnerabilities themselves. *Praitheeshan et al.* gives more information about not only the vulnerabilities themselves but also the security issues related to them [31]. For example, it is mentioned in the survey that unrestricted cryptocurrency transfers lead to failure to store and protect data [31] - in other words, this means that the data integrity cybersecurity property is violated from the

CIA triad. A more detailed example of this particular vulnerability is explained into detail in section 3.2.5.

Wang et al. in the article about Ethereum smart contracts mention a bit about the possible outcomes related to vulnerabilities, and because these attacks can be classified as cyberattacks, the paper gives some insight about the criminal intent in exploiting smart contract vulnerabilities [57]. While the article explains some methods adversaries can take in order to obtain an unfair advantage or even cause malicious acts such as digital currency theft, it does not go further into assigning cybersecurity-related concepts to vulnerable smart contracts in question. *Li et al.* in the survey published in 2020 perform a systematic analysis of vulnerable contracts with detailed explanations and possible security risks and ways of enhancing existing security methods [58]. The survey also includes attack vectors - an important factor in identifying possible adversary exploitation avenues and applicable remedies. The generalised versions, the Cyber Kill Chain by Lockheed Martin [59] and MITRE's ATT&CK knowledge base of tactics and techniques used by cyber attackers [60] can be possibly applied to the survey, thus expanding it into the direction of cybersecurity.

Dasgupta, Shrein and Gupta go one step further into standardising vulnerability evaluations from the cybersecurity perspective by providing references to CVE bug reports for Bitcoin but the survey lacks exact evaluations of individual vulnerabilities in the light of cybersecurity [61].

2.6 Summary

In this chapter, the theory required for this project has been explained in a level of detail needed for this project. The concepts of the blockchain, smart contracts, cryptocurrencies were briefly explained. The methods of verification of programs were also described briefly - the methods and applications. Cybersecurity concepts were also included in the chapter in order to give a better understanding of applications of cybersecurity to this project. Lastly, a short section on related work done by other people in the academic community showed us what has been done already in the area of smart contract verification and cybersecurity application.

Chapter 3

Research methodology and implementation

In this chapter, the practical part of the project is discussed, and the chapter is divided into several smaller parts. In the first part, the methods on how the tests were selected and written are explained, as well as the testing strategies and tools used to test against. Also, it is shown what parameters in testing are taken into account and their results are going to be discussed in the following chapter (chapter 4). The hypotheses are given as well in order to show whether the data gathered is going to prove those theories later or, otherwise, disprove them. Lastly, all smart contracts as tests are explained in detail in this chapter, so that it would be possible to gain an in-depth understanding on why these particular contracts were written in a way that most of them would contain some kind of a security vulnerability, while others do not have vulnerabilities at all or are irrelevant to the testing strategy altogether.

3.1 Testing approach

This section is divided into five smaller parts in order to explain the testing approach more into detail. In part 3.1.1 a hypothesis is presented, which is going to be used throughout the implementation process and checked against in the evaluation part in chapter 4. In part 3.1.2 various static analysis tools used for the project are discussed, their advertised capabilities, technical implementation and a short comparison between all tools. Part 3.1.3 gives a reader a bit more detail about the auxiliary tools and equipment used in the project, which helped to gather data and derives conclusions from the performance of static analysis tools. In part 3.1.4 it is explained what technical parameters were measured in order to gather information and perform analysis on them. Lastly, in part 3.1.5 the rules of eligibility for testing smart contracts are presented, so that it is easier to understand why some smart contract tests were considered as viable tests while others did not fit the required categories.

The overall testing strategy and research methodology follows a cyclic pattern, as it involves testing each of the tests, described more in detail in section 3.2. 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 out the example with various static analysis tools and write down the results under normal conditions. 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% of memory consumption, 90% of memory consumption, full test: 100% CPU consumption and 90% of 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 for each of the static analysis tools used. Figure 3.1 visually summarises the overall testing strategy and research methodology of this project.

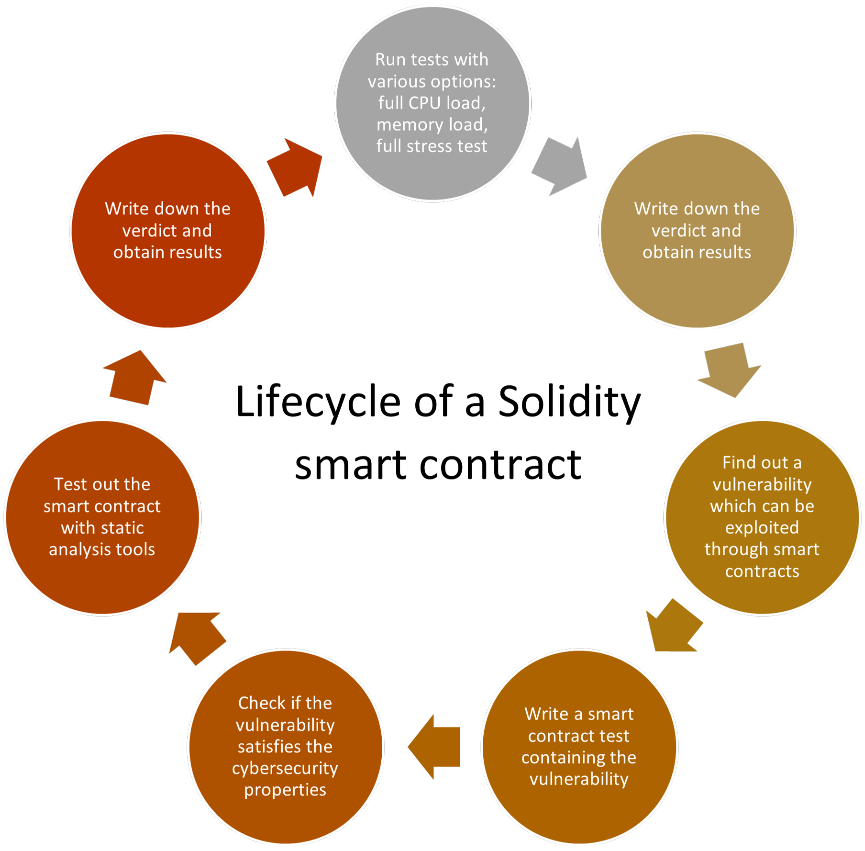

Lifecycle of a Solidity smart contract
Run tests with various options: full CPU load, memory load, full stress test
Write down the verdict and obtain results
Find out a vulnerability which can be exploited through smart contracts
Write a smart contract test containing the vulnerability
Check if the vulnerability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

Figure 3.1: The lifecycle of a Solidity smart contract vulnerability

3.1.1 Hypothesis

While the project did not have a required hypothesis per se, it was interesting to find out whether commonly held beliefs from the field of computer science hold in practice or not. As explained in section 1.1, one of the main aims of this project is to find out the best publicly available static analysis tool for verifying Solidity smart contract source code w.r.t. cybersecurity properties. However, this approach does not prove or disprove some of the generally convincing arguments derived from educated guesses, that is, guessing that one property or another might hold because of the previous expertise in similar areas of research. One of the arguments fitting the category is the following: *"The more complex and technically powerful system or tool or program tends to work slower, consumes more available resources at the expense of producing better results needed for its intended use"*. While it seems that this type of trade-off between accuracy and resource management or speed is common sense, and especially in computer hardware, where accuracy-efficiency trade-off is pronounced and researchers try to overcome this hindrance by creating new methods and/or hardware to improve both speed, energy consumption and accuracy of hardware devices [62] [63]. As in this project, both hardware and software devices will be used, and several of the performance metrics will include accuracy and resource consumption[1], it was thought that it would be a good idea to check if the trade-off holds in the area of smart contract verification. Therefore, the 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 the project, where it will be possible to derive if the hypothesis holds given the data gathered from the testing.

3.1.2 Static analysis tools

In this project, various static analysis tools were considered for testing, but there was a small problem - some of the tools were not maintained for a long time [64, 65], others did not work with newer Solidity compiler versions [66][2]. Some of the tools require a complex setup process, which may not be a viable option in commercial and industrial environments, therefore, they were discarded from the testing process as well. Another important factor was availability - some tools are not publicly available, and one has to buy them in order to be able to use the tool. Although this can be acceptable in the industry, for research purposes in this project these types of Solidity static analysis tools were not used. Therefore, we are left with five Solidity static analysis tools, which satisfy the requirements of availability, usability and preferably maintainability. The tools are as follows: Remix IDE static analysis plug-in, Slither, Oyente, Mythril and SmartCheck. These tools are written in different programming languages with different verification approaches - most of them employ already existing solvers,

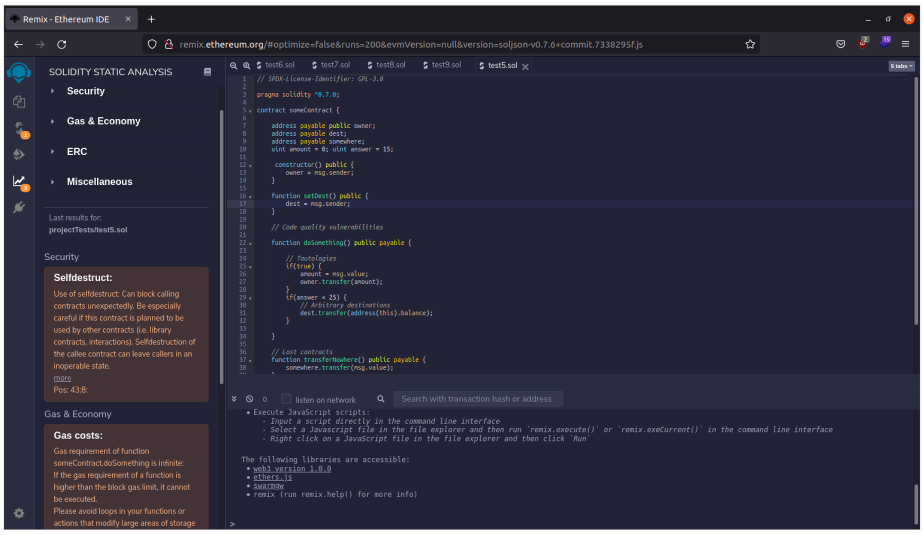
[1]More about the performance metrics is discussed in section 3.1.4.

[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.

such as Z3 SMT solver [67]. Each of the tools, their methods of approach and capabilities are explained in detail in the following paragraphs of this section.

Remix IDE plug-in

Remix IDE is an integrated development environment (IDE) developed by a team of seven people from various parts of the world [68]. As per their documentation website [69], the IDE is written in JavaScript, a popular Web-based programming language. The IDE contains plug-ins, additional functionalities which can be added manually by the programmer and used accordingly to his/her needs. One of those plug-ins is the Solidity static analysis plug-in, which checks for the most common security vulnerabilities in Solidity code. The static analysis is performed after the compilation process, and the vast majority of static analysis tools work in this way. It is worth noting that one does not need to run the actual source code so that the verifier can perform its analysis. This is crucial, especially in the blockchain, where a deployed smart contract cannot be revoked except for extraordinary circumstances (cf. section 2.2 for DAO attack).



Last results for:
projectTests/test5.sol

Security

Selfdestruct:
Use of selfdestruct: Can block calling contracts unexpectedly. Be especially careful if this contract is planned to be used by other contracts (i.e. library contracts, interactions). Selfdestruction of the callee contract can leave callers in an inoperable state.
more
Pos: 43:8:

Gas & Economy

Gas costs:
Gas requirement of function someContract.doSomething is infinite:
If the gas requirement of a function is higher than the block gas limit, it cannot be executed.
Please avoid loops in your functions or actions that modify large areas of storage

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.7.0;

contract someContract {

    address payable public owner;
    address payable dest;
    address payable somewhere;
    uint amount = 0; uint answer = 15;

     constructor() public {
        owner = msg.sender;
    }

    function setDest() public {
        dest = msg.sender;
    }

    // Code quality vulnerabilities

    function doSomething() public payable {

        // Tautologies
        if(true) {
            amount = msg.value;
            owner.transfer(amount);
        }
        if(answer < 25) {
            // Arbitrary destinations
            dest.transfer(address(this).balance);
        }

    }

    // Lost contracts
    function transferNowhere() public payable {
        somewhere.transfer(msg.value);
```

```
• Execute JavaScript scripts:
   - Input a script directly in the command line interface
   - Select a Javascript file in the file explorer and then run `remix.execute()` or `remix.exeCurrent()` in the command line interface
   - Right click on a JavaScript file in the file explorer and then click `Run`

 The following libraries are accessible:
 • web3 version 1.0.0
 • ethers.js
 • swarmgw
 • remix (run remix.help() for more info)

>
```

Figure 3.2: Remix IDE with some Solidity smart contract code

Underneath the plug-in there is the analyser, called Remix Analyzer [70], which is the static analysis tool performing the vulnerability checks. Although not publicised on the Github repository page, from the file types it can be derived that the Remix Analyzer is written in TypeScript, which is essentially JavaScript with enforced error checking. In the implementation source code, the vulnerability checking algorithms seem to be quite simple - a detect-and-report approach, which is not very complicated nor it requires background Maths knowledge (especially first-order logic) in order to write that particular type of algorithm. An example can be seen in listing 3.1. It can be seen from the code snippet that the function *_report* traverses all nodes of the Abstract Syntax Tree (AST) of the source code and searches for nodes containing self-destruct calls. If any of those nodes are found, the user is informed with the warning message.

```
export default class selfdestruct implements AnalyzerModule {
  name = 'Selfdestruct: '
  description = 'Contracts using destructed contract can be broken'
  category: ModuleCategory = category.SECURITY
  algorithm: ModuleAlgorithm = algorithm.HEURISTIC
  version: SupportedVersion = {
    start: '0.4.12'
  }

  abstractAst: AbstractAst = new AbstractAst()

  visit: VisitFunction = this.abstractAst.build_visit(
    (node: any) => isStatement(node) || (node.nodeType === 'FunctionCall' && isSelfdestructCall(node))
  )

  report: ReportFunction = this.abstractAst.build_report(this._report.bind(this))
  // eslint-disable-next-line @typescript-eslint/no-unused-vars
  private _report (contracts: ContractHLAst[], multipleContractsWithSameName: boolean, version: string): ReportObj[] {
    const warnings: ReportObj[] = []

    contracts.forEach((contract) => {
      contract.functions.forEach((func) => {
        let hasSelf = false
        func.relevantNodes.forEach((node) => {
          if (isSelfdestructCall(node)) {
            warnings.push({
              warning: 'Use of selfdestruct: Can block calling contracts unexpectedly. Be especially careful if this contract is planned to be used by other contracts (i.e. library contracts, interactions). Selfdestruction of the callee contract can leave callers in an inoperable state.',
              location: node.src,
              more: 'https://paritytech.io/blog/security-alert.html'
            })
            hasSelf = true
          }
          if (isStatement(node) && hasSelf) {
            warnings.push({
              warning: 'Use of selfdestruct: No code after selfdestruct is executed. Selfdestruct is a terminal.',
              location: node.src,
              more: `https://solidity.readthedocs.io/en/${version}/introduction-to-smart-contracts.html#deactivate-and-self-destruct`
            })
            hasSelf = false
          }
        })
      })
    })
    return warnings
  }
}
```

Listing 3.1: Example algorithm: detection of calling low-level function "selfdestruct" [1]

Another example is the vulnerability check for re-entrancy bugs - it follows a similar fashion to the example in listing 3.1, but in this case, it is not so straightforward to decide whether the code has a possible re-entrancy bug or not. Therefore, the checker tests if there are any variable or function state changes within the calculation process in the AST of the source code. The full code of the re-entrancy vulnerability checker can be found in listing 3.2.

```
export default class checksEffectsInteraction implements
    AnalyzerModule {
  name = 'Check-effects-interaction: '
  description = 'Potential re-entrancy bugs'
  category: ModuleCategory = category.SECURITY
  algorithm: ModuleAlgorithm = algorithm.HEURISTIC
  version: SupportedVersion = {
    start: '0.4.12'
  }

  abstractAst: AbstractAst = new AbstractAst()

  visit: VisitFunction = this.abstractAst.build_visit((node:
    FunctionCallAstNode | AssignmentAstNode | UnaryOperationAstNode
     | InlineAssemblyAstNode) => (
    node.nodeType === 'FunctionCall' && (isInteraction(node) ||
    isLocalCallGraphRelevantNode(node))) ||
          ((node.nodeType === 'Assignment' || node.nodeType === '
    UnaryOperation' || node.nodeType === 'InlineAssembly') &&
    isEffect(node)))

  report: ReportFunction = this.abstractAst.build_report(this.
    _report.bind(this))

  private _report (contracts: ContractHLAst[],
    multipleContractsWithSameName: boolean, version: string):
    ReportObj[] {
    const warnings: ReportObj[] = []
    const hasModifiers: boolean = contracts.some((item) => item.
    modifiers.length > 0)
    const callGraph: Record<string, ContractCallGraph> =
    buildGlobalFuncCallGraph(contracts)
    contracts.forEach((contract) => {
      contract.functions.forEach((func) => {
        func['changesState'] = this.checkIfChangesState(
          getFullQuallyfiedFuncDefinitionIdent(
            contract.node,
            func.node,
            func.parameters
          ),
          this.getContext(
            callGraph,
            contract,
            func)
        )
      })
      contract.functions.forEach((func: FunctionHLAst) => {
        if (this.isPotentialVulnerableFunction(func, this.
    getContext(callGraph, contract, func))) {
          const funcName: string =
    getFullQuallyfiedFuncDefinitionIdent(contract.node, func.node,
    func.parameters)
          let comments: string = (hasModifiers) ? 'Note: Modifiers
    are currently not considered by this static analysis.' : ''
          comments += (multipleContractsWithSameName) ? 'Note:
    Import aliases are currently not supported by this static
    analysis.' : ''
          warnings.push({
            warning: `Potential violation of Checks-Effects-Interaction pattern in ${funcName}: Could potentially lead to re-entrancy vulnerability. ${comments}`,
            location: func.node.src,
            more: `https://solidity.readthedocs.io/en/${version}/security-considerations.html#re-entrancy`
          })
        }
      })
    })
    return warnings
  }

  private getContext (callGraph: Record<string, ContractCallGraph>, currentContract: ContractHLAst, func: FunctionHLAst): Context {
    return { callGraph: callGraph, currentContract: currentContract, stateVariables: this.getStateVariables(currentContract, func) }
  }

  private getStateVariables (contract: ContractHLAst, func: FunctionHLAst): VariableDeclarationAstNode[] {
    return contract.stateVariables.concat(func.localVariables.filter(isStorageVariableDeclaration))
  }

  private isPotentialVulnerableFunction (func: FunctionHLAst, context: Context): boolean {
    let isPotentialVulnerable = false
    let interaction = false
    func.relevantNodes.forEach((node) => {
      if (isInteraction(node)) {
        interaction = true
      } else if (interaction && (isWriteOnStateVariable(node, context.stateVariables) || this.isLocalCallWithStateChange(node, context))) {
        isPotentialVulnerable = true
      }
    })
    return isPotentialVulnerable
  }

  private isLocalCallWithStateChange (node: FunctionCallAstNode, context: Context): boolean {
    if (isLocalCallGraphRelevantNode(node)) {
      const func: FunctionCallGraph | undefined = resolveCallGraphSymbol(context.callGraph, getFullQualifiedFunctionCallIdent(context.currentContract.node, node))
      return !func || (func && func.node['changesState'])
    }
    return false
  }

  private checkIfChangesState (startFuncName: string, context: Context): boolean {
    return analyseCallGraph(context.callGraph, startFuncName, context, (node: any, context: Context) => isWriteOnStateVariable(node, context.stateVariables))
  }
}
```

Listing 3.2: Example algorithm: detection of re-entrancy vulnerabilities [2]

The Remix IDE static analysis documentation states that the analyser can detect various security vulnerabilities such as usage of tx.origin, re-entrancy, inline assembly code, block timestamp usage, value truncation and possible out-of-gas vulnerabilities [71]. All of these vulnerabilities are tested out in the smart contract tests, which are discussed in section 3.2.

The example of Remix IDE and its static analysis plug-in can be seen in figure 3.2. It has a good User Interface (UI), while the results from the analyser are clearly seen on the left side of the IDE. Each vulnerability is classified into four main categories, and vulnerabilities have well-documented descriptions as well as suggestions on how to fix the vulnerabilities in question.

Slither

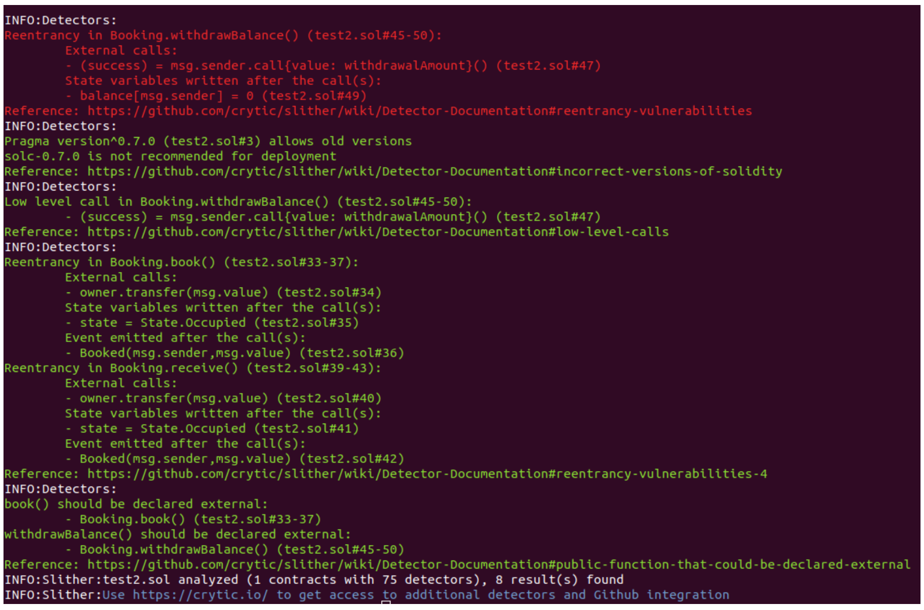
Slither is a static analysis tool for Solidity programming language, developed by Josselin Feist, Gustavo Grieco and Alex Groce [33] and written in Python version 3. Slither has a similar high-level architecture to other SAT or SMT-based bounded model checking (BMC) tools, such as ESBMC, where the compiled source code is converted to an AST, from that a control-flow graph (CFG) is produced, which basically shows all possible execution paths of a given program in the graph notation. After the symbolic execution stage in ESBMC case, it is possible to get the single static assignment (SSA) form of the program, which is then fed into a SMT solver [29]. Unlike in ESBMC, where the user can choose already existing SMT solvers such as Z3 [67], Boolector [72], MathSAT [73], CVC4 [74] and others [29], Slither employs its own vulnerability detection system, where three main code analysis parts are performed - read and write of variables, controlled access (in the original paper it is named "controlled functions") and data dependency of variables [33]. Then, bug detectors are attached to the whole system in order to detect required vulnerabilities.

In the Slither bug detector documentation, there are 75 different vulnerability detectors for Solidity smart contracts, ranging from the classical re-entrancy or tx.origin for validation vulnerabilities to dead code, double constructor usage or existence of tautologies types of vulnerabilities [75]. Figure 3.3 shows an example output trace of Slither static analysis tool. It is well-documented and colour-coded to identify which vulnerabilities are critical and which ones are only advisory.

For comparison, there are two figures - figure 3.4 depicting the internal structure of Slither static analysis tool and figure 3.5 depicting the structure of ESBMC. It can be clearly seen that the overall structure of both tools are similar, although they are used for different programming languages. As the authors of Slither claim that their tool was the most robust, the most accurate and the fastest out of four tools tested [33], it will be interesting to see if the results in chapter 4 correspond to the ones in the original Slither paper.

Oyente

Oyente is yet another static analysis tool developed in 2016 written in Python version 2 [76]. Oyente originally (and currently) uses Z3 as their SMT solver,



