

MASTER'S THESIS 2023

# Study of bug detection tools on Ethereum smart contracts

Edward Axlund, Frans Sjöström

Elektroteknik  
Datateknik

ISSN 1650-2884

LU-CS-EX: 2023-79

DEPARTMENT OF COMPUTER SCIENCE

LTH | LUND UNIVERSITY





EXAMENSARBETE  
Datavetenskap

LU-CS-EX: 2023-79

**Study of bug detection tools on Ethereum  
smart contracts**

Studie av säkerhetsverktyg för Ethereums  
smarta kontrakt

Edward Axlund, Frans Sjöström



---

# Study of bug detection tools on Ethereum smart contracts

---

Edward Axlund  
edaxlund@gmail.com

Frans Sjöström  
fr5536sj-s@student.lu.se

May 25, 2023

Master's thesis work carried out at  
the Department of Computer Science, Lund University.

Supervisor: Niklas Fors, `niklas.fors@cs.lth.se`

Examiner: Cristoph Reichenbach, `christoph.reichenbach@cs.lth.se`



## Abstract

Blockchains were introduced through the creation of Bitcoin in 2008. Since then, many projects have been starting in the area, including Ethereum, which introduced the smart contract, a computer program that can be deployed to a blockchain with public functions for interactability. These contracts aim to automate some current financial services like lending, swapping, or investing. With the introduction of these smart contracts, there have been a lot of different kinds of hacks exploiting vulnerable code for the attacker's financial gain. The hacks have created much focus on software security within smart contracts and the development of automated software auditing tools. The field is very young and has yet to reach the maturity we might be used to in Java or C.

This report contains a comparative study between the symbolic execution tools Maian and Mythril and the static analysis tool Slither. Comparing the different tools also shows some of the strengths and weaknesses of static analysis and symbolic execution. This project builds upon an earlier study that evaluated smart contract security tools on the source code of smart contracts. Our study details true/false positives and true/false negatives about both methods and in which circumstances one might be better to use.

We find that Mythril is best at finding multiple-step vulnerabilities performing the best on our benchmark dataset. Slither has the best completeness, finding more of the true positives that Mythril sometimes misses. Maian lacks a lot in analyzing newer contracts where it fails; we think this is the simple reason it is no longer maintained.

**Keywords:** Blockchain, smart contracts, program analysis, static analysis, symbolic execution, solidity, self-destruct





# Acknowledgements

---

We want to thank Mikkel Jensen our supervisor from Chainalysis that have supported us throughout the project and added an industry perspective to the project.

We like to thank Niklas Fors for taking the time to supervise the project and making sure we are creating a contribution.

We would like to thank Sundas Munir for her insight in static analysis tools and her curiosity about what is pushing the boundaries for smart contract analysis tools.

We would like to thank Martina Rossini for creating the data-set we have used for the smart contract source code.



# Contents

---

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Contribution . . . . .	8
1.2	Goals of thesis . . . . .	8
1.3	Outline . . . . .	9
1.4	Remark . . . . .	9
<b>2</b>	<b>Background</b>	<b>11</b>
2.1	Blockchain and Smart Contracts . . . . .	11
2.1.1	Blockchain Technology . . . . .	11
2.1.2	Ethereum . . . . .	12
2.1.3	EVM - Ethereum Virtual Machine . . . . .	12
2.1.4	Solidity . . . . .	13
2.1.5	Smart contracts . . . . .	13
2.1.6	Program analysis on smart contracts . . . . .	14
2.1.7	Self-destruct . . . . .	14
2.2	Program analysis methods . . . . .	15
2.2.1	Symbolic execution . . . . .	15
2.2.2	SMT solvers . . . . .	16
2.2.3	Static analysis . . . . .	16
2.3	Tools used . . . . .	17
2.3.1	Selection of tools . . . . .	17
2.3.2	Maian . . . . .	18
2.3.3	Mythril . . . . .	19
2.3.4	Slither . . . . .	19
<b>3</b>	<b>Vulnerabilities on the Ethereum blockchain</b>	<b>21</b>
3.1	Datasets . . . . .	21
3.2	Setup of experiments . . . . .	22
3.2.1	Tools . . . . .	23
3.3	Activity on the Ethereum blockchin . . . . .	23

3.4	SWC-id . . . . .	23
3.4.1	Integer Arithmetic bugs (SWC: 101) . . . . .	24
3.4.2	Reachable/Unprotected self-destruct (SWC: 106) . . . . .	24
3.4.3	Delegatecall to user-supplied address (SWC: 112) . . . . .	24
3.4.4	Write to an arbitrary storage location (SWC: 124) . . . . .	25
3.4.5	Reentrancy exploit (SWC: 107) . . . . .	27
3.4.6	Dependence on predictable environment variable (SWC: 116, 120) . . . . .	28
3.4.7	Jump to an arbitrary instruction (SWC: 127) . . . . .	29
3.4.8	Hardcoded guards (No SWC) . . . . .	29
3.5	Results and Discussion . . . . .	30
3.6	Closing remarks and zooming in on self-destruct . . . . .	36
<b>4</b>	<b>Self-destruct within the Ethereum blockchain</b>	<b>37</b>
4.1	Definition of reachable self-destruct . . . . .	37
4.2	Access Control . . . . .	38
4.2.1	Flawed Access Control . . . . .	39
4.2.2	Intended reachable self-destruct . . . . .	41
4.3	Slither's static analysis of reachable self-destruct . . . . .	41
4.4	Detecting reachable self-destruct with Mythril's symbolic execution . . . . .	42
4.5	Method . . . . .	42
4.6	Results and Discussion . . . . .	43
4.6.1	Manual review . . . . .	43
4.6.2	Closing remarks on zooming in on reachable selfdestruct . . . . .	48
4.7	Benchmark dataset . . . . .	48
4.8	Limits of tools and possible improvements . . . . .	51
4.8.1	Can you detect intended reachable self-destruct . . . . .	51
4.8.2	Slither . . . . .	51
4.8.3	Mythril . . . . .	55
<b>5</b>	<b>Related work</b>	<b>57</b>
5.1	Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts . . . . .	57
5.2	A critical comparison on six static analysis tools: Detection, agreement, and precision . . . . .	58
5.3	Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities . . . . .	59
<b>6</b>	<b>Conclusion</b>	<b>61</b>
6.1	Future work . . . . .	62
6.1.1	Surveying more . . . . .	62
6.1.2	Implementing improvement of tools . . . . .	62
6.1.3	Detecting vulnerabilities using neural networks . . . . .	62
	<b>References</b>	<b>65</b>