```
INFO:Detectors:
Reentrancy in Booking.withdrawBalance() (test2.sol#45-50):
        External calls:
        - (success) = msg.sender.call{value: withdrawalAmount}() (test2.sol#47)
        State variables written after the call(s):
        - balance[msg.sender] = 0 (test2.sol#49)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities
INFO:Detectors:
Pragma version^0.7.0 (test2.sol#3) allows old versions
solc-0.7.0 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in Booking.withdrawBalance() (test2.sol#45-50):
        - (success) = msg.sender.call{value: withdrawalAmount}() (test2.sol#47)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Reentrancy in Booking.book() (test2.sol#33-37):
        External calls:
        - owner.transfer(msg.value) (test2.sol#34)
        State variables written after the call(s):
        - state = State.Occupied (test2.sol#35)
        Event emitted after the call(s):
        - Booked(msg.sender,msg.value) (test2.sol#36)
Reentrancy in Booking.receive() (test2.sol#39-43):
        External calls:
        - owner.transfer(msg.value) (test2.sol#40)
        State variables written after the call(s):
        - state = State.Occupied (test2.sol#41)
        Event emitted after the call(s):
        - Booked(msg.sender,msg.value) (test2.sol#42)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-4
INFO:Detectors:
book() should be declared external:
        - Booking.book() (test2.sol#33-37)
withdrawBalance() should be declared external:
        - Booking.withdrawBalance() (test2.sol#45-50)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:test2.sol analyzed (1 contracts with 75 detectors), 8 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Figure 3.3: Slither static analysis tool in action

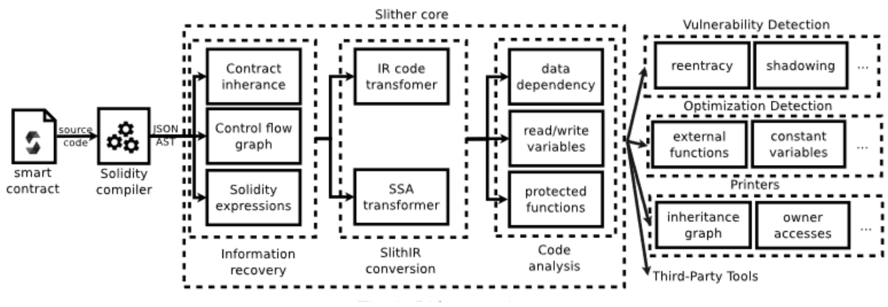


Figure 3.4: The operational schematic of Slither static analysis tool [33]

and the internal structure of this static analysis tool is very similar to Slither compared to ESMBC, discussed earlier in this subsection. Oyente tends to be more restrictive in terms of what kind of vulnerabilities it is able to catch given Solidity source code. In particular, Oyente detects possible re-entrancies, mishandled exceptions, timestamp dependencies and transaction-ordering dependencies (TOD). TOD contracts are dangerous in a similar fashion to programs using concurrency - concurrent programs might use common variables or constants, and if there are no critical sections enforced with mutual exclusion locks (mutexes), then we can have a race condition, where one may not be sure what the final result may be. Here, in TOD contracts, if different transactions invoke the same contract (an allusion to shared variables in concurrent programs) at the same time, the users will not know which transaction will be executed first and if order matters, then this state can lead to lost transactions, and consequently, materials used in the transaction, e.g. cryptocurrency. The example

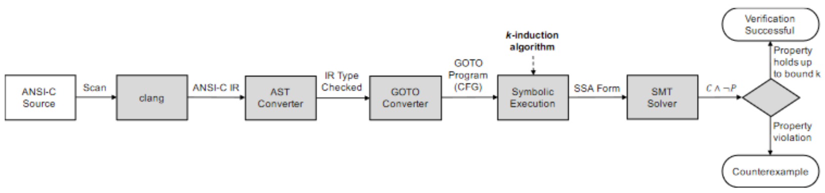
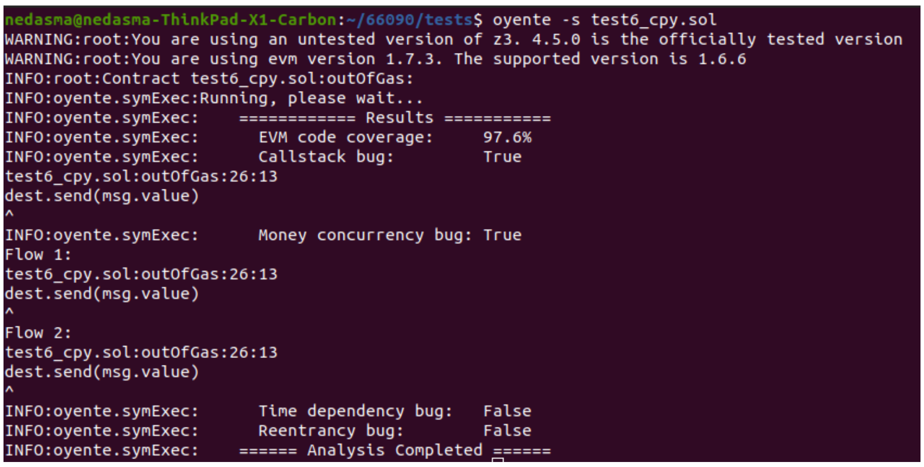


Figure 3.5: The operational schematic of ESBMC [29]

output trace of Oyente can be seen in figure 3.6. The output from Oyente is quite concise and clear, but as it can be seen from the figure, it is only capable of catching four types of vulnerabilities. A nice feature is the code coverage percentage - an important metric in testing to check how much code is covered by the testing tool, in this case, the analyser.



```
nedasma@nedasma-ThinkPad-X1-Carbon:~/66090/tests$ oyente -s test6_cpy.sol
WARNING:root:You are using an untested version of z3. 4.5.0 is the officially tested version
WARNING:root:You are using evm version 1.7.3. The supported version is 1.6.6
INFO:root:Contract test6_cpy.sol:outOfGas:
INFO:oyente.symExec:Running, please wait...
INFO:oyente.symExec:    ============ Results ===========
INFO:oyente.symExec:      EVM code coverage:     97.6%
INFO:oyente.symExec:      Callstack bug:         True
test6_cpy.sol:outOfGas:26:13
dest.send(msg.value)
^
INFO:oyente.symExec:      Money concurrency bug: True
Flow 1:
test6_cpy.sol:outOfGas:26:13
dest.send(msg.value)
^
Flow 2:
test6_cpy.sol:outOfGas:26:13
dest.send(msg.value)
^
INFO:oyente.symExec:      Time dependency bug:   False
INFO:oyente.symExec:      Reentrancy bug:        False
INFO:oyente.symExec:    ====== Analysis Completed ======
```

Figure 3.6: Example output trace of Oyente static analysis tool

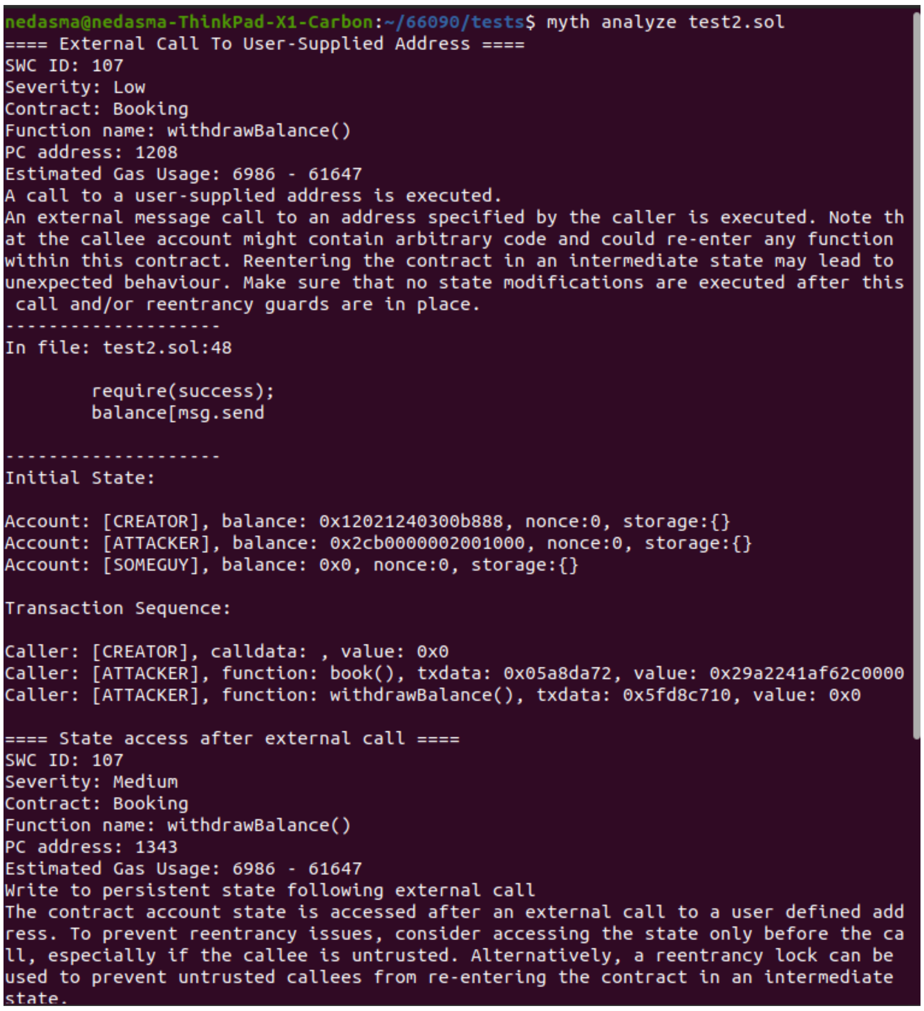
Although Oyente is an outdated tool at the time of writing, as its majority of source code was not updated for 4-5 years, and the last update was in November 2020[77], it was generally thought that it would be nice to include this tool in the project, as it seems that Oyente is a popular Solidity static analysis tool in the related areas of research[3].

Mythril

Mythril is a static analysis tool developed by the company named ConsenSys in 2018 and presented at the 9th Annual HITB Security Conference (HITB-SecConf) in Amsterdam, Netherlands [78]. Mythril uses LASER-ethereum as its symbolic execution backend [79], and that software uses the same concepts mentioned with Oyente and Slither - taking program states and putting them into CFGs, which can act as visual helpers in backtracking the vulnerability,

[3]In Google Scholar, searching for "oyente solidity static analysis tool" in August 2021 will give about 380 results.

and using some kind of intermediate representation (IR) of the original source code or converting to plain assembly code, which is then used with existing SAT/SMT solvers. LASER-ethereum uses Z3 SMT solver [78], and by the fact that Mythril uses LASER-ethereum, it is possible to deduct that the internal structure of the whole analysis tool is similar to Oyente. However, Mythril is capable of finding way more vulnerabilities than Oyente is, including but not limited to suicidal contracts, unchecked return values and DoS through external calls [80]. Also, it is worth noting that Mythril's vulnerability detection list is closely related to the SWC registry, which, as mentioned in section 2.4, is a good attempt at standardising and documenting existing vulnerabilities in Solidity smart contracts. The example output trace can be seen in figure 3.7. Mythril produces a clear and well-documented output trace of a vulnerability caught with counterexamples.



```
nedasma@nedasma-ThinkPad-X1-Carbon:~/66090/tests$ myth analyze test2.sol
==== External Call To User-Supplied Address ====
SWC ID: 107
Severity: Low
Contract: Booking
Function name: withdrawBalance()
PC address: 1208
Estimated Gas Usage: 6986 - 61647
A call to a user-supplied address is executed.
An external message call to an address specified by the caller is executed. Note th
at the callee account might contain arbitrary code and could re-enter any function
within this contract. Reentering the contract in an intermediate state may lead to
unexpected behaviour. Make sure that no state modifications are executed after this
 call and/or reentrancy guards are in place.
--------------------
In file: test2.sol:48

        require(success);
        balance[msg.send

--------------------
Initial State:

Account: [CREATOR], balance: 0x12021240300b888, nonce:0, storage:{}
Account: [ATTACKER], balance: 0x2cb0000002001000, nonce:0, storage:{}
Account: [SOMEGUY], balance: 0x0, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , value: 0x0
Caller: [ATTACKER], function: book(), txdata: 0x05a8da72, value: 0x29a2241af62c0000
Caller: [ATTACKER], function: withdrawBalance(), txdata: 0x5fd8c710, value: 0x0

==== State access after external call ====
SWC ID: 107
Severity: Medium
Contract: Booking
Function name: withdrawBalance()
PC address: 1343
Estimated Gas Usage: 6986 - 61647
Write to persistent state following external call
The contract account state is accessed after an external call to a user defined add
ress. To prevent reentrancy issues, consider accessing the state only before the ca
ll, especially if the callee is untrusted. Alternatively, a reentrancy lock can be
used to prevent untrusted callees from re-entering the contract in an intermediate
state.
```

Figure 3.7: Example output trace of Mythril static analysis tool

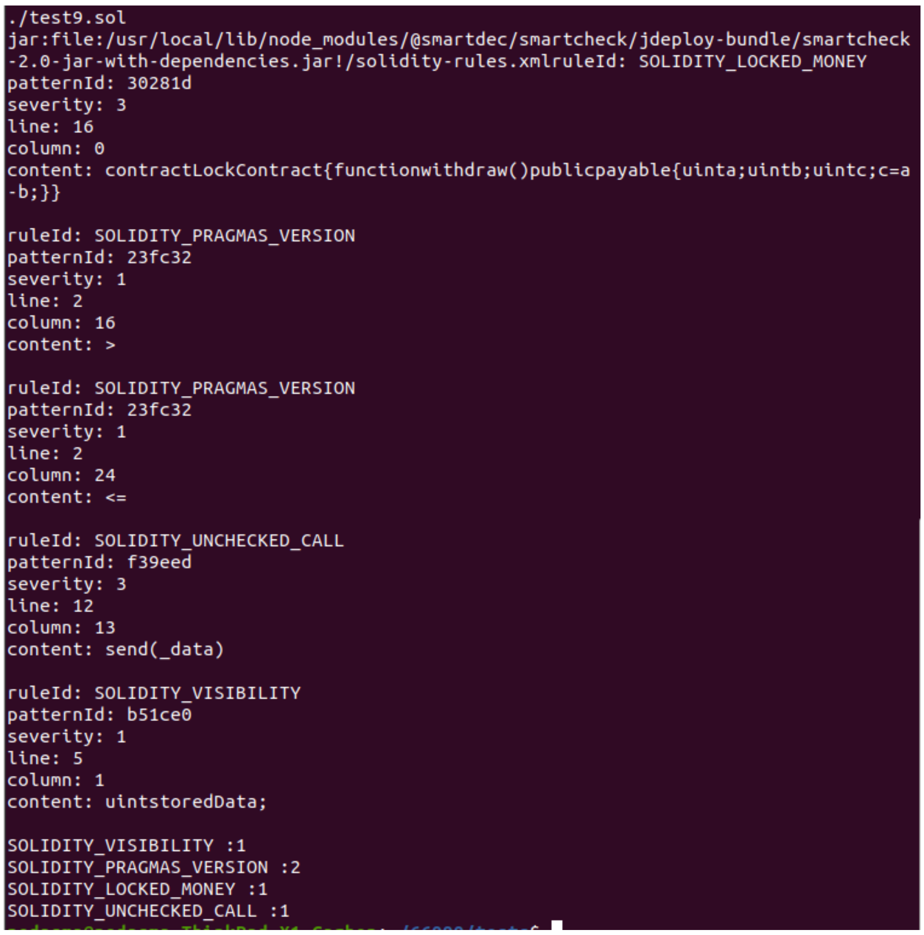
SmartCheck

SmartCheck is the last of the five static analysis tools discussed, introduced in 2018 and written in Java programming language. [81] This fact stands out from other tools, as they are written in languages that generally require fewer resources to be consumed - e.g. Python is a scripting language, so it does not need a compiler in order to run its code, and therefore it saves the number of files needed to run the program. The same goes with Web-based programming languages. Furthermore, SmartCheck uses ANTLR parser generator [82] and a custom Solidity grammar to build an AST, which, combined together, generate an XML parse tree acting as an IR [81]. This is an interesting approach not seen in other static analysis tools, but XML tends to be convenient when it comes to producing various tree-type structures. The original paper also mentions that vulnerabilities are detected through XPath queries, which effectively means that there is a list of vulnerabilities hard-coded somewhere in the analyser and it tries to compare the list with the given XPath pattern. This is a similar, find-and-report approach used by Remix IDE static analysis plug-in. However, this approach is simple, although it can be ineffective against snippets of code that have the same type of vulnerability the tool should find, but the source code is not an exact match with the ones the analysers have inside them. The authors of SmartCheck admit that more sophisticated patterns may produce false positives [81]. An example output of SmartCheck can be seen in figure 3.8. One of the drawbacks of SmartCheck's output trace is that the results are a little bit cryptic, and sometimes it can be difficult to understand what kind of vulnerability is actually caught by the analyser.

3.1.3 Tools and equipment

For this project, one laptop was used with mediocre technical parameters to simulate average working conditions and tools available in a commercial/industrial setting. The laptop has Intel i7-3667U type CPU with four cores running at 2 GHz. The laptop has 8 GB of total available memory, however, 7.32 GB of RAM can be used for any purpose - the remaining part is consumed by the system. 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 project, 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, several publicly available tools were used. In order to track resource consumption rates better, *htop* was used [83]. It is similar to 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. The example image of *htop* can be seen in figure 3.9. For benchmarking and average script running time tracking, *hyperfine* command-line benchmarking tool was installed in the laptop and used throughout the project [84]. One of the main advantages of *hyperfine* is that it allows benchmarking commands as well as running several benchmarks at the same time, thus saving time running a static analysis tool on each of the tests manually. An example running several Solidity smart contracts with the



```
./test9.sol
jar:file:/usr/local/lib/node_modules/@smartdec/smartcheck/jdeploy-bundle/smartcheck
-2.0-jar-with-dependencies.jar!/solidity-rules.xmlruleId: SOLIDITY_LOCKED_MONEY
patternId: 30281d
severity: 3
line: 16
column: 0
content: contractLockContract{functionwithdraw()publicpayable{uinta;uintb;uintc;c=a
-b;}}

ruleId: SOLIDITY_PRAGMAS_VERSION
patternId: 23fc32
severity: 1
line: 2
column: 16
content: >

ruleId: SOLIDITY_PRAGMAS_VERSION
patternId: 23fc32
severity: 1
line: 2
column: 24
content: <=

ruleId: SOLIDITY_UNCHECKED_CALL
patternId: f39eed
severity: 3
line: 12
column: 13
content: send(_data)

ruleId: SOLIDITY_VISIBILITY
patternId: b51ce0
severity: 1
line: 5
column: 1
content: uintstoredData;

SOLIDITY_VISIBILITY :1
SOLIDITY_PRAGMAS_VERSION :2
SOLIDITY_LOCKED_MONEY :1
SOLIDITY_UNCHECKED_CALL :1
```

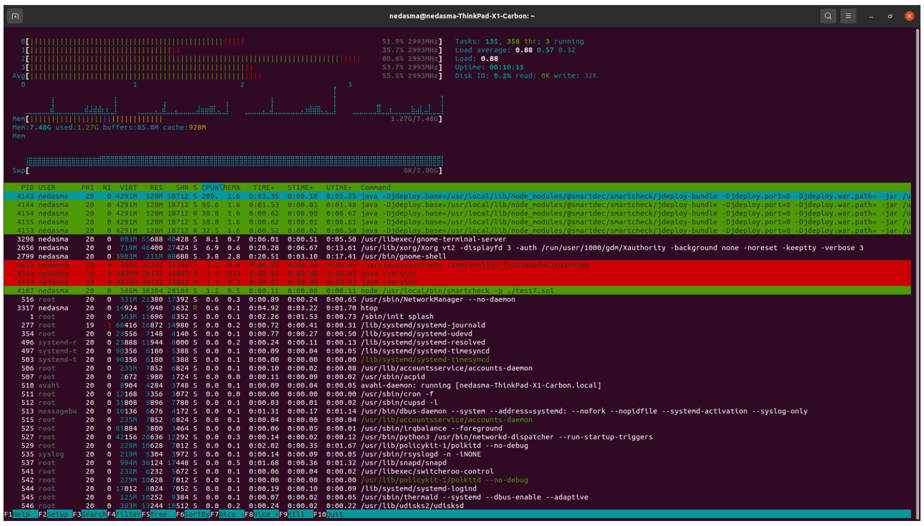
Figure 3.8: SmartCheck producing output trace for a given Solidity smart contract

static analysis tool can be seen in figure 3.10.

3.1.4 Performance metrics

In order to gather data for this project and derive results, it was necessary to establish what qualities of the verification process will be measured. As mentioned in section 1.1, it is important to find the most efficient as well as accurate static analysis tool, therefore, the following performance metrics were chosen for this project:

- 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 specialise in some types of vulnerabilities, for example, out-of-gas-vulnerabilities [22][23]. A good accuracy score shows that the

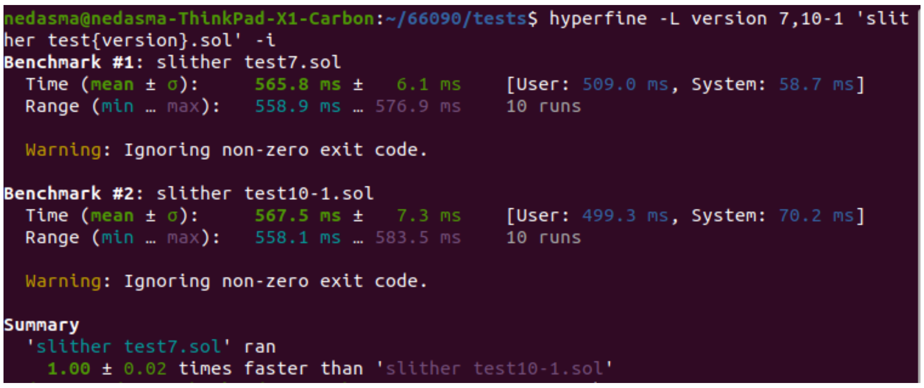


```
nedasma@nedasma-ThinkPad-X1-Carbon: ~

  0[                                          51.9% 2993MHz]   Tasks: 131, 358 thr; 3 running
  1[                                          35.7% 2993MHz]   Load average: 0.88 0.57 0.32
  2[                                          80.6% 2993MHz]   Load: 0.88
  3[                                          53.7% 2993MHz]   Uptime: 00:10:11
Avg[                                          55.5% 2993MHz]   Disk IO: 0.2% read: 0K write: 32K
Mem[                                          1.27G/7.48G]
Mem:7.48G used:1.27G buffers:85.0M cache:928M
Swp[                                          0K/2.00G]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU%▽MEM%   TIME+   STIME+   UTIME+  Command
 4143 nedasma    20   0 4291M  120M 18712 S 209.  1.6  0:03.35  0:00.10  0:03.25 java -Djdeploy.base=/usr/local/lib/node_modules/@smartdec/smartcheck/jdeploy-bundle -Djdeploy.port=0 -Djdeploy.war.path= -jar /u
 4144 nedasma    20   0 4291M  120M 18712 S 95.6  1.6  0:01.53  0:00.05  0:01.48 java -Djdeploy.base=/usr/local/lib/node_modules/@smartdec/smartcheck/jdeploy-bundle -Djdeploy.port=0 -Djdeploy.war.path= -jar /u
 4154 nedasma    20   0 4291M  120M 18712 R 38.8  1.6  0:00.62  0:00.00  0:00.62 java -Djdeploy.base=/usr/local/lib/node_modules/@smartdec/smartcheck/jdeploy-bundle -Djdeploy.port=0 -Djdeploy.war.path= -jar /u
 4155 nedasma    20   0 4291M  120M 18712 S 38.8  1.6  0:00.62  0:00.01  0:00.61 java -Djdeploy.base=/usr/local/lib/node_modules/@smartdec/smartcheck/jdeploy-bundle -Djdeploy.port=0 -Djdeploy.war.path= -jar /u
 4153 nedasma    20   0 4291M  120M 18712 R 32.5  1.6  0:00.52  0:00.02  0:00.50 java -Djdeploy.base=/usr/local/lib/node_modules/@smartdec/smartcheck/jdeploy-bundle -Djdeploy.port=0 -Djdeploy.war.path= -jar /u
 3298 nedasma    20   0  803M 55688 40428 S  8.1  0.7  0:06.01  0:00.51  0:05.50 /usr/libexec/gnome-terminal-server
 2656 nedasma    20   0  719M 46400 27424 S  6.9  0.6  0:20.28  0:06.67  0:13.61 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/1000/gdm/Xauthority -background none -noreset -keeptty -verbose 3
 2799 nedasma    20   0 3903M  211M 88688 S  3.8  2.8  0:20.51  0:03.10  0:17.41 /usr/bin/gnome-shell
 4115 nedasma    20   0  595M 31732 25248 S  3.1  0.4  0:00.05  0:00.00  0:00.05 /usr/local/bin/node /tmp/shelljs_f553582e0dc2cd34a3e8
 4123 nedasma    20   0 3829M 20132 15052 S  1.3  0.3  0:00.02  0:00.00  0:00.02 java -version
 4124 nedasma    20   0 3829M 20132 15052 R  1.3  0.3  0:00.02  0:00.00  0:00.02 java -version
 4107 nedasma    20   0  566M 36304 28164 S  1.2  0.5  0:00.11  0:00.00  0:00.11 node /usr/local/bin/smartcheck -p ./test7.sol
  516 root       20   0  331M 21380 17392 S  0.6  0.3  0:00.89  0:00.24  0:00.65 /usr/sbin/NetworkManager --no-daemon
 3317 nedasma    20   0 14924  5940  3632 R  0.6  0.1  0:04.92  0:03.22  0:01.70 htop
    1 root       20   0  163M 11696  8352 S  0.0  0.1  0:02.26  0:01.53  0:00.73 /sbin/init splash
  277 root       19  -1 68416 16872 14980 S  0.0  0.2  0:00.72  0:00.41  0:00.31 /lib/systemd/systemd-journald
  354 root       20   0 23556  7148  4140 S  0.0  0.1  0:00.77  0:00.27  0:00.50 /lib/systemd/systemd-udevd
  496 systemd-r  20   0 23888 11944  8000 S  0.0  0.2  0:00.24  0:00.11  0:00.13 /lib/systemd/systemd-resolved
  497 systemd-t  20   0 90356  6180  5388 S  0.0  0.1  0:00.09  0:00.04  0:00.05 /lib/systemd/systemd-timesyncd
  503 systemd-t  20   0 90356  6180  5388 S  0.0  0.1  0:00.00  0:00.00  0:00.00 /lib/systemd/systemd-timesyncd
  506 root       20   0  235M  7852  6824 S  0.0  0.1  0:00.10  0:00.02  0:00.08 /usr/lib/accountsservice/accounts-daemon
  507 root       20   0  2672  1980  1724 S  0.0  0.0  0:00.11  0:00.09  0:00.02 /usr/sbin/acpid
  510 avahi      20   0  8904  4284  3748 S  0.0  0.1  0:00.09  0:00.04  0:00.05 avahi-daemon: running [nedasma-ThinkPad-X1-Carbon.local]
  511 root       20   0 12168  3356  3072 S  0.0  0.0  0:00.00  0:00.00  0:00.00 /usr/sbin/cron -f
  512 root       20   0 31808  9896  7780 S  0.0  0.1  0:00.03  0:00.01  0:00.02 /usr/sbin/cupsd -l
  513 messagebu  20   0 10136  6676  4172 S  0.0  0.1  0:01.31  0:00.17  0:01.14 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
  515 root       20   0  235M  7852  6824 S  0.0  0.1  0:00.04  0:00.00  0:00.04 /usr/lib/accountsservice/accounts-daemon
  525 root       20   0 81884  3800  3464 S  0.0  0.0  0:00.06  0:00.05  0:00.01 /usr/sbin/irqbalance --foreground
  527 root       20   0 42156 20636 12292 S  0.0  0.3  0:00.14  0:00.02  0:00.12 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
  529 root       20   0  229M 10628  7012 S  0.0  0.1  0:02.02  0:00.35  0:01.67 /usr/lib/policykit-1/polkitd --no-debug
  535 syslog     20   0  219M  5304  3972 S  0.0  0.1  0:00.14  0:00.09  0:00.05 /usr/sbin/rsyslogd -n -iNONE
  537 root       20   0  994M 36124 17448 S  0.0  0.5  0:01.68  0:00.36  0:01.32 /usr/lib/snapd/snapd
  541 root       20   0  232M  6232  5672 S  0.0  0.1  0:00.06  0:00.04  0:00.02 /usr/libexec/switcheroo-control
  542 root       20   0  229M 10628  7012 S  0.0  0.1  0:00.00  0:00.00  0:00.00 /usr/lib/policykit-1/polkitd --no-debug
  544 root       20   0 17012  8024  7052 S  0.0  0.1  0:00.19  0:00.10  0:00.09 /lib/systemd/systemd-logind
  545 root       20   0  125M 10252  9384 S  0.0  0.1  0:00.07  0:00.02  0:00.05 /usr/sbin/thermald --systemd --dbus-enable --adaptive
  546 root       20   0  383M 13244 10512 S  0.0  0.2  0:00.24  0:00.02  0:00.22 /usr/lib/udisks2/udisksd
F1Help  F2Setup  F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 3.9: htop interactive process viewer on Linux machine



```
nedasma@nedasma-ThinkPad-X1-Carbon:~/66090/tests$ hyperfine -L version 7,10-1 'slit
her test{version}.sol' -i
Benchmark #1: slither test7.sol
  Time (mean ± σ):     565.8 ms ±   6.1 ms    [User: 509.0 ms, System: 58.7 ms]
  Range (min … max):   558.9 ms … 576.9 ms    10 runs

  Warning: Ignoring non-zero exit code.

Benchmark #2: slither test10-1.sol
  Time (mean ± σ):     567.5 ms ±   7.3 ms    [User: 499.3 ms, System: 70.2 ms]
  Range (min … max):   558.1 ms … 583.5 ms    10 runs

  Warning: Ignoring non-zero exit code.

Summary
  'slither test7.sol' ran
    1.00 ± 0.02 times faster than 'slither test10-1.sol'
```

Figure 3.10: Hyperfine benchmarking tool performing tests on commands

static analysis tool is competent in reporting different types of bugs and can be used in business environments.

- Speed. Everyone hates slow systems, and very often, time is money, therefore, if verification of a single smart contract takes ages, then it might not be usable under stress, where hundreds of smart contracts could be deployed every day and all of these contracts have to be checked for possible security flaws. On the other hand, 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.

- CPU consumption. Efficient tools are important so that their operations would not block other processes happening at the same time. It is very likely that a user would run a static analysis tool, while at the same time browse the Internet, have several programs and/or applications open or running in the background.

- Memory consumption. The same argument mentioned about the CPU consumption applies here, although a high memory consumption rate for a prolonged period of time can have unwanted consequences, e.g. other programs or threads (or even the static analyser 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 in order to prevent errors and crashes from happening.

3.1.5 Smart contract eligibility for testing

Solidity smart contracts, its code, as in other programming languages, can have various security flaws in them, but some vulnerabilities can be minor and even if they can be exploited, the damage caused will be minuscule. This is why in this project the smart contracts are carefully written so that they would contain bugs that can cause severe damage and violate at least one of the CIA triad properties explained in section 2.4. Of course, some smart contracts might not contain any vulnerabilities at all, but these types of smart contracts are used to check for possible false positives.

3.2 Smart contract tests

In this section, twelve different Solidity smart contracts are going to be described in detail, with vulnerabilities (or the lack of them) in each of the smart contracts. Each test is evaluated in terms of several cybersecurity concepts, that is, whether the particular vulnerability violates one or more of CIA properties - confidentiality, integrity and availability. Also, each smart contract is given a priority level discussed in section 2.4 about the SEI CERT C Coding Standard risk assessment. The tests are prioritised 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.

3.2.1 Test 1 - no vulnerabilities (canary test)

The first test in the series of Solidity smart contract vulnerability tests, it is mainly used for plainly testing out if the static analysis tool in question is working properly and does not produce false positives. Trivially, the source code of this test is too small in order to produce meaningful false positives but sometimes simplicity is key in finding possible errors produced by the program. The source code of this smart contract test can be seen in listing 3.2.1. It contains a single contract called *SimpleStorage*, which in turn contains two functions - *set()* and *get()*. The *set()* function allows setting any given unsigned integer value to variable *storedData*, whose type is an unsigned integer as well. The *get()* function, on the other hand, allows getting the set value of *storedData*, and if this variable was not set previously, then the function would return zero. It is important to note that in Solidity there are undefined values or variables, therefore, it is not required to give a starting value to variables, although it is recommended.

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Listing 3.2.1: Test No. 1 - a “canary” test for finding possible false positives and checking if the tools work as intended

Because this contract is very simple in its nature, it would be hard to obfuscate this contract to the static analysis tool so that it would report as the contract having a vulnerability - a false positive - and indeed, there are no security bugs in this code.

As for cybersecurity properties, none of the following was violated: confidentiality, integrity or availability. The risk assessment does not apply to this test as there are no security vulnerabilities in this smart contract.

3.2.2 Test 2 - re-entrancy vulnerability

This is the second smart contract to be used as a test in the project. It is a bit more expansive than the first test described in section 3.2.1, contains more functions and other concepts in Solidity programming language, such as constructors, modifiers, events and the like. The source code can be seen in listing 3.2.2. It is important to note that this contract has a more restrictive compiler version, written in line 3 - the “pragma” argument in Solidity denotes which compiler versions are compatible with the smart contract code. In this case, only Solidity compiler versions from 0.7.0 to 0.7.6 can be used to compile this contract. In the first test, any compiler version starting with 0.4.16 and ending with 0.9.0 can be used. It is worth noting that new compiler versions include bugfixes, security patches, deprecations of old functions and introducing new restrictions or relaxations on how programmers should work with Solidity. Therefore, there are many tests, including test 10, which are entirely dependent on Solidity compiler versions. This shows that it is crucial to have up-to-date code which can be compiled with the newest Solidity compiler version.

This smart contract has a constructor, which sets the contract owner of type “address payable”. Technically speaking, all identifiers used in Solidity, for example contract ownership, setup of payers and payees in the transaction and identification of users are through addresses, which are 20-byte Ethereum address values. The “payable” option allows the particular variable to send and receive Ether - a monetary-type resource used to perform transactions between parties. Also, this contract has two modifiers - these are similar to guard statements in other programming languages. Modifiers have one or more *require* statements, which contain a logical statement and a message field if that statement cannot be satisfied. One may not include the message, but then the error message would be a plain transaction error from the blockchain instead of a user-friendly message.

The real vulnerability of this smart contract hides in lines 45 to 52. However, the first problem is the double-spend security bug, which can be tried and simulated in this contract, although the end result would not be quite the correct one. In the double-spend attack, an adversary can use the same digital currency (cryptocurrency), such as Ether, to make more than one payment to different transactions at the same time with the same funds in the wallet. A real-life analogy would be paying the same £5 note in two different shops at the same time. While this is not possible physically in reality, in blockchain technology, if this is done, then several blocks are put onto the blockchain and they have to be validated, but due to the race condition happening in this case in which it is not known whether Transaction One or Transaction Two would succeed in being confirmed first, therefore, only one out of several recipients of the malicious transaction will be able to receive cryptocurrency the adversary paid from his/her digital wallet. This can be simulated in this contract by calling both functions *book()* and *receive()* at the same time, as they essentially do the same thing - transfer currency from the payer's wallet to the contract owner's wallet. As both functions masquerade themselves as able to book a, say, hotel room, the owner of this contract becomes the malicious user by tricking the payer to click both functions and instead of paying 3 Ether, the payer would pay 6 Ether - twice the price of the room. It is also worth noting that the function *receive()* does not have a guard modifier for checking if the room is vacant or occupied. Therefore, it is possible that the user can book the already booked room, which is essentially cheating from the owner's side.

The original security vulnerability in this test contract is the re-entrancy bug, where an attacker can abuse the *call.value* low-level Solidity call function by employing the fallback function on his/her side. Knowing that *call.value* changes the state of the variable, in this case, msg.sender (also a specific Solidity variable to denote payer's address), the adversary can repeatedly call the function *withdrawBalance()* and get the digital currency he/she does not have it stored in the balances. This is a dangerous bug because exploiting this is relatively easy, fast and can drain the money held by the contract very quickly. Therefore, in the risk assessment, this vulnerability would have the severity rating of "Medium", as the integrity of data is violated, but it is not severe enough to be able to inject arbitrary code, for example. The likelihood rating is "Likely", as these types of vulnerabilities are well-documented and happen quite often. The remediation cost level is "Medium", as the detection is automatic by static analysis tools, but repairs have to be done manually. By multiplying the priorities (Severity: 2, Likelihood: 3, Remediation: 2) we get the priority level of P12 and the overall risk level of L1. In terms of CIA properties, the integrity property is violated as constraints for checking re-entrancy are non-existent or are implemented incorrectly.

3.2.3 Test 3 - dead code vulnerability

This is the third test of the Solidity smart contract vulnerability series. This smart contract is tricky, as it can be seen from the first glance that it does not

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.7.0;

contract Booking {

    enum State {Vacant, Occupied}
    State public state;

    address payable public owner;
    mapping (address => uint) private balance;

    event Booked(address _occupant, uint _amount);

    constructor() public {
        owner = msg.sender;
        state = State.Vacant;
    }

    modifier onlyIfVacant {
        require(state == State.Vacant, "The room is occupied!");
        _;
    }

    modifier costs(uint _amount) {
        require(msg.value >= _amount, "Insufficient funds!");
        _;
    }

    // Simulated double-spend vulnerability due to two functions doing the same
↪   thing without protection
    // Also, the second function does not check if the room is already occupied

    function book() public payable onlyIfVacant costs(3 ether) {
        owner.transfer(msg.value);
        state = State.Occupied;
        emit Booked(msg.sender, msg.value);
    }

    receive() external payable costs(3 ether) {
        owner.transfer(msg.value);
        state = State.Occupied;
        emit Booked(msg.sender, msg.value);
    }

    // Re-entrancy: function can be called repeatedly until the balance is set
↪   to 0, the attacker can withdraw
    // money he/she does not have
   function withdrawBalance() public {
            uint withdrawalAmount = balance[msg.sender];
            (bool success, ) = msg.sender.call.value(withdrawalAmount)("");
            require(success);
            balance[msg.sender] = 0;
    }
}
```

Listing 3.2.2: Test No. 2 - a smart contract test containing a re-entrancy security bug

Table 3.1: Risk assessment calculation table for test 2

Risk assessment type	**Value**
Severity	2
Likelihood	3
Remediation cost	2
Priority level	$2x3x2 = 12$ (P12)
Overall risk level	L1

contain any security bugs in itself, but the adversary can exploit the fact that there are functions in the code which do not contribute to the final answer given on lines 26-27. The code snippet can be seen in listing 3.2.3. While function *doSomething()* does *something*, the following functions *doSomethingElse()* and *doUselessCalculations()* perform some operations, but their given results do not go anywhere. The variable *answer* stores the value returned by the two functions in question, but that value does not go to the money transfer function call or the event call. Therefore, the code from line 30 to line 63 can be considered as dead code. It is also true that unreachable statements in the code can be called dead, but that would not probably impact the overall running speed of the contract, let alone the static analysis tools. Functions without their meaningful applications, on the other hand, can cause a DoS type of cyberattack, as invoking the function *doSomething()* multiple times can exhaust the available system resources, thus crashing the Ethereum Virtual Machine (EVM) used on the node the contract is deployed. Thus, the contract might be unreachable for some time, violating one of the important CIA cybersecurity properties - availability.

As for the risk assessment of this test smart contract, the severity is low due to the fact that the adversary can initiate a DoS attack, but nothing more. The likelihood value would be 2 (probable) because while programmers tend to know what they write and which part of code is responsible for, it is not uncommon that some functionalities are updated, re-written while the old functions are kept in the source code, although they do not contribute to the final result. The remediation cost is "Low" - these types of vulnerabilities can be detected automatically and refactored automatically with the help of IDEs as well. By multiplying the priorities (Severity: 1, Likelihood: 2, Remediation: 1) we get the priority level of P2 and the overall risk level of L3.

Table 3.2: Risk assessment calculation table for test 3

Risk assessment type	**Value**
Severity	1
Likelihood	2
Remediation cost	1
Priority level	$1x2x1 = 2$ (P2)
Overall risk level	L3

```solidity
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.7.0;

contract Test {

    address payable public owner;
    uint amount = 0; uint answer = 0;


    event Success(address _from, uint _amount);

    constructor() public {
        owner = msg.sender;
    }

    // Dead code vulnerability as functions are not checked by the static
↪   analysis tool if they contribute to the final answer

    function doSomething() public payable {

        amount = msg.value;
        uint val = 2;

        answer = doSomethingElse(amount, val);

        owner.transfer(amount);
        emit Success(msg.sender, amount);
    }

    function doSomethingElse(uint a, uint b) private returns (uint value) {
        uint c = 0; uint res = 0;
        for(uint i = 0; i < b; i++) {
            if(i == 0) {
                c++;
                res += doUselessCalculations(a, b, c);
            }
            else if(a < b) {
                c += a;
                res += doUselessCalculations(b, a, c);
            }
            else {
               c++;
               res += doUselessCalculations(c, b, a);
            }
        }
        return res;
    }

    function doUselessCalculations(uint a, uint b, uint c) private returns
↪   (uint res) {
        uint someAnswer = 1;
        for(uint i = 1; i <= 1000; i++) {
            someAnswer += mulmod(a, b, addmod(c, c, a));
        }
        for(uint j = 1; j <= 1000; j++) {
            someAnswer += mulmod(a, b, addmod(c, 5, a));
        }
        for(uint k = 1; k <= 1000; k++) {
            someAnswer += mulmod(a, b, addmod(c, 10, a));
        }
        return someAnswer;
    }

}
```

Listing 3.2.3: Test No. 3 - dead code vulnerability

3.2.4 Test 4 - weak PRNG and timestamp dependency vulnerabilities

Test 4 is the fourth smart contract used in the project, as its numerical title suggests. This contract is mainly concerned with the weak pseudo-random number generation (PRNG) and timestamp dependencies, that is, calculations based on block timestamps, block hashes or even current server time. The problem lies in the technology of blockchain itself - it does not provide anything which can be both pseudorandom and cryptographically secure. The block hashing function retains the properties of producing pseudo-random numbers, but miners can tamper with the block properties and thus, influence the outcome of the hashing function. The same goes for timestamps, but, in this case, it is even more pronounced - if the adversary knows when the block with the contract was deployed on the blockchain (or the current time the block is deployed), he/she can make a calculated guess and correctly identify numbers which should be random in theory. Also, it should be stressed that everything in the contract, all source code is visible to everyone, therefore, if the number for guessing is generated from a small pool of numbers, as in this example case seen in listing 3.2.4 function *takeAGuess()*, malicious users can calculate all numbers beforehand and use a brute-force approach in order to guess the number correctly and profit from that.