# Chapter 1

## Introduction

---

Blockchain has been a word introduced to the world through the recent rise of Cryptocurrencies. Cryptocurrencies are digitally distributed currencies meant to be transferred without a central authority controlling them. Since the release of the first cryptocurrency, Bitcoin [27], in 2008, a lot of development has occurred and now in 2023 there exist many cryptocurrencies. The presence of the strong economic incentive of being early in a new type of world economy has led to a fast and hasty development of products where security is not always prioritized.

Blockchains such as Ethereum are used to develop decentralized applications dAPPS, with some common applications being financial services, NFTs, and collectibles. These applications use cryptocurrency as the monetary asset needed to interact with them. The blockchain works as a decentralized ledger where nodes interact with each other and verify transactions in order to keep the blockchain safe without having to involve a central authority. In a blockchain, transactions are recorded in blocks, which are cryptographically linked to each other, forming a chain of blocks that cannot be altered without the consensus of the network.

To facilitate decentralized financial services, some blockchains have enabled programmers to write code that gets stored within the blockchain. This code can then execute when transactions on the blockchain trigger them. This is done without human involvement when conditions in the code are met. These programs are often referred to as smart contracts. Ethereum is the blockchain currently holding the most monetary value that utilizes this type of technology. It was launched in 2015 and was the first blockchain to make smart contracts available. Smart contracts are meant to enable users to interact directly with each other without involvement of a central organization and still being assured that the transactions are safe. Ethereum has a virtual machine where bytecode gets executed when transactions have been validated. For this virtual machine many languages have been created, with the most commonly used one being solidity which is an object-oriented language similar to JavaScript in syntax[17].

Smart contracts being in their infancy combined with the large assets stored within them

---

have created incentive for malicious users to look for vulnerabilities that can net them monetary gains. A lot of money has been stolen through finding vulnerabilities in contract code. A well-known incident is the parity wallet-hack, where attackers took advantage of faulty access control and managed to make use of the self-destruct function of a smart contract and gain access to approximately 30 million USD[20]. Another instance where a large sum of money was stolen was the Wormhole hack, where attackers managed to obtain cryptocurrency through faulty version control of the developers [20]. This illustrates that even code from well-known actors within the crypto eco-system can be faulty.

Because of the high stakes of uploading code that has control over millions of dollars onto a blockchain, many have tried to develop tools to find faulty code. For Ethereum numerous tools can find different types of vulnerabilities. The most common tools are either using static analysis or symbolic execution for finding these bugs. So far many tools have made big claims to be able to find faulty code, but they still are far behind many tools for traditional languages such as Java and C.

## 1.1 Contribution

The main goal of this project is to present the current state of bug-detecting tools on the Ethereum blockchain. We run the tools to estimate the number of bugs on the Ethereum blockchain over time. We also present a more in-depth review of the tools by analyzing how accurate and precise the three tools are at detecting reachable self-destruct and their weaknesses.

The results from this research will increase users' awareness of smart contracts, what tools they can trust, and to what extent. It will also show some of the flaws of the different program analysis methods and recommend in what cases to use each. The report creates recommendations for improving Slither, Maian, and Mythril without going into specific implementations.

The symbolic execution tools used in the research are Mythril and Maian, and the static analysis tool used is Slither. Slither and Mythril could be counted as state-of-the-art tools where they are maintained open-source projects with backers such as Consensys and Trail of Bits, two prominent players in the Ethereum space.

## 1.2 Goals of thesis

- Compare the tools based on verified and unverified datasets.
- Create benchmark dataset for reachable self-destruct.
- Evaluate results of static analysis versus symbolic execution.
- Improvement for Maian, Slither, and Mythril.
- Present the trend of general vulnerabilities in smart contracts deployed on the Ethereum blockchain over time.

## 1.3 Outline

The thesis is structured as follows. Chapter 2 introduces some of the basics that are needed to understand how Solidity and the EVM work. This is where the master thesis is split into two different parts. First, Chapter 3, where we present our insights into the Ethereum blockchain ecosystem, focuses on how current bugs are and how the ecosystem has evolved over time. Chapter 4 covers the specific reachable self-destruct vulnerability and tries to understand how good Maian, Slither, and Mythril are when trying to detect reachable self-destruct. In chapter 5 conclusions are drawn from insights from the earlier two chapters and reasons for future work.

## 1.4 Remark

This thesis tries not to take a stance if cryptocurrency and DeFi are good or bad. It highlights new technology and shows how it has changed over time, it also demonstrates the importance of writing safe code and how classifying code as safe can have its inaccuracies.





# Chapter 2

## Background

---

This chapter is divided into three parts. First, a description of blockchain technology and how it is used in Ethereum together with the EVM to enable the use of smart contracts. Secondly, the two major program analysis methods that are used in this project are produced Static analysis and symbolic execution. Finally, the tools used in the report are introduced and a brief description of their inner workings is given.