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.6.0;

contract rngTest {

    uint public rng;

    // Weak RNG functions
    function takeAGuess() external{
      rng = uint256(blockhash(10000)) % 10;
    }
    function takeAnotherGuess() external{
      rng = uint256(block.timestamp) % 50;
    }
    function takeAGuessNow() external{
      rng = uint256(now) % 99;
    }
}
```

Listing 3.2.4: Test No. 4 - code example of timestamp and unsafe pseudo-randomness dependencies

Two out of three CIA triad's cybersecurity properties are violated in this smart contract - confidentiality and integrity. Confidentiality is violated because malicious miners can obtain information about the deployed block in the blockchain, e.g. the hash of the block, the timestamp, current Unix time etc. The integrity property is also violated. Although the functionality of the smart contract is not interrupted, however, due to its vulnerable properties the miners can exploit those properties and affect the outcome of the whole program. For example, if the smart contract deployed is a lottery, the adversary miner can influence the lottery results for his/her own advantage. The risk assessment of this smart contract is as follows:

- Severity - medium, data integrity can be violated by tampering with operations inside the block.
- Likelihood - unlikely, the vulnerability is documented, it has its own SWC registry entry [85] and the use of timestamps and block hashes are greatly discouraged.
- Remediation cost - medium, the detection is automatic through static analysis tools but fixing requires manual labour of programmers.

Table 3.3: Risk assessment calculation table for test 4

Risk assessment type	**Value**
Severity	2
Likelihood	1
Remediation cost	2
Priority level	$2x1x2 = 4$ (P4)
Overall risk level	L3

3.2.5 Test 5 - lost contracts and unprotected self-destruction vulnerabilities

This is the fifth Solidity smart contract test in the series. It contains several vulnerabilities, although one of them will not be included in the analysis as it does not satisfy the requirement of being a cybersecurity problem. The source code of the contract can be seen in listing 3.2.5. As it can be seen from the code, there are three main types of vulnerabilities: poor code quality, lost contracts (sending Ether to uninitialised addresses) and unreserved use of *selfdestruct()* function. The poor code quality, such as using tautologies or not giving initial values to variables that have address types is a bad coding practice but it cannot be considered as a security vulnerability in this project. However, some static analysis tools detect the places where the code quality could be improved, and this will be explained more in detail in chapter 4.

The arbitrary data send vulnerability can be seen in lines 31 and 38. The problem here is that both variables *dest* and *somewhere* do not have initial values, therefore, their addresses are 0x0, or NULL in C language terms. Solidity, unlike in C, does not have undefined behaviour, that is why the contract will not crash if these functions will be called. However, function calls like "transfer" or "send", capable of transferring Ether to other addresses, can remove the cryptocurrency from the victim's wallet without the ability to recover the losses. The more severe version of data structures being uninitialised is the "contract suicide", or calling low-level functions such as *suicide()* or *selfdestruct()* without proper authorisation. Calling these functions will terminate the contract and all transactions in action with the contract would be terminated as well. This can have severe consequences if someone called those functions when they should not have access to those functions in the first place. The Parity multi-sig wallet exploit is a prime example of an adversary obtaining the rights of an owner of the contract and then he/she killed the whole multi-sig wallet system [86].

```solidity
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.7.0;

contract someContract {

    address payable public owner;
    address payable dest;
    address payable somewhere;
    uint amount = 0; uint answer = 15;

     constructor() public {
        owner = msg.sender;
    }

    function setDest() public {
        dest = msg.sender;
    }

    // Code quality vulnerabilities

    function doSomething() public payable {

        // Tautologies
        if(true) {
            amount = msg.value;
            owner.transfer(amount);
        }
        if(answer < 25) {
            // Arbitrary destinations
            dest.transfer(address(this).balance);
        }

    }

    // Lost contracts
    function transferNowhere() public payable {
        somewhere.transfer(msg.value);
    }

    // Unprotected self-destruct
    function kill() public{
        selfdestruct(msg.sender);
    }
}
```

Listing 3.2.5: Test No. 5 - unprotected self-destruction and arbitrary data send vulnerabilities

Hypothetically speaking, the adversary could inject arbitrary code or commands into uninitialised memory addresses, therefore, all three CIA properties would be violated. The confidentiality property would be violated if the adversary can abuse the fact that the memory address is not given explicitly and the data can be transferred to the memory address the attacker can extract and use. Due to the same fact, the data integrity can be damaged and the availability of the contract can be affected as well. If the self-destruction functions are called, then the contract is rendered inaccessible.

The risk assessment for this smart contract is as follows:

- Severity - high, a possibility of running arbitrary code.
- Likelihood - probable, especially in large scale contracts, where one can get lost of the fact that an address somewhere is not initialised or a function can be called to undefined addresses.
- Remediation cost - medium, the detection is automatic through static analysis tools but bugfixes need to be done manually.

Table 3.4: Risk assessment calculation table for test 5

Risk assessment type	**Value**
Severity	3
Likelihood	2
Remediation cost	2
Priority level	$3x3x2$ = 18 (P18)
Overall risk level	L1

3.2.6 Test 6 - out-of-gas vulnerability

The sixth Solidity smart contract test contains out-of-gas vulnerabilities. Gas, as mentioned in section 2.1, is a tool to effectively “tax” the users of the blockchain in order to prevent excessive computation power usage. Also, the Ethereum network has a gas limit for a block and its computations cannot exceed that limit, and if the limit is exceeded, the contract automatically reverts to its previous state, thus creating DoS conditions. Dynamic array modifications and actions involving low-level calls inside loops are the most susceptible to possible DoS attacks. The example in the smart contract can be seen in listing 3.2.6, line 24 - the vulnerability here is that there is a low-level call *send*, and because each operation costs some amount of gas, this function call will eventually run out of gas because the loop is infinite.

Due to the nature of the DoS types of attacks, this security bug contains the availability property violation, as DoS attacks crash the systems on their targets and the systems become unavailable for some time. As for the risk assessment, the severity is low, as nothing more besides DoS can be done by the adversary. The likelihood type is “likely” because loops are very common programming constructs, therefore it is almost unavoidable that there will be no low-level calls in loops. However, these calls need to be protected adequately and fallback cases provided in case the contract runs out of gas. The remediation cost is medium, as out-of-gas vulnerabilities can be detected with static analysis tools. However, fixing would require programmers to patch vulnerable parts of the source code manually.

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.7.0;

contract outOfGas {

    address payable public owner;
    address payable public dest;

    uint counter = 0;

     constructor() public {
        owner = msg.sender;
    }

    function setDest() public {
        dest = msg.sender;
    }


    // Out of gas vulnerability due to infinite loop

    function run() public payable {
        for(uint i = 0; i >= 0; i++) {
            dest.send(msg.value);
            counter++;
        }
    }

}
```

Listing 3.2.6: Test No. 6 - possible DoS conditions in loops create out-of-gas vulnerabilities

Table 3.5: Risk assessment calculation table for test 6

Risk assessment type	Value
Severity	1
Likelihood	3
Remediation cost	2
Priority level	$1x3x2 = 6$ (P6)
Overall risk level	L2

3.2.7 Test 7 - gas griefing vulnerability

Test 7, like test 6 mentioned previously in section 3.2.6, is also related to gas-type security vulnerabilities. The source code of this test can be seen in listing 3.2.7, which was modified by Kaden Zipfel from ConsenSys Diligence company [3]. It was decided to include this example to show a bit more advanced types of vulnerabilities. The subtlety of this program lies in using sub-calls of contracts other than the original one. It is possible that one contract might have just enough gas to execute the whole transaction, while the sub-call might invoke an out-of-gas exception. The problem arises when the user is left with two options when no gas is left for the sub-call - either the transaction is reverted to its original state or it continues working as usual. This leads to a security vulnerability where a malicious user (a griefer) can supply the right amount of gas to complete the transaction, but the sub-call will fail. This action is not profitable for the attacker per se, but it causes damage to the victim of this attack - thus the action of *griefing*. This bug can be easily avoided by adding a guard statement to the target contract, checking if there is a sufficient amount of gas left for execution.

From the CIA triad perspective, the availability property is violated by the security bug in the code because the attacker can make the contract inaccessible to the victim using the contract, as the adversary can permanently send low amounts of gas to execute the transaction, but the sub-call will not have enough gas to complete successfully. The risk assessment's severity metric can be determined as "low", as only abnormal terminations of transactions can occur by abusing the security vulnerability in this test. The likelihood level is "probable", as it is difficult to take out-of-gas vulnerabilities into account, because technically they can happen anywhere in the contract provided that there is an insufficient amount of gas. However, programmers should pay more attention to sub-calls to other contracts and include guard statements checking for out-of-gas exceptions. The remediation cost is high, as static analysis tools have a hard time detecting this type of vulnerability[4], therefore programmers must rely on manual code reviewing methods.

3.2.8 Test 8 - usage of tx.origin for validation vulnerability

This is the eighth test in the series of Solidity smart contract vulnerability tests. This test contains a well-documented vulnerability of using tx.origin

[4]This is explained further in section 4.1.1.

```solidity
/*
 * @source:
↪  https://consensys.github.io/smart-contract-best-practices/known_attacks/#insufficient-gas-griefing
 * @author: ConsenSys Diligence
 * Modified by Kaden Zipfel (adapted Solidity versions by Nedas Matulevicius)
 */
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.24 <0.9.0;

contract Relayer {
    uint transactionId;

    struct Tx {
        bytes data;
        bool executed;
    }

    mapping (uint => Tx) transactions;

    function relay(Target target, bytes memory _data) public returns(bool) {
        // replay protection; do not call the same transaction twice
        require(transactions[transactionId].executed == false, 'same
↪  transaction twice');
        transactions[transactionId].data = _data;
        transactions[transactionId].executed = true;
        transactionId += 1;

        (bool success, ) =
↪  address(target).call(abi.encodeWithSignature("execute(bytes)", _data));
        return success;
    }
}

// Contract called by Relayer
contract Target {
    function execute(bytes memory _data) public {
        // Execute contract code
    }
}
```

Listing 3.2.7: Test No. 7 - gas griefing vulnerability [3]

Table 3.6: Risk assessment calculation table for test 7

Risk assessment type	Value
Severity	1
Likelihood	2
Remediation cost	1
Priority level	$1x2x3 = 6$ (P6)
Overall risk level	L2

(transaction origin address) as a value for validating if the owner of the contract is the same who initiated the transaction. This vulnerability is described in Solidity official documentation [87]. The problem of tx.origin is that it gets the original transaction creator address, not the address of the person interacting with the contract. The constructor sets the owner of this contract to the caller address (msg.sender) on line 10 (the source code can be seen in listing 3.2.8) and because on line 14 the code checks if the owner variable is the same as tx.origin, the adversary can set the destination address to his/her own address by creating a fallback function with this line of code:

```
TestContract(msg.sender).transferTo(owner, msg.sender.balance);
```

In this way, the contract, whose owner is set to the caller address, takes its address and sends all the caller funds to the adversary's wallet because the variable *owner* would be set to the adversary address, and tx.origin would take this address and compare with the address the adversary gave to the function, thus completing the transaction and leaving the victim with no cryptocurrency in the digital wallet.

```
1   // SPDX-License-Identifier: GPL-3.0
2   pragma solidity >0.5.0 <=0.7.0;
3
4   // tx.origin for authorisation - a vulnerability
5
6   contract TestContract {
7       address owner;
8
9       constructor() public {
10          owner = msg.sender;
11      }
12
13      function transferTo(address payable _destination, uint _amount) payable
    ↪   public {
14          require(tx.origin == owner);
15          _destination.transfer(_amount);
16      }
17  }
```

Listing 3.2.8: Test No. 8 - tx.origin gives a false sense of validation of the sender address, while it takes the original initiator address instead

In terms of the cybersecurity properties, the confidentiality property might be violated as it is probably unintended that one verifies the transaction creator address, not the payee address. The integrity property is violated as well due to the fact that the money transferred does not reach the intended destination, instead, it goes to the adversary's wallet, tampering with the data integrity of the contract. The severity from the risk assessment is medium, as data (digital currency) is transferred to the attacker address and the victim is left with nothing. The likelihood level is "unlikely", as there is plenty of examples of this type of vulnerability as well as the Solidity documentation mentions about the usage of tx.origin, alongside re-entrancy security vulnerability [87]. The remediation cost is medium, as the tx.origin validation can be detected easily with static analysis tools, although fixing might require manual patching by Solidity smart contract programmers.

Table 3.7: Risk assessment calculation table for test 8

Risk assessment type	Value
Severity	2
Likelihood	1
Remediation cost	2
Priority level	$2x1x2 = 4$ (P4)
Overall risk level	L3

3.2.9 Test 9 - unchecked value transfer and contract locking vulnerabilities

The ninth test used in this project is mainly concerned with contract locking and unchecked send statements, which can be considered as vulnerabilities. The source code of this test can be found in listing 3.2.9. The first vulnerability, the unchecked transfer of data, is similar to data send to arbitrary addresses described in test 5 in section 3.2.5, but the crucial difference here is that the contract can become locked if the *send()* function fails for various reasons. Locking means that the data transferred, in most cases cryptocurrency such as Ether, is rendered unusable and cannot be retrieved back, thus incurring financial losses for the user. The same problem occurs in LockContract contract, where the *withdraw* function claims to be capable of having payable functions, such as *send()* or *transfer()*, but it does not have any of them. Therefore, if the victim calls this function in the transaction, all Ether sent to this contract will be locked and lost as a consequence.

The integrity cybersecurity property from the CIA triad is violated, as the data at the victim's disposal is lost when interacting with functions that can lock cryptocurrency. Other properties, such as confidentiality or availability, are not violated. For the risk assessment, the severity level of this test is medium due to integrity property violation. The likelihood level is "probable", as it is easy to overlook functions without return or in this case, payable, statements inside the functions. The remediation cost is also medium - the contract locking properties can be detected by static analysis tools, but fixing requires manual effort from the programmers. All in all, the overall risk level of this security vulnerability is L2 - medium severity, probable, and has a medium cost to repair.

Table 3.8: Risk assessment calculation table for test 9

Risk assessment type	Value
Severity	2
Likelihood	2
Remediation cost	2
Priority level	$2x2x2 = 8$ (P8)
Overall risk level	L2

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >0.4.16 <=0.7.0;

contract SimpleStorage {
        uint storedData;

        function set(uint x) public {
                storedData = x;
        }

        function get(uint _data) public payable {
                msg.sender.send(_data);
        }

}
contract LockContract {
        function withdraw() public payable {
                uint a; uint b; uint c;
                c = a - b;
                // No withdrawal capacity to payable function - vulnerability
        }
}
```

Listing 3.2.9: Test No. 9 - unchecked *send()* function and missing withdrawal capabilities to payable functions are considered as vulnerabilities

3.2.10 Test 10 - double constructor vulnerability

This is the tenth test used in the project for testing out static analysis tools for Solidity smart contracts. This particular test is specific - it can be replicated only with Solidity compiler version 0.4.22. During the Solidity history, the special function *constructor()* did not exist prior to version 0.4.22 and the constructors were defined as functions with the same name as the contract [88]. This method was deprecated with Solidity version 0.5.0, but between versions 0.4.22 and 0.5.0 it was possible to have two constructors - one with the old method, and one with the new method. As constructors in object-oriented programming (OOP) languages are the very first functions to be called to initialise objects in a given class (in Solidity's case, the class is named as "contract"), and having several functions being incorrectly overloaded can lead to the race condition, especially if both constructors are involved in initialising the same variables. This is exactly what is happening on lines 10-15 in listing 3.2.10. Had one of the constructors taken different arguments, the constructors could work perfectly well, as they can be chosen accordingly given different arguments to constructors. However, in this code snippet, it can be seen that both constructors take the same number and type of arguments (zero), therefore it cannot be determined at runtime what the value of the variable *minter* will be. It can either be the sender address, or it can be a hard-coded address 0xdeadbeef.

Since Solidity 0.5.0 multiple constructors are not allowed, and smart contracts can have only one type of constructor - the special function *constructor()*. In a refined version of test 10, test 10.1, seen in listing 3.2.11, it can happen during refactoring process that a programmer mistyped the old constructor name and it suddenly becomes a publicly available function with the capabilities of a constructor. This is especially dangerous, as an adversary could call

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity 0.4.22;

contract Coin {
        address public minter;
        mapping (address => uint) public balances;

        event Sent(address from, address to, uint amount);
        // Two constructors - old version (pre 0.5) and new version. Using both
↪  can lead to some unexpected results. Works only with Solidity v. 0.4.22.
        function Coin() public {
                minter = msg.sender;
        }
        constructor() public {
                minter = address(0xdeadbeef);
        }

        function mint(address receiver, uint amount) public {
                require(msg.sender == minter);
                require(amount < 1e60);
                balances[receiver] += amount;
        }

        function send(address receiver, uint amount) public {
                if (amount > balances[msg.sender])
                    revert("Insufficient balance");
                balances[msg.sender] -= amount;
                balances[receiver] += amount;
                emit Sent(msg.sender, receiver, amount);
        }
}
```

Listing 3.2.10: Test No. 10 - double constructors lead to non-deterministic variable initialisation and race conditions occur

the constructor and then the constructor-like function, thus establishing himself/herself as the owner of the contract. This contract hijacking, combined with single-signature self-destruction functions mentioned in section 3.2.5 can cause havoc in even the most sophisticated systems deployed on the blockchain. The Parity wallet cyberattack happened due to a similar reason - the attacker could make himself/herself the owner of the contract and then, he/she abused the unprotected self-destruct function to kill the contract and all the transactions interacting with the contract lost all of their digital currency [86].

```
1   // SPDX-License-Identifier: GPL-3.0
2   pragma solidity <0.9.0;
3
4   contract Coin {
5           address public minter;
6           mapping (address => uint) public balances;
7
8           event Sent(address from, address to, uint amount);
9           // Two constructors - old version (pre 0.5) and new version. Using both
    ↪  can lead to some unexpected results
10          // Same as test10.sol, but the old type constructor contains a typo,
    ↪  meaning that the compiler would not recognise it as a candidate to a
    ↪  constructor.
11          function coin() public {
12                  minter = msg.sender;
13          }
14          constructor() public {
15                  minter = address(0xdeadbeef);
16          }
17
18          function mint(address receiver, uint amount) public {
19                  require(msg.sender == minter);
20                  require(amount < 1e60);
21                  balances[receiver] += amount;
22          }
23
24          function send(address receiver, uint amount) public {
25                  if (amount > balances[msg.sender])
26                      revert("Insufficient balance");
27                  balances[msg.sender] -= amount;
28                  balances[receiver] += amount;
29                  emit Sent(msg.sender, receiver, amount);
30          }
31  }
```

Listing 3.2.11: Test No. 10.1 - accidental typo on line 11 exposes the function with constructor features to the users, allowing the contract to be hijacked

In terms of cybersecurity properties, the integrity of the contract is affected because of the race condition between constructors – one might not know what the value of the variable can be. The availability property is also violated as the values of initialised variables would not be constant and users cannot rely on the same values with the same execution paths when the specific data is requested by users. The risk assessment of Test 10 and Test 10.1 is as follows:

- Severity - medium, due to data integrity violations.
- Likelihood - unlikely, as only very old contracts can be affected with the

multiple constructor problem, although constructor-like functions can occur in all types of smart contracts.

- Remediation cost - high, while explicit multiple constructors can be detected, typos are extremely difficult to detect by static analysis tools, as they would think that the constructor-like function is an ordinary function with the capability of data assignment to variables.

Table 3.9: Risk assessment calculation table for test 10 and test 10.1

Risk assessment type	**Value**
Severity	2
Likelihood	1
Remediation cost	3
Priority level	$2x1x3 = 6$ (P6)
Overall risk level	L2

3.2.11 Test 11 - inline assembly code usage vulnerability

This is the penultimate test in the series of Solidity smart contract vulnerabilities. The source code of this test in listing 3.2.12 contains some inline assembly code, which is one of the more interesting and powerful Solidity features available. Inline assembly code allows better control of operations done inside contracts, as it allows accessing directly the EVM. However, this is a double-edged sword - while programmers using inline assembly can write faster contracts or libraries, assembly bypasses some Solidity compiler security checks and if the assembly code contains vulnerabilities, it can have severe consequences to the contract. This particular smart contract does not have any vulnerabilities in its code, but the fact that it contains inline assembly can be treated as a possible security vulnerability and the programmer is responsible for having correct inline assembly code. The Solidity compiler nor static analysis tools will tell if there are bugs inside inline assembly code snippets.

In the case of an exploited security bug in inline assembly, it can be treated that all three of CIA triad's cybersecurity properties would be violated, as the adversary would be able to execute arbitrary code and inject it directly to the EVM. For the risk assessment, the severity level is high due to the reasoning, while the likelihood type is "unlikely" because

a. it is strongly discouraged to use inline assembly code in Solidity smart contracts,

b. the applications of inline assembly are mainly limited to libraries and their functionalities.

The remediation cost is medium because the inline assembly usage can be detected by static analysis tools, but converting assembly code to high-level Solidity code can be a task requiring a substantial amount of manual refactoring by the programmers.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

library VectorSum {
    function sumSolidity(uint[] memory _data) public pure returns (uint sum) {
        for(uint i = 0; i < _data.length; i++)
        sum += _data[i];
    }
    function sumAsm(uint[] memory _data) public pure returns (uint sum) {
       for(uint i = 0; i < _data.length; i++) {
         assembly {
          sum := add(sum, mload(add(add(_data, 0x20), mul(i, 0x20))))
          }
       }
    }
    function sumPureAsm(uint[] memory _data) public pure returns (uint sum) {
       assembly {
         let len := mload(_data)
         let data := add(_data, 0x20)
         for
        { let end := add(data, mul(len, 0x20)) }
           lt(data, end)
        { data := add(data, 0x20) }
          {
             sum := add(sum, mload(data))
           }
        }
     }
}
```

Listing 3.2.12: Test No. 11 - inline assembly usage in Solidity is discouraged due to security bypasses

Table 3.10: Risk assessment calculation table for test 11

Risk assessment type	Value
Severity	3
Likelihood	1
Remediation cost	2
Priority level	$3x1x2 = 6$ (P6)
Overall risk level	L2

3.2.12 Test 12 - no vulnerabilities (code size test)

The last test in the project used is a combination of all tests amalgamated into one large Solidity file with lots of contracts and functions. Of course, the test does contain vulnerabilities as its code is taken from previous tests, the main objective of this test is to check whether static analysis tools are prone to "code explosion", that is, the time taken for the static analysis tool to verify the Solidity file increases proportionally to the amount of code written in the file. This can expose problems some static analysers might face, for example, one of the surveys [89] show that by obfuscating code including data location and usage emulation one can trick the analyser into thinking that some pieces of code are relevant to the contract and need to be checked for, thus slowing down the overall verification process. If the static analysis tool is implemented using BMC method, then the control-flow graph (CFG) is pruned to a given bound, either set by the user or by default, therefore the analyser maintains speed and accuracy regardless of the code size.

As the test is not concerned with any cybersecurity properties or vulnerabilities, therefore the CIA triad and the risk assessment based on the SEI C Coding Standard does not apply for this Solidity smart contract.

3.3 Summary

This chapter explained the methodology used in this project, raised the hypothesis and gave more in-depth knowledge about each of the static analysis tools and benchmarking tools used in the testing process. Also, each test was described in great detail with code examples, cybersecurity property evaluations and risk assessments.

Chapter 4

Project evaluation and analysis

This chapter gives more detail about the testing results from running static analysis tools mentioned in section 3.1.2 with tests described in section 3.2. The first section explains what kinds of data were measured and collected for analysis with descriptions of each of the metrics. The second section shows the setup and the performance of all static analysis tools with regards to their accuracy rate and resource consumption rate in a graphical form. The third section briefly explains the objectives of the testing. The fourth section analyses the obtained results and presents evaluations and recommendations according to the analysis. The last section determines if the hypothesis given in section 3.1.1 holds and finds out the best static analysis tool in terms of all the metrics observed and evaluated.

4.1 Description of benchmarks

Overall, all five static analysis tools were measured with all 13 tests[1]. Each test was run 10 times with each static analysis tool, therefore each static analysis tool was run 130 times. In total, 650 test runs were done to obtain the data represented in tables 4.1 to 4.9. Table 4.1 shows the running times of each test for static analysis tools. As the verifiers do not run perfectly at a constant time, each static analysis tool running time is represented in lower and upper bounds, determined from repeated test runs. It is also worth noting that the time given for a static analysis tool to run is 10 minutes - if the tool exceeds this threshold, the test run is considered to be a timeout, and the threshold time of 10 minutes is written in the table. The timeout runs are marked in red to improve the visibility of the table. The average time for lower and upper bounds is also calculated and displayed separately in the last row of the table.

The benchmarks were divided into several parts:

- Normal conditions (time and accuracy test) - all 13 tests were run under the ordinary operation of the laptop without any extra processor or

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

memory load. The primary objective was to obtain average running times of code verification of each of the smart contract analysers and to check the outcomes of static analysis tools as well.

- Normal conditions (resource management 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 for each smart contract test.
- 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. Every static analysis tool used all tests in order to get results represented in table 4.6.
- Stress test 2 (77% memory load) - all tests were run under the 77% of RAM usage by executing several commands in order to artificially increase the memory load. The objective was similar to other tests - observe the behaviour of analysers and record running times.
- Stress test 3 (90% memory load) - 13 tests were run under the 90% of RAM usage. The objective is the same as for the stress test 2.
- 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 behaviour and get the average running times for each static analysis tool used with all source code of test smart contracts verified.

4.1.1 Normal conditions (time and accuracy test)

As it can be seen from the data in the table, only Oyente and Mythril managed to produce timeouts for test runs. Oyente timed out on the sixth test about gas griefing by supplying less gas for sub-call execution, explained in detail in section 3.2.7, while Mythril timed out on the 11th test about the inline assembly code, explained in detail in section 3.2.11. As the source code for Test 7 contains several contracts, it is possible that Oyente produced a very large CFG during the code verification process and that is why it ran over ten minutes. It is worth noting that compared to other Oyente runs, it is an outlier, although Tests 3 and 11 also ran significantly slower compared to other tests. Mythril, on the other hand, tended to run slower on average compared to the remaining static analysis tools, but Test 11 is a significant outlier here. Therefore, it can be safely assumed that Mythril's code verification method does not handle well inline assembly code written in Solidity, as it probably has a hard time abstracting the assembly code.

Overall, looking at the averages in table 4.1, no static analysis tool took over 2 minutes on average to verify Solidity smart contract code. However, the averages might have been better if Oyente's and Mythril's outliers would have been discarded from the evaluation. The fastest static analysis tool on average is

Table 4.1: Time performance table under normal conditions (minutes:seconds.milliseconds)

Test No.	**Remix low**	**Remix high**	**Slither low**	**Slither high**	**Oyente low**	**Oyente high**	**Mythril low**	**Mythril high**	**SmartCheck low**	**SmartCheck high**
T1	00:00.660	00:00.780	00:00.543	00:00.571	00:00.631	00:00.724	00:04.782	00:04.897	00:01.612	00:01.768
T2	00:00.460	00:00.790	00:00.566	00:00.597	00:01.023	00:01.265	00:46.805	00:48.606	00:01.828	00:02.106
T3	00:00.720	00:00.740	00:00.610	00:00.647	07:59.057	09:00.192	00:21.686	00:28.760	00:01.869	00:02.279
T4	00:00.520	00:00.920	00:00.558	00:00.582	00:26.215	00:28.911	00:07.951	00:09.177	00:01.671	00:01.786
T5	00:00.590	00:00.720	00:00.568	00:00.597	00:01.062	00:01.113	02:03.789	02:08.459	00:01.788	00:02.154
T6	00:00.460	00:00.720	00:00.556	00:00.576	00:03.286	00:03.925	00:34.148	00:36.964	00:01.760	00:01.968
T7	00:00.590	00:00.780	00:00.587	00:00.600	10:00.000	10:00.000	00:54.149	00:58.926	00:01.759	00:01.867
T8	00:00.650	00:00.710	00:00.556	00:00.580	00:00.648	00:00.693	00:12.458	00:14.567	00:01.611	00:02.159
T9	00:00.460	00:00.720	00:00.556	00:00.591	00:00.698	00:00.724	00:45.638	00:52.222	00:01.658	00:01.874
T10	00:00.720	00:00.790	00:00.574	00:00.597	00:01.631	00:01.761	00:22.714	00:25.609	00:01.907	00:02.096
T10.1	00:00.590	00:00.660	00:00.572	00:00.602	00:02.411	00:03.330	00:36.355	00:39.087	00:01.714	00:01.919
T11	00:00.990	00:01.650	00:00.572	00:00.611	05:31.432	05:56.100	10:00.000	10:00.000	00:01.704	00:01.855
T12	00:00.920	00:01.250	00:00.877	00:01.020	01:32.450	01:52.561	05:02.722	06:16.913	00:02.807	00:03.183
Average	00:00.641	00:00.864	00:00.592	00:00.629	01:58.503	02:07.023	01:41.015	01:49.553	00:01.822	00:02.078

Table 4.2: Time performance table under normal conditions with outliers discarded (minutes:seconds.milliseconds)

Test No.	**Remix low**	**Remix high**	**Slither low**	**Slither high**	**Oyente low**	**Oyente high**	**Mythril low**	**Mythril high**	**SmartCheck low**	**SmartCheck high**
T1	00:00.660	00:00.780	00:00.543	00:00.571	00:00.631	00:00.724	00:04.782	00:04.897	00:01.612	00:01.768
T2	00:00.460	00:00.790	00:00.566	00:00.597	00:01.023	00:01.265	00:46.805	00:48.606	00:01.828	00:02.106
T3	00:00.720	00:00.740	00:00.610	00:00.647	07:59.057	09:00.192	00:21.686	00:28.760	00:01.869	00:02.279
T4	00:00.520	00:00.920	00:00.558	00:00.582	00:26.215	00:28.911	00:07.951	00:09.177	00:01.671	00:01.786
T5	00:00.590	00:00.720	00:00.568	00:00.597	00:01.062	00:01.113	02:03.789	02:08.459	00:01.788	00:02.154
T6	00:00.460	00:00.720	00:00.556	00:00.576	00:03.286	00:03.925	00:34.148	00:36.964	00:01.760	00:01.968
T7	00:00.590	00:00.780	00:00.587	00:00.600	—DISCARDED—	—DISCARDED—	00:54.149	00:58.926	00:01.759	00:01.867
T8	00:00.650	00:00.710	00:00.556	00:00.580	00:00.648	00:00.693	00:12.458	00:14.567	00:01.611	00:02.159
T9	00:00.460	00:00.720	00:00.556	00:00.591	00:00.698	00:00.724	00:45.638	00:52.222	00:01.658	00:01.874
T10	00:00.720	00:00.790	00:00.574	00:00.597	00:01.631	00:01.761	00:22.714	00:25.609	00:01.907	00:02.096
T10.1	00:00.590	00:00.660	00:00.572	00:00.602	00:02.411	00:03.330	00:36.355	00:39.087	00:01.714	00:01.919
T11	00:00.990	00:01.650	00:00.572	00:00.611	05:31.432	05:56.100	—DISCARDED—	—DISCARDED—	00:01.704	00:01.855
T12	00:00.920	00:01.250	00:00.877	00:01.020	01:32.450	01:52.561	05:02.722	06:16.913	00:02.807	00:03.183
Average	00:00.641	00:00.864	00:00.592	00:00.629	01:18.379	01:27.608	00:59.433	01:08.682	00:01.822	00:02.078

Slither, with the average varying from 592 to 629 milliseconds, while the slowest is Oyente with the average varying from 1 min 58 seconds to 2 min 7 seconds. If we discard the timeouts from Oyente and Mythril, the averages improve to 1min 18s - 1min 27s and 59s - 1min 8s respectively, as it can be seen in table 4.2.

Table 4.3 shows the accuracy of each Solidity smart contract static analysis tool. The table shows if the vulnerability (or the lack of it) was detected correctly, and is denoted by the capital letter Y (Yes). If the verifier did not accurately detect the vulnerability, it is shown as a capital letter N (No) in the table. For better readability and visual comprehension, the results in the table are colour-coded, with green standing for a hit and red standing for a miss. The last row of the table shows the total accuracy percentage of each static analysis tool tested.

Table 4.3: Static analysis accuracy table under normal conditions (Y - vulnerability caught, N - vulnerability not caught)

Test No.	Remix	Slither	Oyente	Mythril	SmartCheck
T1	Y	Y	Y	Y	Y
T2	Y	Y	Y	Y	N
T3	N	N	N	N	N
T4	Y	Y	N	N	N
T5	Y	Y	N	Y	N
T6	Y	Y	Y	N	N
T7	N	N	N	N	N
T8	Y	Y	N	Y	N
T9	Y	Y	N	Y	Y
T10	N	Y	N	N	N
T10.1	N	N	N	N	N
T11	Y	Y	N	N	Y
Accuracy %	66.67%	75.00%	25.00%	41.67%	25.00%

Looking strictly from the accuracy perspective, it can be clearly derived that Slither is the winner here with three-quarters of all tests being correctly verified and vulnerabilities found (or the lack of it, see Test 1 in section 3.2.1). However, the data shows that some static analysis tools are better defined to test some specific vulnerabilities. For example, Oyente performed poorly on nearly all tests but it managed to correctly find the out-of-gas vulnerability inside the source code of Test 6. Mythril also did not show excellent results, but it coped well with lost contract and unprotected self-destruction finding as well as using tx.origin for validation. SmartCheck found the contract locking vulnerabilities in Test 9 and correctly identified inline assembly code in Test 11.

There were some tests, which were not classified correctly by any of the static analysis tools, though. Tests 3 (dead code, section 3.2.3), 7 (gas griefing, section 3.2.7) and 10.1 (constructor-like functions, section 3.2.10) were the trickiest for the verifiers tested out. It can be found to be interesting that Slither advertises in its documentation that it is able to detect dead code [90], but the results show that it was not able to do so. It should be noted, however, that the example given in the documentation is very simple, and it can be the case that

Slither flags unimplemented code as dead, which is true, but the unreachable or meaningless but functional parts of the code can also be classified as dead code.

Furthermore, there were cases where static analysis tools should, at least in theory, detect a particular vulnerability in Solidity smart contract code, but it did not detect anything related to the vulnerability searched. For example, SmartCheck did not find any re-entrancy bugs, although the printout tells us that it is capable of detecting re-entrancies in smart contracts[2].

4.1.2 Normal conditions (resource management test)

As mentioned in the aims of this project in section 1.1, accuracy is not the only metric that was measured in order to find out the best static analysis tool. Resource consumption rates were tracked as well, specifically CPU and memory usage rates, and the results are shown in tables 4.4 and 4.5 respectively. It can be seen in the CPU consumption table 4.4 that some static analysis tools tend to have fluctuating rates of consumption, while others take the greedy approach and use all available power the CPU can give. The most efficient static analysis tool is the Remix IDE plug-in, with CPU lows of 4.031% on average. However, it has a large margin between the lowest point of consumption and the highest, indicating that if the Remix IDE plug-in would run longer, then it would consume more CPU power. But, as it can be seen in table 4.1, Remix IDE static analysis plug-in verified the tests in a bit more than half a second on average, therefore, one can forgive the analyser for consuming a bit more CPU power at the expense of speed. Some tools, such as Mythril, are CPU-hungry and took all of its power for verification purposes. It must be noted, however, that Mythril uses only one core and does not utilise parallel computing capabilities, thus, the numbers can look a bit misleading. Overall, a standard run would take about 25% of CPU power from a computer for a second or two, although Mythril is an exception, as it is one of the slowest static analysers observed.

In terms of memory consumption, all static analysis tools showed good results, and table 4.5 displays that only Oyente tended to sometimes use more significantly more memory. It can also be noticed that there is a correlation between high CPU and memory consumption in verifying troublesome tests for Oyente - Tests 7 and 11 were the ones where Oyente timed out by exceeding the 10-minute threshold, and both CPU and memory usage tables show that for these tests the upper bound is close to 100% of resource usage. Mythril used more memory for these tests where there was more source code, as Test 3 contains a lot of useless code, while Test 12 has around 300 lines of source code waiting to be verified.

4.1.3 Stress test 1: maximum CPU load

Another interesting way of checking the robustness of static analysis tools is to put them under pressure. The following tables 4.6, 4.7, 4.8 and 4.9 contain results obtained when static analysis tools were run under unusual CPU and/or memory usage. Obviously, all of the tests were checked for vulnerabilities slower than under normal operating conditions, but some showed a significant increase

[2]For the printout example, see figure 3.8 for details.

Table 4.4: CPU consumption rate for each test running in % of total CPU usage

Test No.	**Remix low**	**Remix high**	**Slither low**	**Slither high**	**Oyente low**	**Oyente high**	**Mythril low**	**Mythril high**	**SmartCheck low**	**SmartCheck high**
T1	2.5	6.3	20.7	25.1	14.8	35.5	77.1	100	16	81.5
T2	2.5	11.9	17.6	25	12.6	39.5	86.7	100	19.8	71.7
T3	1.9	11.4	10.6	28.8	77.6	100	98.9	100	18.9	96.1
T4	4.4	12.5	10.6	26.2	2.7	24.8	100	100	17	71.7
T5	3.1	19.5	15.9	26.2	11.8	42.1	100	100	18.8	69.1
T6	3.7	17	12.6	21.9	15.6	66.2	99.4	100	18.5	88.7
T7	5	19.5	7.5	22.5	22.3	100	100	100	37.3	100
T8	3.7	17.6	21.9	25.6	13.2	20.1	97	100	17.6	86.7
T9	5	20.6	6.2	22.6	16.3	18.3	99.7	100	34.1	94
T10	5	23.2	6.3	25.6	12.6	39.7	99.5	100	20.1	96.4
T10.1	7.5	20.8	16.8	24.5	12.5	31.4	98.9	100	36.9	100
T11	3.1	28	10.7	26.9	43.1	100	98.3	100	29.1	100
T12	5	26.2	15.7	22.6	93.7	100	83.1	100	17.4	100
Average	4.031	18.038	13.315	24.885	26.831	55.200	95.277	100.000	23.192	88.915

Table 4.5: Memory consumption rate for each test running in % of total RAM usage

Test No.	**Remix low**	**Remix high**	**Slither low**	**Slither high**	**Oyente low**	**Oyente high**	**Mythril low**	**Mythril high**	**SmartCheck low**	**SmartCheck high**
T1	4.8	7.2	0.4	0.4	0.7	0.9	1.7	2.1	6.2	7.7
T2	4.9	7.2	0.4	0.4	0.7	0.9	2.2	2.3	5.5	7.5
T3	5.4	7.1	0.3	0.4	3	50.7	20	20.7	5	7.3
T4	4.6	7.1	0.3	0.4	2.7	24.8	1.8	2	6	7
T5	4.8	7.4	0.4	0.4	0.7	0.9	1.9	2.3	5.5	7
T6	5	7.2	0.4	0.4	0.8	1.2	2	2.2	6	7
T7	5	7.1	0.3	0.4	2.1	85.2	3.4	3.5	4.5	8
T8	5.3	7.1	0.4	0.4	0.7	0.9	2	3.5	5	6.5
T9	5.4	7.1	0.3	0.4	0.8	0.9	2	2.4	5.5	7
T10	5.2	5.5	0.2	0.4	0.7	0.9	2.1	2.3	6	8
T10.1	5.7	7.2	0.4	0.4	0.7	0.9	2.3	2.5	5.5	9
T11	5.8	7.6	0.3	0.4	1.9	30.5	1.9	5.5	3.5	7
T12	6	7.6	0.4	0.4	3.4	51.2	2.2	41.4	4.9	7.5
Average	5.223	7.108	0.346	0.400	1.454	19.223	3.500	7.131	5.315	7.423

in time compared to their counterparts. For example, SmartCheck's average running time more than doubled compared to the ordinary resource CPU usage rate - the same applies to Remix IDE static analysis plug-in. Oyente and Mythril, generally running slower under normal conditions, showed some robustness under 100% of CPU load - their running times increased for 30 seconds but compared to the results seen in table 4.1 the increase in operating time is marginal. On the other hand, Oyente timed out thrice, reducing its overall performance rate by one test, while Mythril timed out on the same test with inline assembly code.

4.1.4 Stress tests 2 and 3: 77% and 90% of memory loads

The memory stress tests, together with static analysis tools, were run and results are compiled into tables 4.7 and 4.8. It was not possible to get 100% of memory load because even if locking the analyser kernel killing priority to -1000 might have helped for several seconds, there was a good chance of crashing the computer, therefore, the safer option of 90% memory load was chosen. The data shows that Remix IDE static analysis plug-in and Slither were only slightly impacted by the increased memory usage, while Oyente and Mythril struggled with the fact that they could use less memory for code verification. SmartCheck showed consistent results regardless of the memory load, meaning that it was well-optimised in terms of memory usage. The timeouts were the same as in the CPU stress test - 3 timeouts from Oyente and 1 timeout from Mythril.

4.1.5 Stress test 4: maximum CPU load and 90% memory load

A full stress test was conducted on the testing laptop, with full CPU load and 90% memory load. The results can be found in table 4.9. The best performance was shown by Remix IDE static analysis plug-in and Slither, while the worst performance excluding timeouts was shown by SmartCheck. Oyente timed out three times, the same amount as in previous stress tests, while Mythril managed to time-out twice - in Tests 11 and 12. One fact can be found interesting - while Test 12 had a significantly larger codebase and included code snippets from Test 3, Oyente has verified it in the given bounds. On the other hand, as it has now become customary, Test 3 was not verified in the threshold of 10 minutes.

4.1.6 Summary of tests

Overall, from the resource consumption perspective, Remix IDE plug-in leads in the best CPU usage rates while Slither is the best in managing memory usage. Therefore, lower-end computers will be able to handle both tools pretty well. The worst result in CPU rating was displayed by Mythril and SmartCheck consumes the most memory on average. On the other hand, Oyente 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, again Remix IDE plug-in and Slither showed the best results while having zero timeouts. The third place goes to SmartCheck because it had no timeouts, although stress conditions are not the ones SmartCheck can deal with smoothly. The worst performance is shown

Table 4.6: Time performance table under 100% of CPU load (minutes:seconds.milliseconds)

Test No.	**Remix low**	**Remix high**	**Slither low**	**Slither high**	**Oyente low**	**Oyente high**	**Mythril low**	**Mythril high**	**SmartCheck low**	**SmartCheck high**
T1	00:00.860	00:01.190	00:00.830	00:01.275	00:01.142	00:01.889	00:09.532	00:11.576	00:03.894	00:04.565
T2	00:00.860	00:01.060	00:00.941	00:01.306	00:02.064	00:03.368	01:15.937	01:20.340	00:04.535	00:04.949
T3	00:00.730	00:01.120	00:01.108	00:01.369	10:00.000	10:00.000	00:33.530	00:36.448	00:04.559	00:05.325
T4	00:00.790	00:00.990	00:00.806	00:01.300	00:46.896	00:53.556	00:16.739	00:20.434	00:04.277	00:04.803
T5	00:00.790	00:01.200	00:00.920	00:01.250	00:01.972	00:02.440	03:03.558	03:19.372	00:04.427	00:05.528
T6	00:00.720	00:00.990	00:00.948	00:01.452	00:06.183	00:08.003	00:53.281	00:58.408	00:04.263	00:04.653
T7	00:01.190	00:01.650	00:01.055	00:01.217	10:00.000	10:00.000	01:43.641	01:58.932	00:04.153	00:04.980
T8	00:00.790	00:01.060	00:00.956	00:01.336	00:01.182	00:01.562	00:22.861	00:31.640	00:04.014	00:04.885
T9	00:00.790	00:01.260	00:00.963	00:01.269	00:01.420	00:01.975	01:03.431	01:13.563	00:04.097	00:05.022
T10	00:00.920	00:01.260	00:00.912	00:01.327	00:03.241	00:03.839	00:47.827	00:55.116	00:04.513	00:05.118
T10.1	00:01.660	00:01.850	00:00.895	00:01.329	00:03.530	00:04.167	01:16.972	01:25.306	00:04.396	00:05.559
T11	00:01.250	00:01.640	00:01.002	00:01.298	10:00.000	10:00.000	10:00.000	10:00.000	00:04.503	00:05.000
T12	00:02.110	00:02.440	00:01.551	00:01.977	02:34.049	02:40.095	07:48.405	08:31.888	00:06.336	00:07.347
Average	00:01.035	00:01.362	00:00.991	00:01.362	02:35.514	02:36.992	02:15.055	02:24.848	00:04.459	00:05.210

Table 4.7: Time performance table under 77% of memory load (minutes:seconds.milliseconds)

Test No.	Remix low	Remix high	Slither low	Slither high	Oyente low	Oyente high	Mythril low	Mythril high	SmartCheck low	SmartCheck high
T1	00:00.730	00:00.920	00:00.610	00:00.670	00:00.678	00:01.361	00:06.245	00:07.251	00:02.005	00:03.012
T2	00:00.720	00:00.930	00:00.642	00:00.966	00:01.165	00:01.540	01:09.307	01:23.320	00:02.339	00:03.101
T3	00:00.720	00:00.990	00:00.676	00:00.726	10:00.000	10:00.000	00:33.530	00:36.448	00:02.222	00:02.919
T4	00:00.730	00:00.860	00:00.621	00:00.936	00:31.736	00:33.369	00:09.027	00:09.436	00:02.105	00:02.653
T5	00:00.600	00:00.860	00:00.633	00:00.958	00:01.220	00:01.687	02:11.122	02:16.546	00:02.301	00:03.239
T6	00:00.790	00:00.850	00:00.621	00:00.945	00:03.716	00:04.757	00:34.768	00:37.876	00:02.120	00:02.567
T7	00:01.260	00:01.510	00:00.637	00:00.714	10:00.000	10:00.000	01:01.991	01:12.696	00:02.161	00:02.860
T8	00:00.660	00:00.850	00:00.616	00:00.677	00:00.726	00:00.997	00:14.105	00:19.954	00:02.084	00:02.545
T9	00:00.720	00:00.790	00:00.624	00:00.949	00:00.817	00:01.055	00:48.230	00:53.717	00:02.088	00:02.728
T10	00:00.600	00:00.920	00:00.636	00:00.919	00:01.857	00:02.075	00:24.944	00:26.197	00:02.221	00:02.851
T10.1	00:01.000	00:01.320	00:00.637	00:00.976	00:01.877	00:02.081	00:43.461	00:51.711	00:02.248	00:02.714
T11	00:01.120	00:01.330	00:00.639	00:00.970	10:00.000	10:00.000	10:00.000	10:00.000	00:02.190	00:02.406
T12	00:01.590	00:01.780	00:00.991	00:01.445	01:49.484	03:26.433	03:07.034	03:44.444	00:03.274	00:04.247
Average	00:00.865	00:01.070	00:00.660	00:00.912	02:30.252	02:38.104	01:37.213	01:44.584	00:02.258	00:02.911

Table 4.8: Time performance table under 90% of memory load (minutes:seconds.milliseconds)

Test No.	Remix low	Remix high	Slither low	Slither high	Oyente low	Oyente high	Mythril low	Mythril high	SmartCheck low	SmartCheck high
T1	00:01.180	00:02.620	00:00.655	00:01.172	00:00.709	00:01.142	00:06.245	00:07.251	00:02.337	00:02.707
T2	00:00.920	00:00.980	00:00.703	00:01.093	00:01.240	00:01.981	01:09.555	01:24.048	00:02.224	00:02.733
T3	00:00.920	00:01.260	00:00.733	00:01.163	10:00.000	10:00.000	00:41.086	01:00.643	00:02.215	00:02.565
T4	00:00.860	00:00.980	00:00.689	00:01.179	00:32.802	00:36.581	00:08.997	00:10.447	00:02.021	00:02.757
T5	00:00.860	00:01.590	00:00.670	00:01.425	00:01.260	00:01.801	02:15.928	02:23.866	00:02.122	00:02.399
T6	00:00.860	00:00.920	00:00.677	00:01.170	00:04.153	00:04.436	00:34.978	00:39.428	00:02.046	00:02.501
T7	00:01.380	00:01.650	00:00.693	00:01.132	10:00.000	10:00.000	01:00.595	01:12.837	00:02.507	00:02.935
T8	00:01.190	00:01.580	00:00.680	00:01.121	00:00.781	00:00.930	00:16.471	00:20.851	00:01.875	00:02.081
T9	00:00.790	00:01.260	00:00.651	00:01.146	00:00.810	00:00.974	00:48.727	00:56.575	00:02.024	00:02.182
T10	00:00.720	00:00.990	00:00.695	00:01.121	00:02.038	00:02.869	00:24.740	00:32.956	00:02.043	00:02.822
T10.1	00:01.050	00:01.910	00:00.714	00:01.203	00:02.067	00:02.748	00:43.986	00:53.128	00:02.347	00:05.231
T11	00:01.450	00:01.980	00:00.713	00:01.148	10:00.000	10:00.000	10:00.000	10:00.000	00:01.730	00:02.070
T12	00:01.790	00:03.290	00:01.011	00:01.526	02:16.684	04:55.551	05:04.400	05:46.098	00:03.872	00:05.776
Average	00:01.075	00:01.616	00:00.714	00:01.200	02:32.503	02:45.309	01:47.362	01:57.548	00:02.259	00:02.981

Table 4.9: Time performance table under full stress test (full CPU and memory load) in minutes:seconds.milliseconds

Test No.	Remix low	Remix high	Slither low	Slither high	Oyente low	Oyente high	Mythril low	Mythril high	SmartCheck low	SmartCheck high
T1	00:01.520	00:01.840	00:01.034	00:02.391	00:01.635	00:02.653	00:13.918	00:15.425	00:04.447	00:05.575
T2	00:01.380	00:01.780	00:01.537	00:03.411	00:02.462	00:04.034	01:39.136	01:46.878	00:05.064	00:06.600
T3	00:01.320	00:01.980	00:01.891	00:02.913	10:00.000	10:00.000	00:30.224	00:31.975	00:05.184	00:06.518
T4	00:01.390	00:01.980	00:01.488	00:02.835	01:03.560	03:32.765	00:23.247	00:25.722	00:04.124	00:05.666
T5	00:01.380	00:02.840	00:01.677	00:02.262	00:02.572	00:03.889	03:25.531	03:59.666	00:04.885	00:06.846
T6	00:01.450	00:01.850	00:01.313	00:02.215	00:08.887	00:11.362	01:04.859	01:15.693	00:04.645	00:05.958
T7	00:02.840	00:03.350	00:01.413	00:02.151	10:00.000	10:00.000	02:31.621	02:45.224	00:04.763	00:05.819
T8	00:01.390	00:02.440	00:01.426	00:02.196	00:01.679	00:01.966	00:38.198	00:43.429	00:04.771	00:07.657
T9	00:01.450	00:01.910	00:01.298	00:02.455	00:02.110	00:03.113	01:16.157	01:27.590	00:04.259	00:06.217
T10	00:01.330	00:01.720	00:01.414	00:02.527	00:03.841	00:06.182	01:02.831	01:13.302	00:04.754	00:06.548
T10.1	00:01.920	00:02.960	00:01.521	00:02.054	00:05.158	00:06.269	01:38.456	01:54.801	00:04.750	00:06.092
T11	00:01.990	00:02.560	00:01.511	00:02.262	10:00.000	10:00.000	10:00.000	10:00.000	00:04.775	00:05.914
T12	00:03.690	00:03.880	00:01.989	00:02.472	03:58.817	06:08.307	10:00.000	10:00.000	00:06.756	00:08.721
Average	00:01.773	00:02.392	00:01.501	00:02.473	02:43.902	03:06.195	02:38.783	02:47.670	00:04.860	00:06.472

by Oyente with three timeouts and the longest running times on average for both CPU and memory stress tests.

4.2 Setup

The setup and execution process is very simple in order to run the tests mentioned in chapter 3. To start off, the equipment and tools used are explained in section 3.1.3. 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 analysers, 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* [91] 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:

- 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.
- Slither: `slither test2.sol`
- Oyente: `oyente -s test2_cpy.sol`
- Mythril: `myth analyze test2.sol`
- 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 4.10 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

Table 4.10: 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

nor the logic in the code changes in these modified files, which can be recognised by having a _cpy in the file name at the end. To run hyperfine in the Linux terminal, write:

```
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 analysers is suppressed by the -i flag in order to have *hyperfine* working properly. To adjust the command, use different version numbers explained in table 4.10 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[3], with each tick increasing the usage by 500 MB. The *pv* command gives a nice visualisation 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, along with the benchmarks for resource management of static analysis tools can be found in the *htop* interactive process viewer.

[3]6.83 GB to be more exact in the binary base.

4.3 Objectives

The objectives of this testing are as follows:

- Test out all five static analysis tools in question - namely: Remix IDE static analysis plug-in, Slither, Oyente, Mythril and SmartCheck and see if they work properly;
- If point one is satisfied, then run all designed tests with vulnerabilities and find out the accuracy rating of each of the static analysis tools;
- Find out if there are any tests whose vulnerabilities cannot be caught by any of the static analysis tools;
- Observe the performance of the analysers 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 chapter in section 4.6 gives an overview of the results and achieves the main goals laid out in section 1.1.

4.4 Results

The analysis of the results presented in section 4.1 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 3.1.1 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 4.3 and compare it to the results obtained in tables 4.4 and 4.5.

As an example, consider figures 4.1, 4.2 and 4.3 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 in figure 4.1 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. Figure 4.2 shows 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 in section 4.1.1 it is impossible to determine any patterns between accuracy and consumption, as Test 3 requires less computing power than, say, Test 10, but both are marked that the analyser 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 up to 28% while the upper bound for test verification time seen in figure 4.3 has also increased significantly compared to other tests.

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

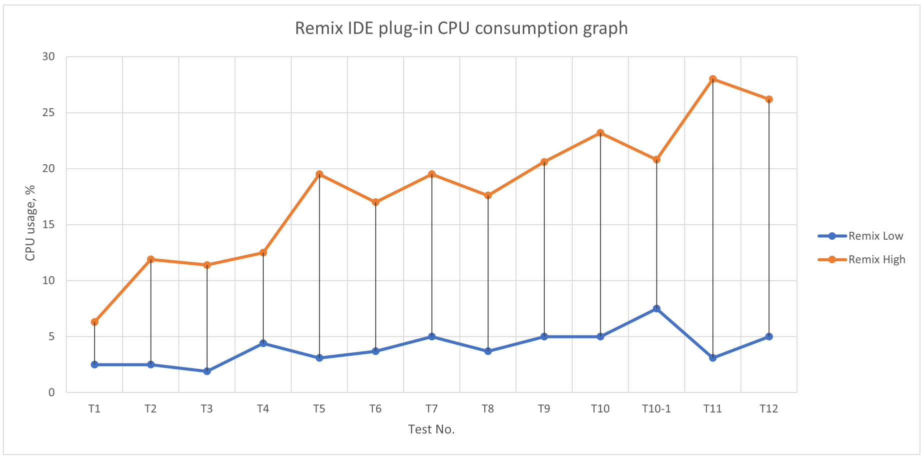


Figure 4.1: Remix IDE static analysis plug-in CPU consumption lower and higher rates for each test

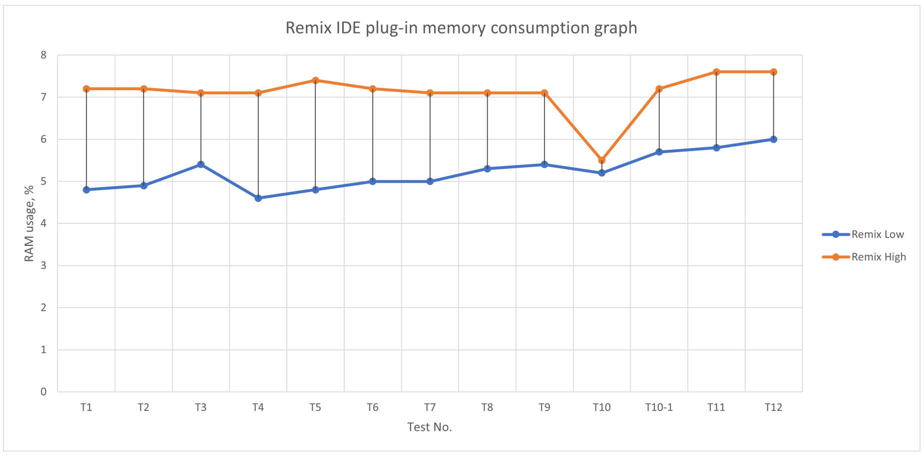


Figure 4.2: Remix IDE static analysis plug-in memory consumption lower and higher rates for each test

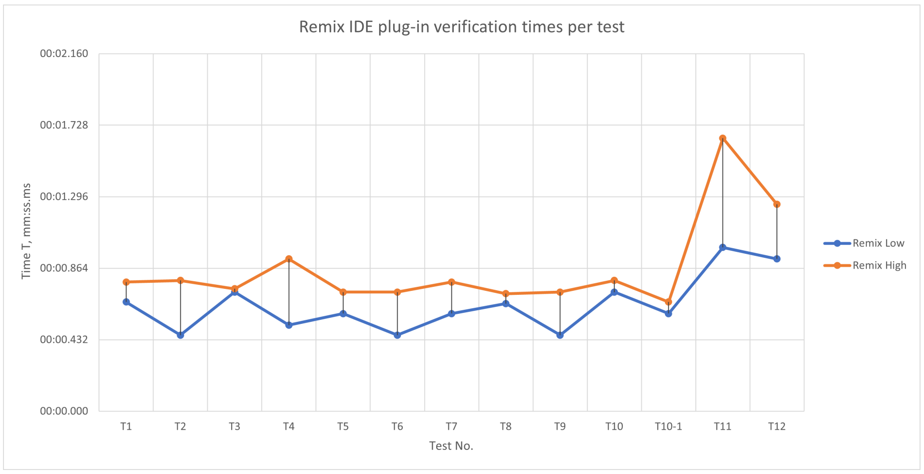


Figure 4.3: Remix IDE static analysis plug-in running times for each test with lower and upper bounds

As a slight jump back to the performance analysis, it has been 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 seen in figure 4.4. Here, the largest upper bound was observed in Test 3 with the CPU usage nearing 30 per cent. The memory usage, seen in figure 4.5, is consistent, similarly to Remix plug-in. Figure 4.6 shows that Slither runs in a consistent fashion 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 have any effect on 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 managed to correctly identify 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 the gap between the upper and lower bounds can hardly be seen in figure 4.3.

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

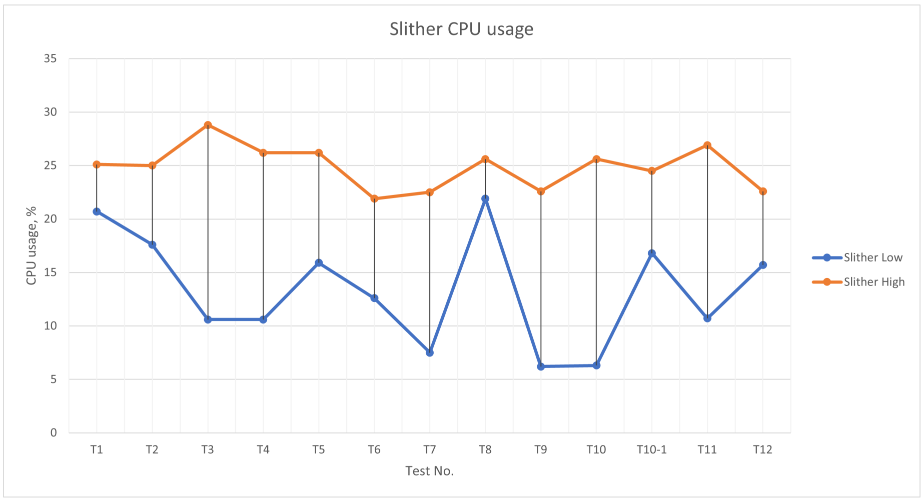


Figure 4.4: Slither CPU consumption lower and upper bounds for each test

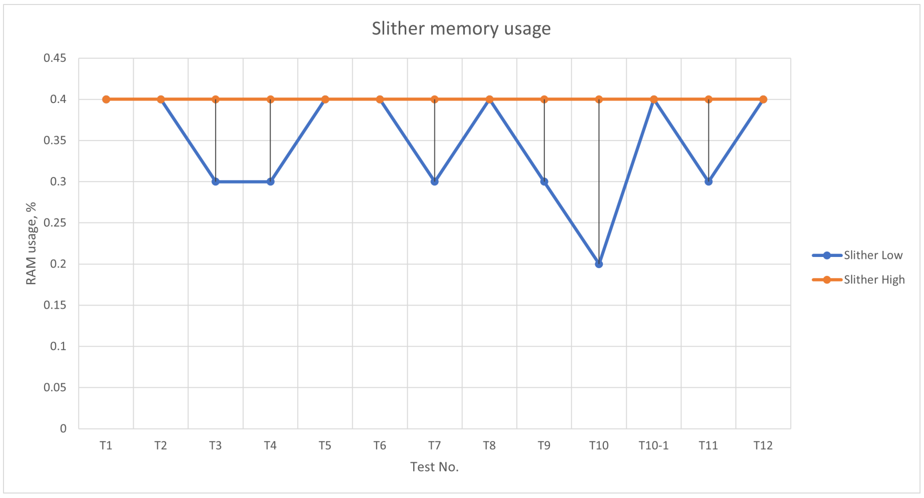


Figure 4.5: Slither memory consumption lower and upper bounds for each test

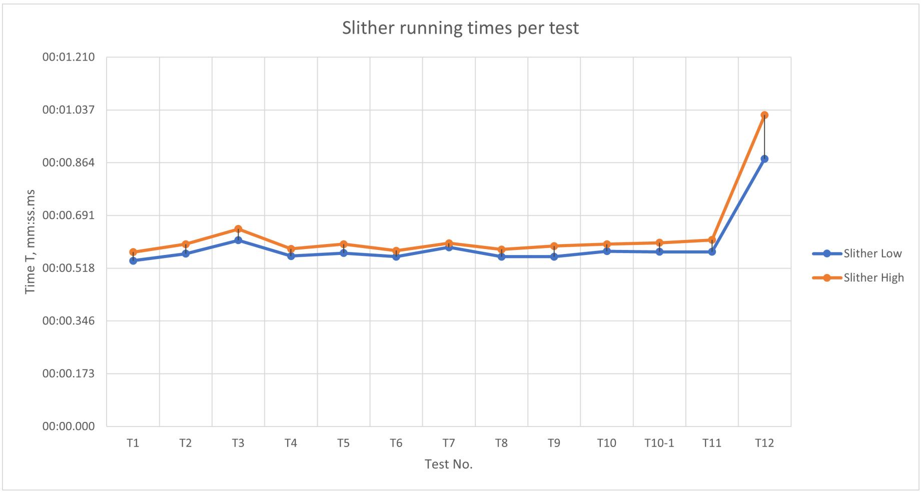


Figure 4.6: Slither running times for each test with lower and upper bounds

Let us look at Oyente. As it can be seen from the CPU consumption graph in figure 4.7, 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 5:30 minutes. This can be seen both in the performance table 4.1 in section 4.1 as well as in figure 4.9, showing the visual “spikes” for Tests 3, 7 and 11. The red horizontal line in the corresponding graph shows the time threshold set for all static analysis tools, which Oyente managed to exceed it once under normal operating conditions. The memory consumption graph, seen in figure 4.9, also indicates the problematic tests as visual “spikes” in the graph. Therefore, in Oyente’s case, one can derive the conclusion that a longer verification process leads to abnormal resource usage by the static analysis tool. If let to run for a prolonged period of time, it is very likely that the kernel would kill the process of the analyser, therefore, making the verification process useless. It also needs to be emphasised that Oyente is an old static analysis tool, with its first incarnation released to the public over 5 years ago [76], where blockchain technology and Solidity programming language were still in their infancy. Therefore, it is only natural that currently there are better optimised, 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 vulnerabilities were identified correctly by Oyente, giving the accuracy percentage of 25%.