## 2.1 Blockchain and Smart Contracts

### 2.1.1 Blockchain Technology

A blockchain is a distributed ledger that is tamper resistant and tamper evident, making it ideal for transaction security. Blockchains enable users to record transactions on a shared distributed ledger without the involvement of a central authority. Transactions on the blockchain cannot be changed and are stored on the blockchain facilitating privacy while still upholding security against tampering and maintaining openness without revealing identity of users. Blockchain combined cryptography, distributive networks and database technology to create electronic currency, i.e crypto currency. Bitcoin launched in 2008 was the first crypto currency and is still the largest currency with a market cap of 457 billion USD. [17] [35]

Blockchains function through protocols, much like the protocols used to allow internet users to interact with each other seamlessly. The protocols decide how transactions get validated within the network and how new currency is produced. Heavy calculations needed to run the blockchain are divided in the network to maintain security and decentralisation. When a transaction gets validated it is referred to as *reaching consensus*, a variety of solutions among different blockchains permit this to be achieved. To reach consensus all nodes involved in the next block need to agree on what transactions occurred. [35]

## 2.1.2 Ethereum

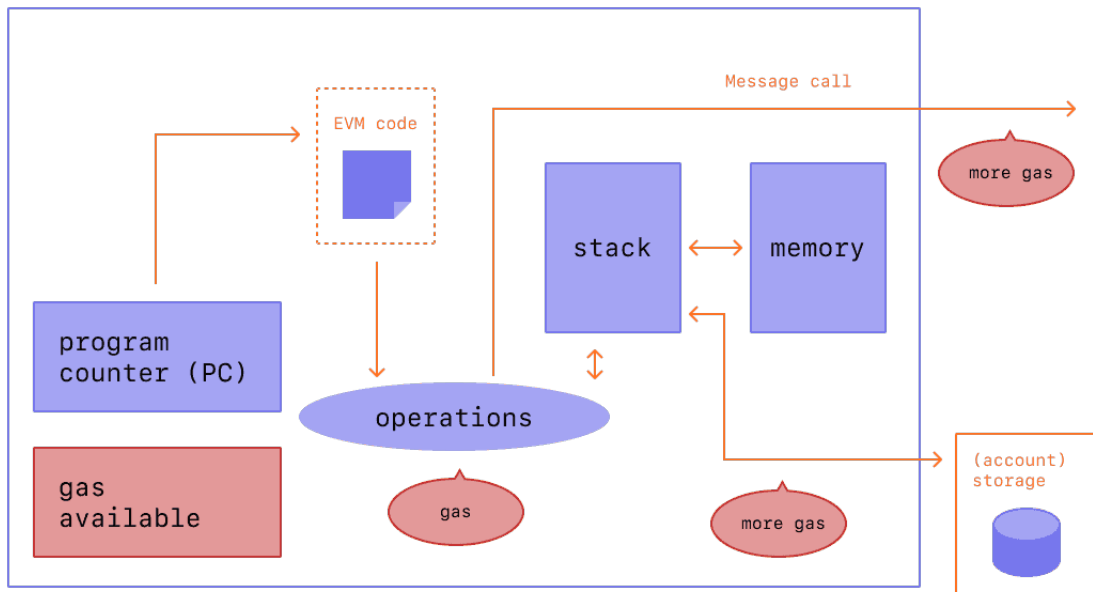
In this report, Ethereum, launched in 2015, will be in focus because it is the blockchain with the most monetary value that facilitates the use of smart contracts, an automatic way of executing transactions when specific conditions are met, on the blockchain using code. A simple example can be a lottery, where a random seed has determined the winning number. If a user buys the winning ticket, the winnings will be transferred to the winning user without any involvement of a central authority. Ethereum is a distributed ledger, like other blockchains but can also be viewed as a state machine, where data such as account balances or evidence of ownership from the real world can be stored. These features are made possible by the Ethereum blockchain's Turing-complete virtual machine, The EVM, short for Ethereum virtual machine, which allows for code to be programmed on the blockchain. All new blocks added to the Ethereum blockchain can be seen as state changes performed through a state transition function. There are rules for how states can change like an account balance can not go up without another going down. That is how the integrity of the transactions of the Ethereum blockchain is managed. [17] [35]

## 2.1.3 EVM - Ethereum Virtual Machine

The Ethereum Virtual machine is the machine that makes code runnable on the Ethereum blockchain. It is quasi-turing complete, with the quasi part referring to that the computations performed on the blockchain are fixed to the cost of running it. The cost of running something is called *gas* and is measured in Ether, and limits the amount of calculations being done. Heavy calculations are therefore usually performed outside of solidity to reduce cost. Programs on the Ethereum blockchain are in this way incentivized to be as small and efficient as possible to keep the cost of running them down.[35]

The EVM is a simple stack based architecture with a word-address byte array memory model. The word size of the EVM is 256. It was chosen to facilitate the keccak256 hash-scheme. The machine has an independent non volatile storage model where the system state is kept. When a smart contract is deployed, its variables get stored in the blockchain. Each node in the blockchain has a local copy of the system state where it keeps the hash of the current system state. When a transaction is conducted, each node validates the new state through the hash copy of the state included in the new block. This ensures that computations are consistent throughout all nodes of the chain. [35]

To avoid overloading the network with heavy computation, a fee for transactions and computation is imposed, referred to as gas. With infinite gas available, the EVM would be considered turing-complete. A gas fee is imposed on creating contracts, utilizing and accessing account storage, executing operations on the EVM and making message calls. The usage of gas has its own sets of vulnerabilities that are outside the scope of this thesis.[35]



**Figure 2.1:** Overview of the EVM architecture, showing how gas is used throughout the architecture. Taken from Ethereum Foundation at Ethereum website.

### 2.1.4 Solidity

Solidity is the leading programming language, used to write code that can be compiled into the EVM. It is a high-level language that is designed to be easy to read and write, with syntax similar to that of JavaScript. It is statically typed, meaning data types are defined at compile-time and cannot be changed during runtime. It also supports inheritance, libraries, and user-defined types. Solidity code needs to be compiled in order to be executed on the Ethereum blockchain. The compilation process involves converting Solidity source code into bytecode, which can be executed by the EVM.[3]

In this report, the code examples are provided in Solidity since it is the most commonly used language for contracts on the EVM. Since the syntax is very similar to many mainstream programming languages, we do not go into detail about how to use the language, but some common solidity specific functionality is explained.[17]

### 2.1.5 Smart contracts

Ever since the launch of the Ethereum blockchain in 2015[5], smart contracts have become more and more relevant because of the substantial monetary assets stored in them and the technical applications it is being used for. Smart contracts are simple programs written to a blockchain to execute automatically when certain conditions are met. They provide the tools necessary to create an automated financial ecosystem with distributed ledgers and are used in cryptocurrency facilitating blockchains like Ethereum.