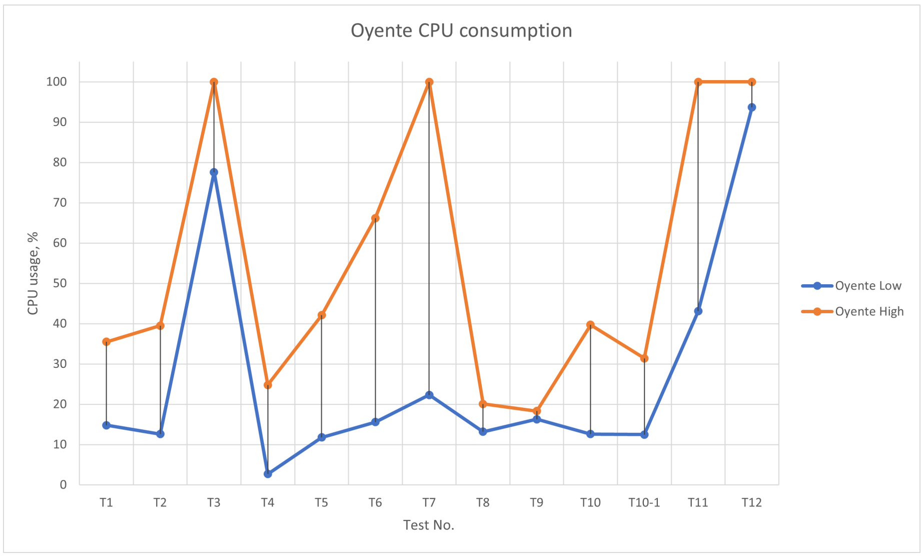


Figure 4.7: Oyente CPU consumption lower and upper bounds for each test

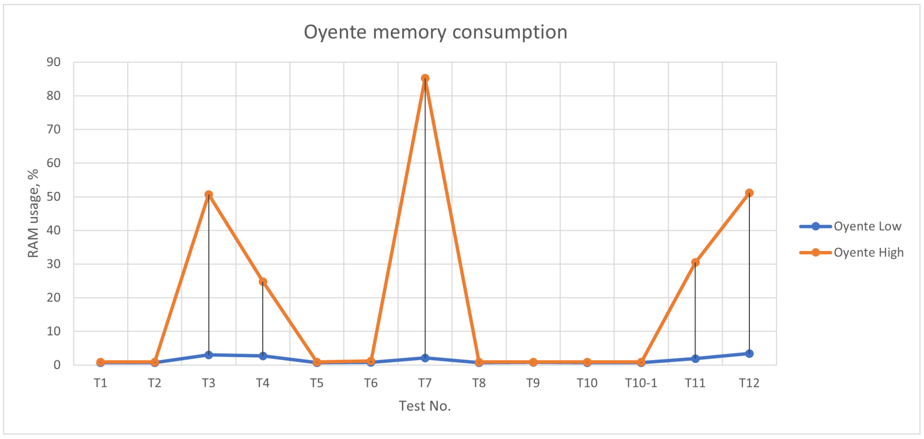


Figure 4.8: Oyente memory consumption lower and upper bounds for each test

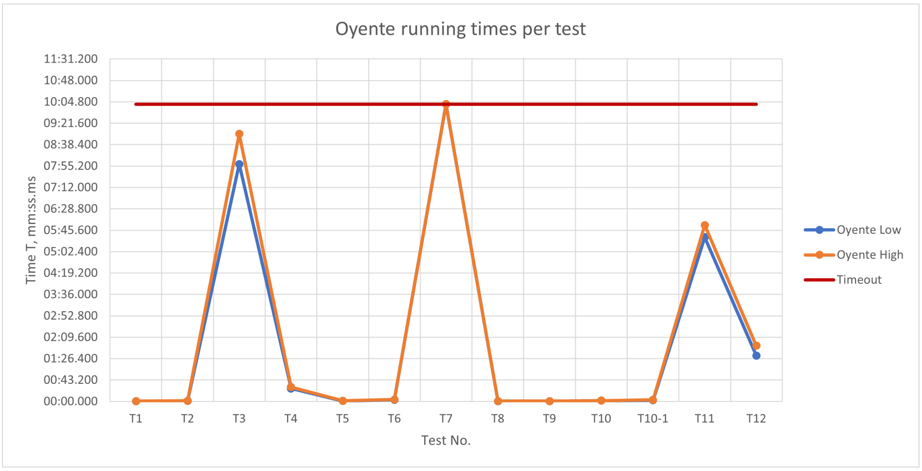


Figure 4.9: Oyente running times for each test with lower and upper bounds

Mythril, on the other hand, 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 with regards to running time. However, as it can be seen from figures 4.10 and 4.11, Mythril takes a greedy approach to computer's resources, especially the computing power of the processor. Almost in all cases, Mythril verifies the tests by using 100% of CPU power, which is not the best outcome if, for example, a user has other CPU-intensive processes running. In that case, the computer would slow down and all processes, including Mythril's static analysis. In terms of memory usage, it is sustainable, although both Slither and Remix IDE plug-in use considerably less memory than Mythril counterpart. It is also important 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 and the running time bounds can be seen in figure 4.12, described in section 3.2.5, takes about two-and-a-half minutes for Mythril to verify the source code, while all other tools take at maximum 2 seconds in order to complete the static analysis. Furthermore, it seems that Mythril does not handle well 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.

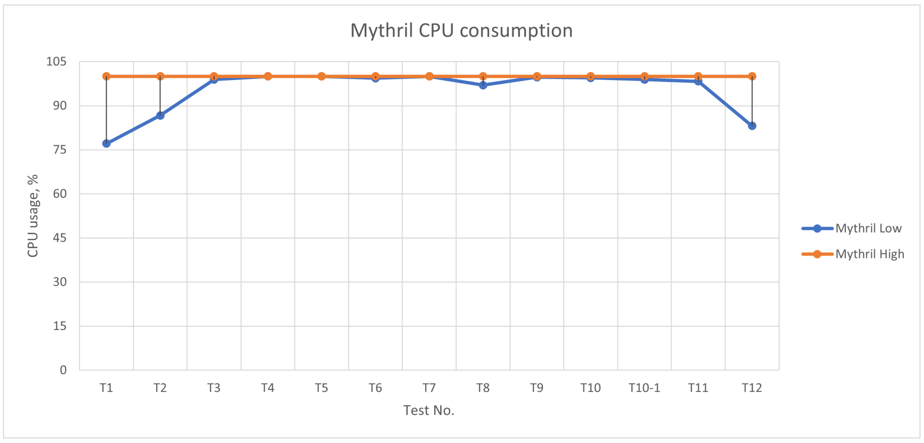


Figure 4.10: Mythril CPU consumption lower and upper bounds for each test

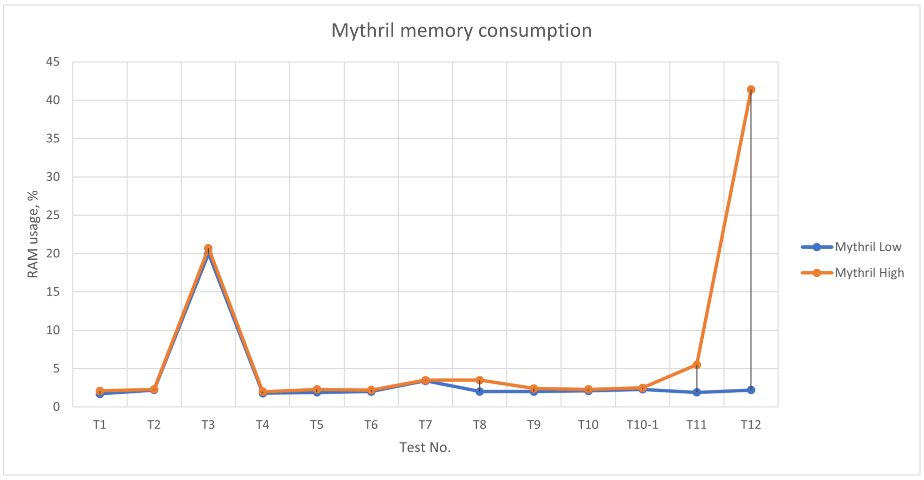


Figure 4.11: Mythril memory consumption lower and upper bounds for each test

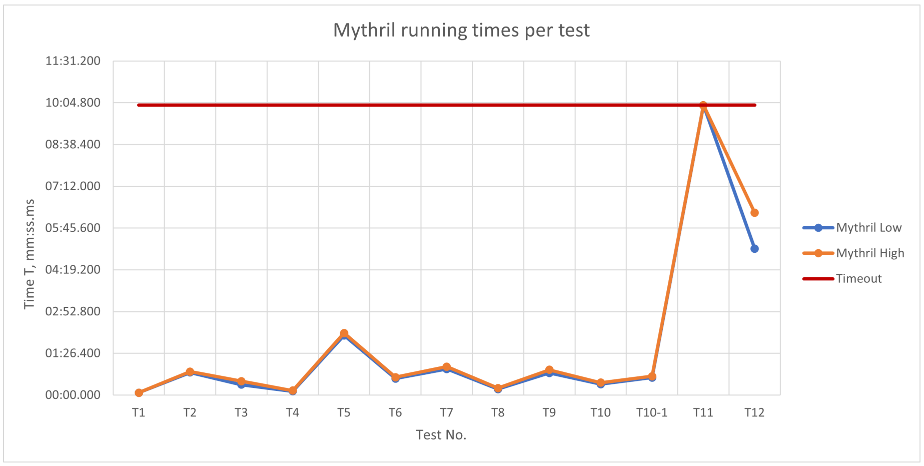


Figure 4.12: Mythril running times for each test with lower and upper bounds

The performance of SmartCheck, in terms of resource management, is satisfactory considering that the analyser 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 - figure 4.14 shows that the static analysis tool does not reach 10% of available RAM for use. On the other hand, figure 4.13 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 figure 4.15, are quite consistent and the graph resembles Slither's graph, figure 4.6. 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 analysers 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.

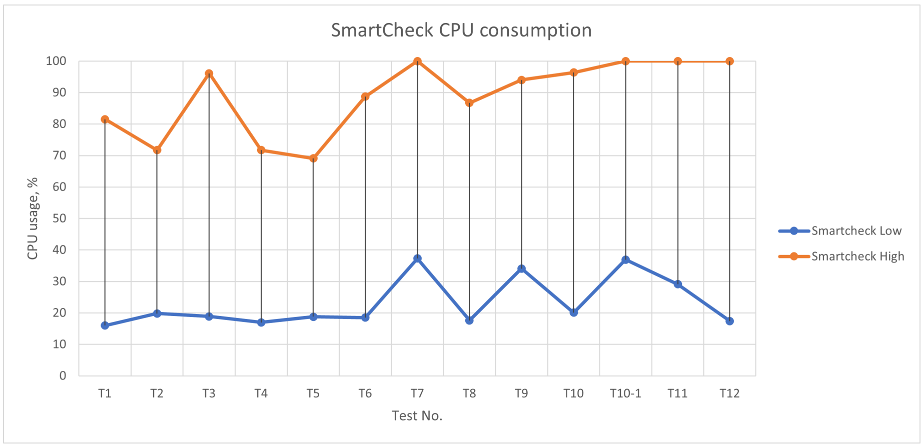


Figure 4.13: SmartCheck CPU consumption lower and upper bounds for each test

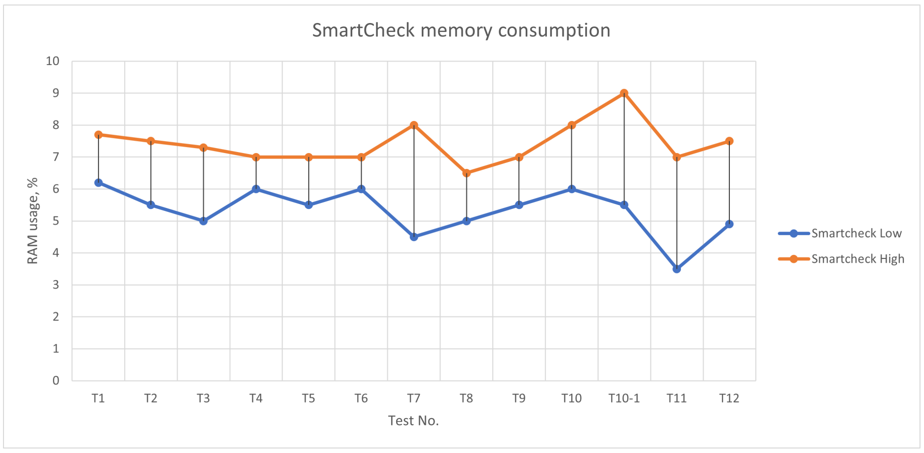


Figure 4.14: SmartCheck memory consumption lower and upper bounds for each test

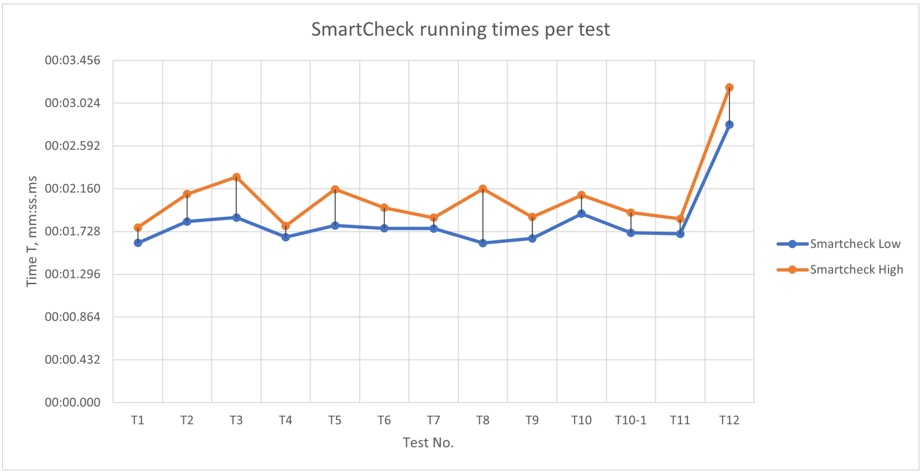


Figure 4.15: SmartCheck running times for each test with lower and upper bounds

4.5 Threat to validity

The experiments done with the static analysis tools for Solidity smart contracts are accurate and done in a consistent fashion in order to prevent any bias in the results gathered. However, the experiments and evaluations are not perfect and there is always room for improvement. Firstly, 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. The SWC registry also contains 36 different vulnerabilities but 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) - it is advisory that the compiler version should be strict for the contract, on the other hand, the smart contract loses flexibility once there is a new Solidity compiler version and the contract needs patching, which might be unnecessary. 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, for example. 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, it is likely that 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 there are 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 analysers, although the rating will be more correct as there would be more cases 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:

- Include more tests with different vulnerabilities, especially if the tests have code that deviates from the standards given by the documentation of static analysers. 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.

- Reduce the number of tests but make them more specialised. For example, if one is particularly interested in re-entrancy vulnerabilities, 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 in fact, there are no vulnerabilities there would improve the testing strategy as well. The specialised tests would allow determining if the tool is truly capable of detecting the security bug in question as well as check for any false positives which 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 project, there is a margin of inconsistent 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. Broadly speaking, it should be thought that having a more powerful machine than the one used in this project 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.

4.6 Summary

As a summary, firstly let us get back to the hypothesis established for the first time in section 3.1.1. As seen in the analysis, compared with the knowledge about the static analysis tools from section 3.1.2, it can be concluded that while Slither is one of the most sophisticated tools tested, with the level of complexity and the capabilities of the analysers the only comparable ones are Oyente and Mythril, Slither does not consume significantly more resources or runs slower on average than other, simpler 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 generally larger tools are inefficient is simply incorrect because as the technology moves forward, better methods of static analysis are created, achieving faster verification rates while retaining low resource consumption rates. If this project was conducted several years ago, one could argue that the technology of the present day has reached its limits, but now, we can see that there are static analysers that are sophisticated, yet powerful, accurate and efficient. This moves to the main aim of the project - what is the best static analysis tool for Solidity smart contracts, which can find the most with cybersecurity-related vulnerabilities? Well, the answer is pretty straightforward, if we are looking at the data presented in this chapter.

Slither is by far gets the title of being the best out of five tools tested, 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 full stress test, Slither managed to not exceed 4 seconds of verification time.

The second place would go to Remix IDE static analysis plug-in, which can be unexpected due to the fact that its implementation is quite simple compared to other tools, but it manages to detect quite a lot of 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 in this project, 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 good, although it falls behind Remix plug-in by 20 percentile points. The resource management can always 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. While being bound to finite amounts of computing power and memory, it is important to have a 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 mediocre 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 really enjoy having.

SmartCheck receives the third place as well, sharing with Mythril due to its low accuracy rating - one of the main factors for a static analyser to be considered as a “good” tool. SmartCheck does not consume a lot of 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 in order 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 not suitable 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 as well, thus distorting the overall performance results. If Oyente would be updated and more efficient methods of traversing CFGs were implemented, from which logical statements are fed into Z3 SMT solver, then Oyente can once again become a powerful static analysis tool for Solidity smart contracts as it was 5 or 6 years ago.

Chapter 5

Conclusion

In this chapter, the concluding statements are given about this project. The chapter is divided into three smaller parts. The first part discusses the achievements of this project, whether it was a success or not. The second part reflects the project itself and if there are any valuable lessons taken from undertaking the project. Lastly, the last section gives some indications about the possibilities of future work, which can be conducted by relying on the work completed for this project.

5.1 Achievements

In summary, the author of this project thinks that the project is a success. It was possible to achieve the main aims, that is, find out publicly available static analysis tools for Solidity smart contracts, write tests w.r.t. cybersecurity properties, run the tests on analysers, observe the behaviour, obtain data and decide which tool is the best currently available for Solidity programmers. Also, a widely perceived belief of heavyweight software being slow or resource-hungry has been proven to be incorrect, and Slither shows that there exist tools that are powerful yet are fully capable of not reserving whole computer resources to their own needs.

One of the main achievements of this project is the successful compilation of Solidity smart contract tests which fit into the cybersecurity properties - it is the main contributory point explained in section 1.3. As there are not many surveys or analyses done, at least to the author's knowledge, where smart contracts with vulnerabilities had risk assessments or comparisons to the CIA triad, this is a significant achievement that may be unique to this project. The results obtained in chapter 4 by running the custom smart contracts developed and explained in detail in chapter 3 give some important insights about the static analysers, their accuracy ratings and their performance under various conditions. From that data, it is possible to 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 priorities of the user wanting to use the analyser - whether the accuracy or the speed is the primary requirement.

5.2 Reflection

As a reflection of this project, the author thinks that there was a steep learning curve in many fields of Computer Science, which allowed the author to learn a lot while working on this project. Firstly, the concepts of blockchain and smart contracts were virtually unknown before undertaking the project, and at the end of the research project not only it is well-known how blockchain works, for example, but also the purpose of smart contracts, their interaction with the blockchain and its users, known and (possibly) unknown vulnerabilities and outcomes when they are exploited by adversaries. Lots of technical insight was gained in the field of cryptocurrency and blockchain technology. Also, a new programming language, Solidity, was learned from scratch and currently the smart contracts in Solidity can be written as fast as more traditional programming languages learned before the project, such as Java, C/C++ or Python.

Secondly, a lot was learned about the static analysis tools themselves, and in-depth knowledge of the internal operations of analysers were learned during the time the project was undertaken. Also, courses on formal methods and static and dynamic analysis of programs were very helpful in understanding how static analysis tools work, irrespective of the programming language. Some concepts, such as bounded model checking (BMC), abstract syntax trees (AST) were unknown before the project but currently, they are well-understood and it can be seen that these concepts are implemented in many code analysers.

Lastly, the concepts of cybersecurity were also a good refresher to the author, and valuable analyses performed on tests were proven to firmly solidify the knowledge about risk assessments, for example. It has also proven that the information theory concepts, such as the CIA triad were crucial in determining which Solidity smart contracts are eligible for cybersecurity analysis and which smart contracts can be discarded, even though lots of the contracts might have been classified as having vulnerabilities in their source code by the static analysis tools. It was important to establish that not all vulnerabilities are important from the cybersecurity perspective, and the severity rating differs from vulnerability to vulnerability.

5.3 Future work

What lies ahead of this project is open-ended. This project lays the groundwork for performing analysis on Solidity smart contracts in more specialised fields - cybersecurity is one of them. There are some surveys that discuss the cybersecurity behind the attacks on smart contract vulnerabilities [27] [28] [31] [57], but they either are quite abstract in terms of the cybersecurity properties provided, or 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. Therefore, there is more in-depth analysis to be done for Solidity smart contracts from the cybersecurity perspective.

As mentioned earlier in this section, cybersecurity is only one of the specialisation fields for Solidity smart contracts. Resource monitoring and consumption analysis delves more into the field of more efficient software writing, which includes SAT/SMT solvers, AST and CFG generation, efficient and fast conversion of code to bytecode or other IR used in the analysers. While general solvers are fast, the bottleneck can occur at the CFG level - large codebases will definitely have huge control-flow graphs, and traversing the whole graph, visiting all nodes can take an unacceptable amount of time for the whole verification process, creating the state space explosion problem. Therefore, the concept of BMC comes into play - by determining the bound until which the graph is traversed, one is sure to have or not to have a vulnerability given some bound.

All in all, the pathways for future work are diverse and future research can rely, for example, on the cybersecurity properties in the smart contracts - especially interesting would be large-scale Decentralised Autonomous Organisations (DAO), as an example given in the official Ethereum webpage [92]. The other pathway can be static analyser accuracy and efficiency - establishing better heuristics or formal methods of detecting more subtle vulnerabilities in contracts or trying to find the bottlenecks in the verification process and trying to remedy them by improving the overall verification flow.

References

[1] *Remix Analyzer - selfdestruct.ts.* Remix Team. URL: `https://github.com/ethereum/remix-project/blob/master/libs/remix-analyzer/src/solidity-analyzer/modules/selfdestruct.ts`. (accessed 02/08/2021).

[2] *Remix Analyzer - checksEffectsInteraction.ts.* Remix Team. URL: `https://github.com/ethereum/remix-project/blob/master/libs/remix-analyzer/src/solidity-analyzer/modules/checksEffectsInteraction.ts`. (accessed 02/08/2021).

[3] Kaden Zipfel. *relayer.sol: Insufficient gas griefing.* ConsenSys Diligence. URL: `https://consensys.github.io/smart-contract-best-practices/known_attacks/#insufficient-gas-griefing`. (accessed 04/08/2021).

[4] Marten Risius and Kai Spohrer. "A blockchain research framework". In: *Business & Information Systems Engineering* 59.6 (2017), pp. 385–409. URL: `https://link.springer.com/content/pdf/10.1007/s12599-017-0506-0.pdf`. (accessed 03/05/2021).