Since smart contracts can hold large sums of cryptocurrency, with some of the wealthiest contracts containing hundreds of millions of US dollars, security is essential for smart

contracts. Several times since the launch of the Ethereum blockchain has smart contracts on the blockchain been taken advantage of by malicious users, often for large sums of money. One example is the Parity Wallet Hack, where hackers managed to steal Ether to a value of 30 million US dollars because of faulty code from the developers [20].

Smart contracts can today be divided into two groups verified and not verified. The division is not a fundamental feature built into the Ethereum blockchain but rather a reaction. Because people run bytecode when interacting with smart contracts on addresses on the Ethereum blockchain, it is hard to know what the bytecode you are interacting with does. Here independent organizations started to provide a service of verifying smart contracts by letting developers post their source code onto a website for everyone to see the logic. Available source code is convenient because it is hard to decompile bytecode and make it readable for humans. When posting the source code to a site, it can easily be checked by compiling it and comparing that bytecode to the code on the blockchain. There is no need for anyone to verify their bytecode, but most serious projects do to be transparent with their users.

### 2.1.6 Program analysis on smart contracts

There are two kinds of data to use for program analysis of smart contracts: bytecode and source code. As mentioned above, according to our numbers, even if all the data on the EVM is public, only 12 percent of it is verified. That makes tools that use bytecode more relevant because of their wider useability. There exists different program analysis methods that can be used for smart contracts; see figure 2.3. This report only covers symbolic execution and static analysis. As seen in figure 2.3, static analysis is both a method and a type of program analysis. In this report, static analysis is a concept summarizing all the different static analysis methods like dataflow analysis, control sensitive analysis, and call sensitive analysis, among others, that can be done during compile time.

There exists tools that can detect a variety of different bugs. Many of them detect various bugs and vary in approach to catching them. No tool has yet become the industry standard, and the surge of new active addresses [32] might be why so many new tools have emerged up in recent years [25].

### 2.1.7 Self-destruct

self-destructs (`selfdestruct(address a)`) functionality is to remove the smart contract and its functionality from the EVM and send all the funds of the contract to a specific address named as the input argument to the function. Any funds sent to the contract from that point on will be accessible by no one, with the exception of someone creating a new smart contract to that specific address using the relatively new `CREATE2` opcode that was introduced in February of 2019[22].

The purpose of including a self-destruct instruction varies between developers but in a survey that was made of smart contract developers, the top 3 motivations were security, cleanup, and quick withdrawal of funds. The survey also included Gas refunds, which is a coded incentive for users of the EVM to remove stored variables and smart contracts [14] [12].

Self-destruct can be included because of concerns that there might be bugs in the smart contract so that the owner/deployer wants to quickly be able to remove any funds from the

contract and mitigate the damage if a security loophole is discovered. Because the code is immutable when on chain it is hard to make updates, hence self-destruct becomes a way of updating your contract is by deleting the old one and deploying a new one.

The gas refund incentive gives the caller of the self-destruct instruction a refund of up to half of a new transaction cost. This gives the incentive to remove old contracts that the developer does not use anymore and receive a discount on a transaction. This functionality has led to the development of GasCoin that are just created to exploit the gas refund feature to create storage for gas on the EVM for when gas is cheap, to later use it when gas is expensive. By this exploit or functionality, suggestions are made to change the gas refund system [14][12].

The most important reason, according to the previously mentioned survey of solidity developers, to not include self-destruct in their smart contracts was the security risk. Illicit actors can recover the owner's private key, or permissions are not handled correctly.

To note is that from Solidity 0.8.18 and up, including self-destruct instruction in your code will generate a deprecation warning because it might be removed or replaced in a later fork [3]. This does not make this report irrelevant because the principles apply to every smart contract deployed to date and the analysis method for finding reachable code can be applied to other critical instructions as well.

This report looks into the angle of how well-automated tools are able to detect these flawed permissions or if they are able to find a publicly reachable self-destruct instruction [12].

## 2.2 Program analysis methods

### 2.2.1 Symbolic execution

Symbolic execution is a method to analyze programs in a way that makes it possible to see what parts of the code can be reachable and if that part can be exploited. It has been used since the 1970s and has been used to find vulnerabilities in code such as null pointers, division by 0, and access control. [9] Symbolic execution generalizes testing with comprehending variables in a program as abstract values. Using the abstract values, you can then analyze if code blocks are reachable by any user or not. To do this SMD solvers are used to see if there is a viable value for the variables to reach the proper code. Here is a visual example taken by Aldrich and Claire [7]. The program in Figure 2.2 has multiple paths that can be taken. Symbolic execution explores all these different paths to get a picture of what code is reachable with what variable values. Here the different variables  $a$ ,  $b$ ,  $c$  are exchanged for  $\alpha$ ,  $\beta$ , and  $\gamma$ . This specific path through this small code block is then created.  $\alpha$  is set to false, and  $\beta$  is at least 5 by the symbolic execution to discover that path. In line 9 the symbolic execution ends with a logical expression  $\neg\alpha \wedge (\beta \geq 5) \wedge (0 + 0 + 0 \neq 3)$ . This expression can then be evaluated as false and therefore fail the assertion. Later on, it will try with  $\alpha = \text{true}$  and other variations to discover the different control flow branches of the program.

```
1 int x=0, y=0, z=0;
2 if(a) {
3     x = -2;
4 }
5 if (b < 5) {
6     if (!a && c) { y = 1; }
7     z = 2;
8 }
9 assert(x + y + z != 3);
```

**Figure 2.2:** Example Code

Exploring all the different paths and evaluating reachability to critical code by applying SMT solvers to the logical expressions to see the necessary input for reaching the section. See *SMT-solvers* subsection for further understanding.

One of the big upsides of symbolic execution is that if it labels something as dangerous, it means that it has found a specific input that can reach a critical area with certainty. The problem is that there can be a lot of paths in a program, if not an infinite amount; infinite loops can cause this problem. Therefore Symbolic execution needs to have a max depth to make sure it comes to a result and also limit its running time. The restriction of the number of paths becomes weakness, since the symbolic execution might miss critical paths that can lead to a vulnerability.