[5] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* bitcoin.org. URL: `https://bitcoin.org/bitcoin.pdf`. (accessed 02/05/2021).

[6] Vitalik Buterin. *Ethereum Whitepaper.* ethereum.org. 9th Feb. 2021. URL: `https://ethereum.org/en/whitepaper/`. Originally published by author in 2013. (accessed 02/05/2021).

[7] Sam Daley. *30 Blockchain Applications and Real-World Use Cases Disrupting the Status Quo.* Built In. 31st Mar. 2021. URL: `https://builtin.com/blockchain/blockchain-applications/`. (accessed 02/05/2021).

[8] *About Solidity.* Solidity Team. URL: `https://soliditylang.org/about/`. (accessed 02/05/2021).

[9] Juri Mattila. *The Blockchain Phenomenon – The Disruptive Potential of Distributed Consensus Architectures.* eng. ETLA Working Papers 38. Helsinki, 2016. URL: `http://hdl.handle.net/10419/201253`. (accessed 14/08/2021).

[10] Toni Caradonna. "Blockchain and society". In: *Informatik Spektrum* 43 (2020), pp. 40–52. DOI: `10.1007/s00287-020-01246-7`. URL: `https://doi.org/10.1007/s00287-020-01246-7`. (accessed 14/08/2021).

[11] Dylan Yaga et al. "Blockchain Technology Overview". In: *CoRR* abs/1906.11078 (2019). arXiv: `1906.11078`. URL: `https://arxiv.org/ftp/arxiv/papers/1906/1906.11078.pdf`. (accessed 02/05/2021).

[12] *Most Common Smart Contract Vulnerabilities.* Hacken OU. 19th Dec. 2018. URL: https://hacken.io/education/most-common-smart-contract-vulnerabilities/. (accessed 02/05/2021).

[13] Manoj Pramesh. *Most common smart contract bugs of 2020.* A Medium Corporation. 30th Nov. 2020. URL: https://medium.com/solidified/most-common-smart-contract-bugs-of-2020-c1edfe9340ac. Ordered article by Solidified Technologies Inc. (accessed 02/05/2021).

[14] Marcell Gogan. *Smart Contract Security: What Are the Weak Spots of Ethereum, EOS, and NEO Networks?* TechNative. 13th Dec. 2018. URL: https://technative.io/smart-contract-security-what-are-the-weak-spots-of-ethereum-eos-and-neo-networks/. (accessed 02/05/2021).

[15] Michael Nofer et al. "Blockchain". In: *Business & Information Systems Engineering* 59.3 (2017), pp. 183–187. URL: https://link.springer.com/content/pdf/10.1007/s12599-017-0467-3.pdf. (accessed 03/05/2021).

[16] Spyridon Samonas and David Coss. "THE CIA STRIKES BACK: REDEFINING CONFIDENTIALITY, INTEGRITY AND AVAILABILITY IN SECURITY." In: *Journal of Information System Security* 10.3 (2014). URL: http://www.proso.com/dl/Samonas.pdf. (accessed 03/05/2021).

[17] Dylan Yaga et al. "Blockchain Technology Overview". In: *CoRR* abs/1906.11078 (2019). arXiv: 1906.11078. URL: https://arxiv.org/ftp/arxiv/papers/1906/1906.11078.pdf. (accessed 02/05/2021).

[18] Tim Swanson. "Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems". In: (2015). URL: https://allquantor.at/blockchainbib/pdf/swanson2015consensus.pdf. (accessed 03/05/2021).

[19] Tim Swanson. "Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems". In: (2015). URL: https://allquantor.at/blockchainbib/pdf/swanson2015consensus.pdf. (accessed 03/05/2021).

[20] Dejan Vujičić, Dijana Jagodić and Siniša Ranđić. "Blockchain technology, bitcoin, and Ethereum: A brief overview". In: *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH).* 2018, pp. 1–6. DOI: 10.1109/INFOTEH.2018.8345547. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=8345547f. (accessed 03/05/2021).

[21] Gavin Wood. "Ethereum: A Secure Decentralised Generalised Transaction Ledger". In: (2021). URL: https://ethereum.github.io/yellowpaper/paper.pdf. (accessed 03/05/2021).

[22] Elvira Albert et al. "GASOL: Gas Analysis and Optimization for Ethereum Smart Contracts". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Armin Biere and David Parker. Cham: Springer International Publishing, 2020, pp. 118–125. ISBN: 978-3-030-45237-7. URL: https://link.springer.com/content/pdf/10.1007%2F978-3-030-45237-7.pdf. (accessed 03/05/2021).

[23] Neville Grech et al. "MadMax: Surviving out-of-Gas Conditions in Ethereum Smart Contracts". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276486. URL: https://doi.org/10.1145/3276486. (accessed 03/05/2021).

[24] Chris Dannen. *Introducing Ethereum and solidity.* Vol. 318. Springer, 2017. URL: http://www.ndl.ethernet.edu.et/bitstream/123456789/26027/1/Chris%20Dannen.pdf. (accessed 03/05/2021).

[25] Andreas Bogner, Mathieu Chanson and Arne Meeuw. "A Decentralised Sharing App Running a Smart Contract on the Ethereum Blockchain". In: *Proceedings of the 6th International Conference on the Internet of Things.* IoT'16. Stuttgart, Germany: Association for Computing Machinery, 2016, 177–178. ISBN: 9781450348140. DOI: 10.1145/2991561.2998465. URL: https://doi.org/10.1145/2991561.2998465. (accessed 03/05/2021).

[26] Shuai Wang et al. "Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends". In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 49.11 (2019), pp. 2266–2277. DOI: 10.1109/TSMC.2019.2895123. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=8643084. (accessed 03/05/2021).

[27] Huashan Chen et al. "A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses". In: *ACM Computing Surveys* 53.3 (June 2020). ISSN: 0360-0300. DOI: 10.1145/3391195. URL: https://doi.org/10.1145/3391195. (accessed 03/05/2021).

[28] Nicola Atzei, Massimo Bartoletti and Tiziana Cimoli. "A Survey of Attacks on Ethereum Smart Contracts (SoK)". In: *Principles of Security and Trust.* Ed. by Matteo Maffei and Mark Ryan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186. ISBN: 978-3-662-54455-6. URL: https://link.springer.com/content/pdf/10.1007%2F978-3-662-54455-6.pdf. (accessed 03/05/2021).

[29] Mikhail R. Gadelha et al. "ESBMC 5.0: An Industrial-Strength C Model Checker". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* ASE 2018. Montpellier, France: Association for Computing Machinery, 2018, 888–891. ISBN: 9781450359375. DOI: 10.1145/3238147.3240481. URL: https://doi.org/10.1145/3238147.3240481. (accessed 03/05/2021).

[30] Lucas Cordeiro et al. "JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode". In: *Computer Aided Verification.* Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 183–190. ISBN: 978-3-319-96145-3. URL: https://link.springer.com/content/pdf/10.1007%2F978-3-319-96145-3.pdf. (accessed 03/05/2021).

[31] Purathani Praitheeshan et al. "Security Analysis Methods on Ethereum Smart Contract Vulnerabilities:A Survey". In: *CoRR* abs/1908.08605 (2019). arXiv: 1908.08605. URL: http://arxiv.org/abs/1908.08605. (accessed 03/05/2021).

[32] Lexi Brent et al. "Vandal: A Scalable Security Analysis Framework for Smart Contracts". In: *CoRR* abs/1809.03981 (2018). arXiv: 1809.03981. URL: https://arxiv.org/pdf/1809.03981.pdf. (accessed 03/05/2021).

[33] Josselin Feist, Gustavo Grieco and Alex Groce. “Slither: A Static Analysis Framework for Smart Contracts”. In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 2019, pp. 8–15. DOI: `10.1109/WETSEB.2019.00008`. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=8823898f`. (accessed 03/05/2021).

[34] Sukrit Kalra et al. “ZEUS: Analyzing Safety of Smart Contracts.” In: *Ndss*. 2018, pp. 1–12. URL: `http://pages.cpsc.ucalgary.ca/~joel.reardon/blockchain/readings/ndss2018\_09-1\_Kalra\_paper.pdf`. (accessed 03/05/2021).

[35] Chao Liu et al. “Understanding Out of Gas Exceptions on Ethereum”. In: *Blockchain and Trustworthy Systems*. Ed. by Zibin Zheng et al. Singapore: Springer Singapore, 2020, pp. 505–519. ISBN: 978-981-15-2777-7. URL: `https://link.springer.com/content/pdf/10.1007%2F978-981-15-2777-7.pdf`. (accessed 03/05/2021).

[36] Lexi Brent et al. “Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, 454–469. ISBN: 9781450376136. DOI: `10.1145/3385412.3385990`. URL: `https://doi.org/10.1145/3385412.3385990`. (accessed 03/05/2021).

[37] Anton Permenev et al. “VerX: Safety Verification of Smart Contracts”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1661–1677. DOI: `10.1109/SP40000.2020.00024`. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=9152791`. (accessed 03/05/2021).

[38] R. B. Whitner and O. Balci. “Guidelines for Selecting and Using Simulation Model Verification Techniques”. In: *Proceedings of the 21st Conference on Winter Simulation*. WSC ’89. Washington, D.C., USA: Association for Computing Machinery, 1989, 559–568. ISBN: 0911801588. DOI: `10.1145/76738.76811`. URL: `https://doi.org/10.1145/76738.76811`. (accessed 27/07/2021).

[39] Peng Li and Baojiang Cui. “A comparative study on software vulnerability static analysis techniques and tools”. In: *2010 IEEE International Conference on Information Theory and Information Security*. 2010, pp. 521–524. DOI: `10.1109/ICITIS.2010.5689543`. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=5689543`. (accessed 27/07/2021).

[40] Anjana Gosain and Ganga Sharma. “Static Analysis: A Survey of Techniques and Tools”. In: *Intelligent Computing and Applications*. Ed. by Durbadaland Kar Mandal et al. New Delhi: Springer India, 2015, pp. 581–591. ISBN: 978-81-322-2268-2. URL: `https://link.springer.com/content/pdf/10.1007%2F978-81-322-2268-2_59.pdf`. (accessed 27/07/2021).

[41] Ashish Aggarwal and Pankaj Jalote. “Integrating Static and Dynamic Analysis for Detecting Vulnerabilities”. In: *30th Annual International Computer Software and Applications Conference (COMPSAC’06)*. Vol. 1. 2006, pp. 343–350. DOI: `10.1109/COMPSAC.2006.55`. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=4020095`. (accessed 27/07/2021).

[42] David Devecsery et al. "Optimistic Hybrid Analysis: Accelerating Dynamic Analysis through Predicated Static Analysis". In: *SIGPLAN Not.* 53.2 (2018), 348–362. ISSN: 0362-1340. DOI: `10.1145/3296957.3177153`. URL: `https://doi.org/10.1145/3296957.3177153`. (accessed 27/07/2021).

[43] Carlos E. Silva and José C. Campos. "Combining Static and Dynamic Analysis for the Reverse Engineering of Web Applications". In: EICS '13. London, United Kingdom: Association for Computing Machinery, 2013, 107–112. ISBN: 9781450321389. DOI: `10.1145/2494603.2480324`. URL: `https://doi.org/10.1145/2494603.2480324`. (accessed 27/07/2021).

[44] Dan Craigen, Nadia Diakun-Thibault and Randy Purse. "Defining Cybersecurity". In: *Technology Innovation Management Review* 4 (2014), pp. 13–21. ISSN: 1927-0321. DOI: `http://doi.org/10.22215/timreview/835`. URL: `http://timreview.ca/article/835`. (accessed 28/07/2021).

[45] Rossouw von Solms and Johan van Niekerk. "From information security to cyber security". In: *Computers & Security* 38 (2013). Cybercrime in the Digital Economy, pp. 97–102. ISSN: 0167-4048. DOI: `https://doi.org/10.1016/j.cose.2013.04.004`. URL: `https://www.sciencedirect.com/science/article/pii/S0167404813000801`. (accessed 28/07/2021).

[46] *ISO/IEC 27000:2018 Information technology — Security techniques — Information security management systems — Overview and vocabulary.* International Organization for Standardization. Feb. 2018. URL: `https://www.iso.org/standard/73906.html`. (accessed 28/07/2021).

[47] *ISO/IEC 27000:2018(en) Information technology — Security techniques — Information security management systems — Overview and vocabulary.* International Organization for Standardization. Feb. 2018. URL: `https://www.iso.org/obp/ui/#iso:std:iso-iec:27000:ed-5:v1:enl`. (accessed 28/07/2021).

[48] *Definition of cybersecurity.* International Telecommunication Union. URL: `https://www.itu.int/en/ITU-T/studygroups/2013-2016/17/Pages/cybersecurity.aspx`. (accessed 28/07/2021).

[49] *Security Tip (ST04-001) - What is Cybersecurity?* Cybersecurity & Infrastructure Security Agency. 14th Nov. 2019. URL: `https://us-cert.cisa.gov/ncas/tips/ST04-001`. (accessed 28/07/2021).

[50] *Cybersecurity - Glossary.* National Institute of Standards and Technology, Information Technology Laboratory, Computer Security Resource Center. URL: `https://csrc.nist.gov/glossary/term/cybersecurity`. (accessed 28/07/2021).

[51] *Common Weakness Enumeration - A Community-Developed List of Software & Hardware Weakness Types.* The MITRE Corporation. URL: `https://cwe.mitre.org/index.html`. (accessed 28/07/2021).

[52] *CVE.* The MITRE Corporation. URL: `https://cve.mitre.org/index.html`. (accessed 28/07/2021).

[53] *Common Vulnerability Scoring System v3.1: User Guide.* FIRST.Org, Inc. URL: `https://www.first.org/cvss/v3.1/user-guide`. (accessed 28/07/2021).

[54] *Decentralized Application Security Project (or DASP) Top 10 of 2018.* NCC Group. URL: https://dasp.co/. (accessed 28/07/2021).

[55] *SWC Registry.* SmartContractSecurity. URL: https://swcregistry.io. (accessed 28/07/2021).

[56] *How this Coding Standard is Organized - Risk Assessment.* Carnegie Mellon University, Software Engineering Institute. URL: https://wiki.sei.cmu.edu/confluence/display/c/How+this+Coding+Standard+is+Organized#HowthisCodingStandardisOrganized-RiskAssessment. (accessed 28/07/2021).

[57] Zeli Wang et al. "Ethereum smart contract security research: survey and future research opportunities". In: 15 (2020). DOI: 10.1007/s11704-020-9284-9. URL: https://link.springer.com/content/pdf/10.1007/s11704-020-9284-9.pdf. (accessed 08/08/2021).

[58] Xiaoqi Li et al. "A survey on the security of blockchain systems". In: *Future Generation Computer Systems* 107 (2020), pp. 841–853. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.08.020. URL: https://www.sciencedirect.com/science/article/pii/S0167739X17318332. (accessed 10/08/2021).

[59] *The Cyber Kill Chain®.* Lockheed Martin Corporation. URL: https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html. (accessed 10/08/2021).

[60] *MITRE ATT&CK®.* The MITRE Corporation. URL: https://attack.mitre.org/. (accessed 10/08/2021).

[61] Dipankar Dasgupta, John M. Shrein and Kishor Datta Gupta. "A survey of blockchain from security perspective". In: *Journal of Banking and Financial Technology* 3 (2019), pp. 1–17. DOI: https://doi.org/10.1007/s42786-018-00002-6. URL: https://www.sciencedirect.com/science/article/pii/S0167739X17318332. (accessed 10/08/2021).

[62] Davide Zoni, Andrea Galimberti and William Fornaciari. "An FPU design template to optimize the accuracy-efficiency-area trade-off". In: *Sustainable Computing: Informatics and Systems* 29 (2021), p. 100450. ISSN: 2210-5379. DOI: https://doi.org/10.1016/j.suscom.2020.100450. URL: https://www.sciencedirect.com/science/article/pii/S2210537920301761. (accessed 02/08/2021).

[63] Marzieh Vaeztourshizi, Mehdi Kamal and Massoud Pedram. "EGAN: A Framework for Exploring the Accuracy vs. Energy Efficiency Trade-off in Hardware Implementation of Error Resilient Applications". In: *2020 21st International Symposium on Quality Electronic Design (ISQED).* 2020, pp. 438–443. DOI: 10.1109/ISQED48828.2020.9137041. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=9137041. (accessed 02/08/2021).

[64] Ivica Nikolic. *Maian.* URL: https://github.com/ivicanikolicsg/MAIAN. (accessed 14/08/2021).

[65] Eric Rafaloff. *SolAnalyzer.* URL: https://github.com/EricR/solanalyzer. (accessed 14/08/2021).

[66] Chao Peng. *SIF (Solidity Instrumentation Framework) - issue No. 6: terminate called after throwing an instance of 'nlohmann::detail::out_of_range'.* URL: `https://github.com/chao-peng/SIF/issues/6`. (accessed 14/08/2021).

[67] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. URL: `https://link.springer.com/content/pdf/10.1007%2F978-3-540-78800-3_24.pdf`. (accessed 02/08/2021).

[68] *Remix - Ethereum IDE.* remix-project.org. 9th Feb. 2021. URL: `https://remix.ethereum.org/`. (accessed 02/05/2021).

[69] *Welcome to Remix's documentation!* Remix Project. URL: `https://remix-ide.readthedocs.io/en/latest/`. (accessed 02/08/2021).

[70] *Remix Analyzer.* Remix Team. URL: `https://github.com/ethereum/remix-project/tree/master/libs/remix-analyzer#how-to-use`. (accessed 02/08/2021).

[71] *Solidity Static Analysis.* Remix Team. URL: `https://remix-ide.readthedocs.io/en/latest/static_analysis.html`. (accessed 02/08/2021).

[72] Robert Brummayer and Armin Biere. "Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Stefan Kowalewski and Anna Philippou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 174–177. ISBN: 978-3-642-00768-2. URL: `https://link.springer.com/content/pdf/10.1007%2F978-3-642-00768-2_16.pdf`. (accessed 03/08/2021).

[73] Roberto Bruttomesso et al. "The MathSAT 4 SMT Solver". In: *Computer Aided Verification.* Ed. by Aarti Gupta and Sharad Malik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 299–303. ISBN: 978-3-540-70545-1. URL: `https://link.springer.com/content/pdf/10.1007%2F978-3-540-70545-1_28.pdf`. (accessed 03/08/2021).

[74] Clark Barrett et al. "CVC4". In: *Computer Aided Verification.* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–177. ISBN: 978-3-642-22110-1. URL: `https://link.springer.com/content/pdf/10.1007%2F978-3-642-22110-1_14.pdf`. (accessed 03/08/2021).

[75] *Detector Documentation.* Trail Of Bits. URL: `https://github.com/crytic/slither/wiki/Detector-Documentation`. (accessed 03/08/2021).

[76] Loi Luu et al. "Making Smart Contracts Smarter". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, 254–269. ISBN: 9781450341394. DOI: `10.1145/2976749.2978309`. URL: `https://doi.org/10.1145/2976749.2978309`. (accessed 03/08/2021).

[77] *Oyente - An Analysis Tool for Smart Contracts.* (current version maintained by Xiao Liang Yu). URL: `https://github.com/enzymefinance/oyente`. (accessed 03/08/2021).

[78] Bernhard Mueller. "Smashing Ethereum Smart Contracts for Fun and Real Profit". In: (2018). URL: `https://raw.githubusercontent.com/b-mueller/smashing-smart-contracts/master/smashing-smart-contracts-1of1.pdf`. (accessed 03/08/2021).

[79] Bernhard Mueller. *LASER-ethereum.* URL: `https://github.com/b-mueller/laser-ethereum`. (accessed 03/08/2021).

[80] *Modules.* ConsenSys AG. URL: `https://mythril-classic.readthedocs.io/en/master/module-list.html`. (accessed 03/08/2021).

[81] Sergei Tikhomirov et al. "SmartCheck: Static Analysis of Ethereum Smart Contracts". In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain.* WETSEB '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 9–16. ISBN: 9781450357265. DOI: `10.1145/3194113.3194115`. URL: `https://doi.org/10.1145/3194113.3194115`. (accessed 03/08/2021).

[82] Terence Parr. *ANTLR (ANother Tool for Language Recognition).* URL: `https://www.antlr.org/`. (accessed 03/08/2021).

[83] *htop - an interactive process viewer.* htop Team. URL: `https://htop.dev/`. (accessed 03/08/2021).

[84] *hyperfine - A command-line benchmarking tool.* The hyperfine developers. URL: `https://github.com/sharkdp/hyperfine`. (accessed 03/08/2021).

[85] *SWC 116 - Block values as a proxy for time.* SmartContractSecurity. URL: `https://swcregistry.io/docs/SWC-116`. (accessed 04/08/2021).

[86] *A Postmortem on the Parity Multi-Sig Library Self-Destruct.* Parity Technologies. 15th Nov. 2017. URL: `https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/`. (accessed 04/08/2021).

[87] *Security Considerations - tx.origin.* Solidity Team. URL: `https://docs.soliditylang.org/en/latest/security-considerations.html?highlight=tx.origin`. (accessed 05/08/2021).

[88] *Contracts - Constructors.* Solidity Team. URL: `https://docs.soliditylang.org/en/latest/contracts.html?highlight=constructor#constructors`. (accessed 05/08/2021).

[89] Andreas Moser, Christopher Kruegel and Engin Kirda. "Limits of Static Analysis for Malware Detection". In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007).* 2007, pp. 421–430. DOI: `10.1109/ACSAC.2007.21`. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=4413008`. (accessed 05/08/2021).

[90] *Detector Documentation - Dead-code.* Trail Of Bits. URL: `https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code`. (accessed 08/08/2021).

[91] *solc-select: A tool to quickly switch between Solidity compiler versions.* Trail Of Bits. URL: `https://github.com/crytic/solc-select`. (accessed 12/08/2021).

[92] *Ethereum Community - Decentralized Autonomous Organizations (DAOs).* ethereum.org. URL: `https://ethereum.org/en/community/#decentralized-autonomous-organizations-daos`. (accessed 08/08/2021).