Smart contracts introduce a new set of vulnerabilities apart from traditional programs in Java or C, which can be found using symbolic execution such as TOD-vulnerabilities (Transaction Order Dependencies) and new types of reachable self-destruct [31]. In this report, two symbolic execution tools, Maian and Mythril, are used to identify reachable self-destruct vulnerabilities in smart contracts.

## 2.2.2 SMT solvers

Satisfiability Modulo Theories or SMT solvers are mathematical solvers that can calculate first-order logic. Where SAT solvers use pure boolean expressions ( $true \wedge (false \vee true)$ ), SMT solvers can use more mathematical representations called first-order logic, see example ( $x \neq y \wedge z \wedge x > 4$ ). In short, you create mathematical constraints representing the program's data flow using a control flow graph; then, the SMT solver calculates if there is a possible solution where all the constraints are met [15].

## 2.2.3 Static analysis

Static analysis is a name for methods to analyze run-time code without actually executing the code. It can be done on both source code and run-time bytecode. Some common static analysis techniques are Syntactic Pattern Matching, Data Flow Analysis, Abstract Interpretation, and Constraint-Based Analysis. With the help of these techniques, errors like memory leaks, null pointer references, invalid arithmetic operations, and non-terminating programs can be found. It can also be useful for type checking. Static analysis often utilizes abstraction to transform complicated programs into simpler representations that share key properties with

the original program. An intermediate representation of the source code is often used for this purpose [19].

A static analysis can not be both fully precise and fully accurate while still being guaranteed to terminate. Therefore developers of static analysis tools have to prioritize one of the two. A tool can also be sound or unsound. A sound analysis guarantees that the analysis holds for all executions of the program, whereas an unsound analysis does not. This comes back to the dilemma if you want the tool to be fully accurate since a sound analysis is not guaranteed to terminate if it is. One example is when checking for integer overflow. If all numbers were to be checked, the analysis is guaranteed to be accurate and precise but not guaranteed to terminate. Developers then have to decide whether finding all true positives while including some false positives is more suitable or if knowing that all the flagged code is actually vulnerable. In the case of smart contracts having a conservative analysis, i.e finding all bugs including false positives, should be preferred since the stakes of faulty code can be so high [19] [24].

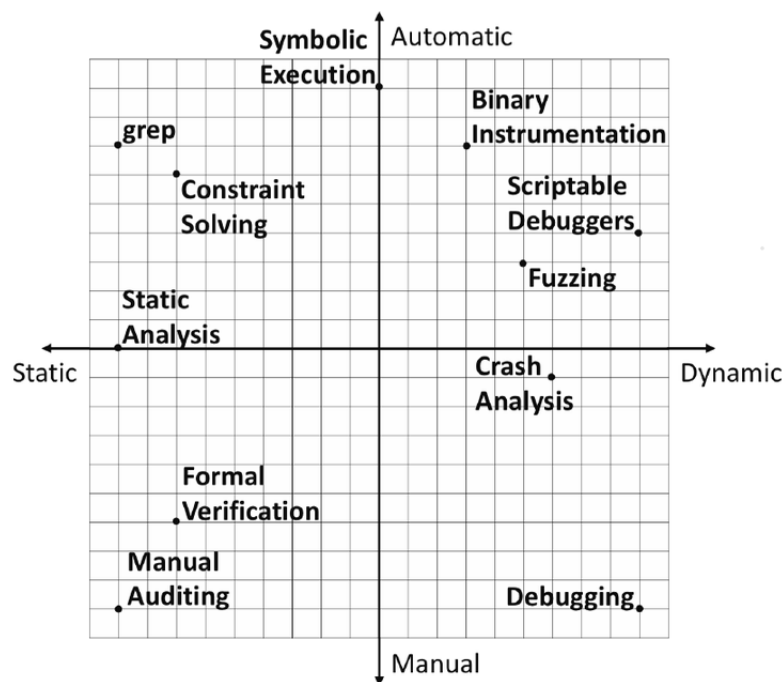


Figure 2.3: Types of program analysis

## 2.3 Tools used

### 2.3.1 Selection of tools

The tools were selected based on four criteria:

- Including at least one tool from the two major techniques of program analysis tools for smart contracts, static analysis and symbolic execution.

- Tools that are considered state of the art in detecting access control vulnerabilities
- The tool being capable of finding reachable selfdestruct
- Being able to run the tool locally and the tool being fast enough to analyze enough contracts within our limited time-span. This means that tools have to be open-source or available in another way.

Slither and Mythril were chosen because the T.Durieux et Al. suggested this combination to cover as many vulnerabilities as possible and they could both detect reachable self-destruct. Maian was chosen because it was developed to find composite self-destruct vulnerabilities, such as the parity wallet hack.

Other tools considered was Securify, Ethainter and teEther. Securify and teEther did not meet our speed limitations and Ethainter was not available to us, even though we tried to get hold of it.

### 2.3.2 Maian

Maian is one of the two symbolic execution tools used in this report. It was released in 2018 by (I.Nikolic et al). In their paper “Finding The Greedy, Prodigal, and Suicidal Contracts at Scale” they describe how the tool was created and what results they got from it when testing. The tool can identify three types of vulnerabilities in smart contracts, greedy, prodigal, and suicidal. This solely focuses on how it finds suicidal contracts and compares the results with the ones we obtain on our own.[28]

In the paper, the authors ran almost a million contracts, finding 34 200 unique vulnerabilities. 3,759 of these contracts were sampled for concrete validation and manual analysis, which resulted in the conclusion that the tool had an 89% true positive rate. The authors claim an average execution time of 10 seconds per contract, which is inconsistent with the results from the Smartbugs data set, where Maian had an average execution time of 5 minutes and 16 seconds. This can also be due to overhead in the Smartbugs framework.[16][28]

Maian's aim was to be able to identify trace vulnerabilities which are vulnerabilities that can happen through multiple invocations of a contract. Since the state of the blockchain can change when you invoke a contract, contracts can appear safe if you check one invocation. The parity wallet hack is an example of this type of behavior. To achieve this, the authors first classified the three types of vulnerabilities they aimed to investigate. The authors also defined two violations that can make a contract vulnerable to trace vulnerabilities, safety violations, which make contracts vulnerable through a trace that has been on the blockchain that causes the contract to violate safety properties and liveness violations which are determined by which actions cannot be taken from a certain state on the chain [28].

To achieve this Maian first uses symbolic analysis using an EVM-like language inspired by ETHERLITE [1]. The goal of the analysis is to find a state where one of the safety or liveness properties are met. Candidate values are found to then try to reach a vulnerable state through multiple invocations of the contract. If such a path is found, Maian uses the Z3 to find the concrete values to reach this state [15]. To verify that the contract is vulnerable Maian creates a private blockchain that then verifies that this vulnerable state can be reached in practice [28].



### 2.3.3 Mythril

Mythril is one of the two symbolic execution tools for detecting reachable self-destruct in this paper. It can find an array of different vulnerabilities in most EVM-compatible blockchains, such as Ethereum. It does this by running symbolic execution on EVM bytecode, hence making it compatible with all blockchains that use it. Mythril, similarly to Maian, goes through all the possible states of the contract within the call depth. Mythril was developed by Consensys in 2019, which was founded by one of the creators of Ethereum.

It uses the Z3 solver just like Maian and has also developed its own symbolic virtual machine, LASER, for generating the symbolic values needed. The analysis generates symbolic values for the Ethereum bytecode and then regenerates the EVM opcode to then be able to execute the opcode when looking for vulnerabilities. [32]

### 2.3.4 Slither

Slither is a framework for static analysis on Ethereum source code; it is the static analysis tool we use in this paper. It is designed to provide a granular analysis of smart contract code and to be flexible enough to build upon. It was released in 2019 by Trail of Bits. It is right now used for automated vulnerability detection, automated optimization detection, code understanding, and assisted code review. Slither uses the abstract syntax tree generated by the EVM compiler to build an intermediate representation called SiltherIR that it uses to build its control flow graph. The methods of built-in code analysis that Slither then uses are read/write, protected functions, and data dependency analysis. The full architecture of Slither can be seen in. [18]

The open-source version of Slither comes with more than 20 bug detectors, including a check for unprotected self-destruct. Slither is written in Python and comes with tools far more advanced than finding an access-control vulnerability such as unprotected self-destruct. Slither was chosen over similar programs such as Securify since it has been proven to be far more accurate and much quicker [18] [16].



## Chapter 3

# Vulnerabilities on the Ethereum blockchain

---

This chapter presents an overview of the current state of the Ethereum blockchain and provides a time perspective on bugs present on the Ethereum blockchain. Since the blockchain is still in its infancy, it is still under rapid change, both regarding the technical aspects and in what types of contracts are deployed. We present how common bugs have been over time and how active the contracts flagged for vulnerabilities are by measuring the number of transactions these contracts have been a part of. This chapter aims to give insight into how relevant different bugs are today and how their presence in the Ethereum ecosystem has changed since the launch of Ethereum. Analyzing the vulnerabilities present helped us confirm the relevance of reachable self-destruct, described more in-depth in Chapter 4.

There are two datasets used in this chapter, unique bytecodes deployed on the Ethereum blockchain and a subset of this, verified contracts where the source code is also available. The same code can be deployed multiple times onto the blockchain, but the first time is counted as a data point.

### 3.1 Datasets

Smart contracts on the Ethereum blockchain can be divided into two categories, verified and unverified. Verified contracts mean that the developers have shared their source code to provide transparency and make users more comfortable in interacting with the contract. Unverified contracts are contracts where the source code is not available. Currently on the blockchain 13% of the contracts are verified. This is done by uploading your code onto [etherscan.io](https://etherscan.io), which is a website that stores all verified contracts.

	Amount
Unverified bytecodes	850037
Verified sourcecodes	115852 (13%)
Sampled bytecodes	85004 (10%)
Verified in sample	11711 (1.4%)

**Table 3.1:** Data overview

The verified dataset was taken from [huggingface.co](https://huggingface.co) and was a way to get a head start on the project. It was created by Martina Rossini and describes the state of the verified smart contracts in July 2022 [29]. The results of Rossini’s work can be found here The unverified bytecodes are publicly available and can be downloaded by anyone for example through Infura, a company that provides an API to get information from the Ethereum blockchain.

The original 850 000 unverified smart contracts would take too long to analyze with the chosen tools. To handle this problem a selection of 10% (85 thousand) contracts were sampled for reducing the run-time on our remote Linux server to around a week. To both have verified and unverified smart contracts was important for us to be able to understand the different presence of bugs in the two categories. Also, it was important to have the source code of contracts with bugs, since then we could understand how the tools reasoned when classifying a contract as containing a vulnerability.

## 3.2 Setup of experiments

Chainanalysis provided us with a remote Fedora Linux server on which we could run the different tools on. It has an AMD Ryzen 9 3900 12-core processor that the processes were running on.

To run each tool we wrote scripts using Python and its data analysis library Pandas. All the datasets were stored in .parquet files which is a column-oriented way of storing data in compressed form. When using the data the file was first decompressed into a pandas data frame and then all the necessary data was collected. For the symbolic execution tools, a timeout was decided for each run for two purposes, to limit runtime over the big datasets and to collect data on how many of the contracts took significantly longer to run than an average contract. The relevant contracts, i.e the contracts that either timed out or flagged were saved back into .parquet file format for easy handling.

All graphs in this chapter and the following was made with **Matplotlib**, which is a python library for creating graphs.

### 3.2.1 Tools

Tools	Type of analysis	Verified sourcecodes	Sampled bytecodes
Maian	Symbolic Exectuion	X	X
Myhtril	Symbolic Execution	X	X
Slither	Static dataflow analysis	X	

**Table 3.2:** Tools and which datasets they were ran on

## 3.3 Activity on the Ethereum blockchin

The activity on the Ethereum blockchain can be analyzed with a lot of different parameters. We decided to look at the following parameters:

- Overview of number of verified and unverified smart contracts *figure 4.1*
- Number of unique smart contracts deployed *figure 3.12*
- Number of unique verified smart contracts deployed *figure 3.12*
- Number of transactions to unique bytecodes *figure 3.10*
- Number of bytecodes flagged by bug detection tools *figure 3.3 figure 3.13*
- What bugs are most present over time *figure 3.14*

These graphs are used for comparing different tools, what findings the tools have, and seeing overall trends for vulnerabilities present in verified and unverified contracts.

## 3.4 SWC-id

SWC stands for Smart contract weakness classification and is an open-source registry for bugs in Solidity code maintained by the team behind MythX.[33] MythX is a tool that uses Myhtril and therefore the SWC-id and Myhtril are closely connected. SWC-id is loosely aligned with Common Weakness Enumeration which is a more well known community developed list of vulnerabilities in hardware and software systems. These kinds of ids are created to serve as a common ground when talking about different kinds of security vulnerabilities and in this report SWC-id are used. [4] [33]

In this section, bugs classified with high or medium severity is briefly explained and simplified examples are provided. The examples are written by us but inspired by the examples provided by the SWC website. [33]

### 3.4.1 Integer Arithmetic bugs (SWC: 101)

These bugs utilize that an integer has limited storage space. This can cause integer overflow or overflow where a variable gets rewritten to an undesired value since the value is outside the possible value of the storage space. For a short example see 3.1.

```
1 contract IntegerOverflow {  
2     uint public count = 1;  
3  
4     function run(uint256 input) public {  
5         count -= input;  
6     }  
7 }
```

**Figure 3.1:** A simple example of how integer underflow could happen. Subtracting count with 2 until the value goes out of memory range would cause an integer underflow.

### 3.4.2 Reachable/Unprotected self-destruct (SWC: 106)

Unprotected self-destruct is when anyone can destroy a contract at any time. This results in the funds will be transferred from the contract to the address specified in the function, and the contract's functionality and memory will not exist anymore. In the case of 3.2, the funds contained in the contract will be sent to the caller of the function.

These can vary in complexity everything from very simple ones, see code in figure 3.2 to more advanced examples shown in Chapter 4.

```
1 contract SimpleSuicide {  
2     function sudicideAnyone() {  
3         selfdestruct(msg.sender);  
4     }  
5 }
```

**Figure 3.2:** A simple example of an unprotected `selfdestruct` call, `msg.sender` is the developer that called the function of the contract

### 3.4.3 Delegatecall to user-supplied address (SWC: 112)

With `DELEGATECALL`, the called contract's code is executed in the context of the calling contract, and any storage updates are made to the calling contract's storage. The execution of external code allows for shared libraries and code reuse, as well as reducing the gas cost of the operation. Executing logic from another contract can compromise the security of a smart contract. With a delegate call to a user-supplied address, an attacker can introduce new logic to a contract that makes it vulnerable to attacks[3]. As seen in 3.3, the owner can be changed outside the context of the original contract.

```
1 contract Exploitable {
2     address owner;
3
4     constructor() public {
5         owner = msg.sender;
6     }
7
8     function forward(address callee, bytes _data) public {
9         require(callee.delegatecall(_data));
10    }
11 }
12
13 contract ExternalOwnerChange {
14     function changeOwner(address new_owner){
15         owner = new_owner;
16     }
17 }
```

**Figure 3.3:** Example of how a user is able to supply the address to use in the `delegatecall(address a, data d)` function. By calling another contracts logic, the owner of the contract can change if a function with owner changing functionality is called.

### 3.4.4 Write to an arbitrary storage location (SWC: 124)

This bug can occur when an attacker is able to rewrite sensitive variables such as the owner of a contract by overwriting it using another variable. Since each variable has limited storage, a write outside of this storage space can change another variable and hence make all sensitive variables of a contract vulnerable. This can manipulate call functions to bypass requirements and checks [33]. An example of this can be seen in figure 3.4

```
1 contract Wallet {  
2     uint[] private bonusCodes;  
3     address private owner;  
4  
5     constructor() public {  
6         bonusCodes = new uint[] (0);  
7         owner = msg.sender;  
8     }  
9  
10    function UpdateBonusCodeAt(uint idx, uint c) public {  
11        require(idx < bonusCodes.length);  
12        bonusCodes[idx] = c;  
13    }  
14  
15    function Destroy() public {  
16        require(msg.sender == owner);  
17        selfdestruct(msg.sender);  
18    }  
19 }
```

**Figure 3.4:** Example of an arbitrary write where the bonuscodes array bonuscodes is used to rewrite the owners adress. This gives acces to the Destroy() function.



### 3.4.5 Reentrancy exploit (SWC: 107)

```

1 contract SimpleDAO {
2     mapping (address => uint) public credit;
3
4     function donate(address to) payable public{
5         credit[to] += msg.value;
6     }
7
8     function withdraw(uint amount) public{
9         if (credit[msg.sender]>= amount) {
10
11             //Here is where the external call is done before the
12             //state change
13             require(msg.sender.call.value(amount)());
14
15             #This part will not be reached until the contract is empty
16             credit[msg.sender]-=amount;
17         }
18     }
19 }
20
21 contract Attack {
22     IEtherVault public immutable etherVault;
23
24     constructor(IEtherVault _etherVault) {
25         etherVault = _etherVault;
26     }
27
28     receive() external payable {
29
30         //This is the fallback functions that handels the reciving
31         //funds from row 11
32         //In the vulnerable contract. That calls upon the widraw
33         //function once more
34         if (address(etherVault).balance >= 1 ether) {
35             etherVault.withdrawAll();
36         }
37
38         //This is first called then the fallback function will reenter
39         //the
40         //contract untill empty
41         function attack() external payable {
42             require(msg.value == 1 ether, "Require 1 Ether to attack");
43             etherVault.deposit{value: 1 ether}();
44             etherVault.withdrawAll();
45         }
46     }
47 }

```

**Figure 3.5:** Example of code that can be exploited using reentrancy and the attacking contract using it's fallback function to reenter the attacked contract

The reentrancy exploit is when the state changes in improper order after an external call, which might cause the contract to obtain an invalid state. The best practice is to always do the external calls last to avoid this hazard. The weakness is introduced when doing a state change after an external call as seen in the code in figure 3.5. When not changing the balance of the caller before in this code example the attacker can use their fallback function to call the withdraw function multiple times before their balance is reduced.

### 3.4.6 Dependence on predictable environment variable (SWC: 116, 120)

There are multiple SWC-ids for this error because of the different environment variables that are present and how they are used in the logic. Two types of dependencies on environment variables are SWC-120 and SWC-116. SWC-116 uses *block.number* or *block.timestamp* to get a time reference. This is not good practice because nodes can only synchronize their time to a limited extent and malicious nodes can alter the time on a block to work to their advantage.

120 describes when a smart contract uses blockchain attributes for deriving randomness see 3.6 for examples of both SWC 120 and 116. Deriving randomness from predictable variables or time from variables controlled by the nodes is bad practice [33].

```
1 contract GuessTheRandomNumberChallenge {
2     uint8 answer;
3
4     function GuessTheRandomNumberChallenge() public payable {
5         require(msg.value == 1 ether);
6
7         //Example of Dependence on a predictable environment
variable
8         //where block.timestamp is used SWC-116
9         require(block.timestamp >= 1546300800);
10
11
12
13         //Example of Dependence on a predictable environment
variable with a predictable source of randomness SWC-120
14         answer = uint8(keccak256(block.blockhash(block.number - 1),
15 now));
16     }
17
18     function guess(uint8 n) public payable {
19         require(msg.value == 1 ether);
20
21         if (n == answer) {
22             msg.sender.transfer(2 ether);
23         }
24     }
25 }
```

**Figure 3.6:** Code used showing bad dependencies on environment variables

### 3.4.7 Jump to an arbitrary instruction (SWC: 127)

Code in 3.7 gives the opportunity for the user to update a chosen memory slot and execute assembly code. This makes the contract vulnerable since it is not provided what assembly code is used. If the assembly code for instance changes the owner of the contract, the contract becomes vulnerable to all the different types of vulnerabilities associated with bad access control. Because of the possibility of a malicious user overwriting critical information like the owner seen in 3.7, this bug makes the contract vulnerable to reachable self-destruct among other access control vulnerabilities.

```

1 contract FunctionTypes {
2     address private owner;
3
4     constructor() public payable {
5         require(msg.value != 0);
6         owner = msg.sender;
7     }
8
9     function withdraw() private {
10        require(msg.value == 0, 'dont send funds!');
11        address(msg.sender).transfer(address(this).balance);
12    }
13
14    function frwd() internal
15        { withdraw(); }
16
17    struct Func { function () internal f; }
18
19    function breakIt() public payable {
20        require(msg.value != 0, 'send funds!');
21        Func memory func;
22        func.f = frwd;
23        assembly { mstore(func, add(mload(func), msg.value)) }
24        func.f();
25    }
26
27    function destroy(address addr) public {
28        require(msg.sender == owner)
29    }
30 }

```

**Figure 3.7:** Code that makes the user able to write to an arbitrary location in the memory and can therefore might overwrite the owner. Line 23 calls assembly code, which is where this vulnerability is present.

### 3.4.8 Hardcoded guards (No SWC)

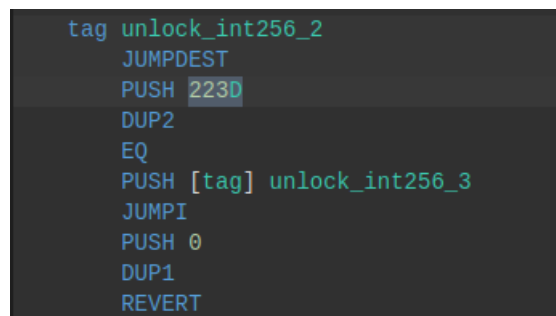
All data stored and executed on the Ethereum blockchain is public. This includes transactions and their associated payloads, as well as the bytecode that is executed using the EVM. Although the bytecode may be less comprehensible than the source code, decompilers ex-

ist for bytecode. Therefore, hardcoding passwords or keys on the Ethereum blockchain is strongly discouraged as these values will be visible to anyone, as illustrated in Figures 3.8 and 3.9. The same security concerns apply to hashed keys, as front-running attacks can exploit any visible transaction not yet completed. Front running is not the focus of this thesis but is an interesting topic that can be read more about following this source [34].

Hard coded guards are something that may not be used that much on the Ethereum blockchain but we still think is relevant for Solidity security tools to check for since the security concerns it brings are of high severity and can cause serious financial losses.

```
1 contract CompositeSuicide {
2     bool locked;
3
4     constructor() public {
5         locked = true;
6     }
7
8     function unlock(int key) public {
9         require(key == 8765);
10        locked = false;
11    }
12
13    function restritctedSudicide() external {
14        require(locked == false)
15        selfdestruct(msg.sender);
16    }
17 }
```

Figure 3.8: Example of hard coded passwords



```
tag unlock_int256_2
JUMPDEST
PUSH 223D
DUP2
EQ
PUSH [tag] unlock_int256_3
JUMPI
PUSH 0
DUP1
REVERT
```

Figure 3.9: Password 8765 visible in bytecode, hex representation 223D

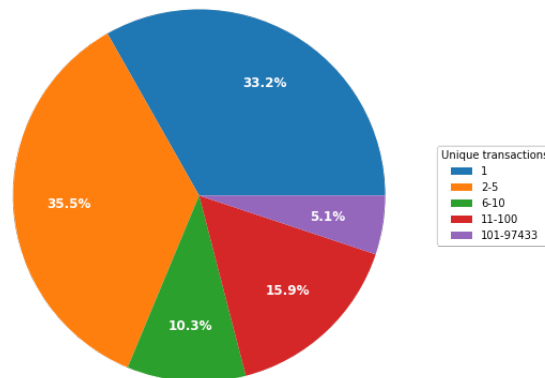
## 3.5 Results and Discussion

This section contains all the data collected from the Ethereum blockchain with Slither, Marian, and Mythril. The graphs represent the surveys that have been done during the project.

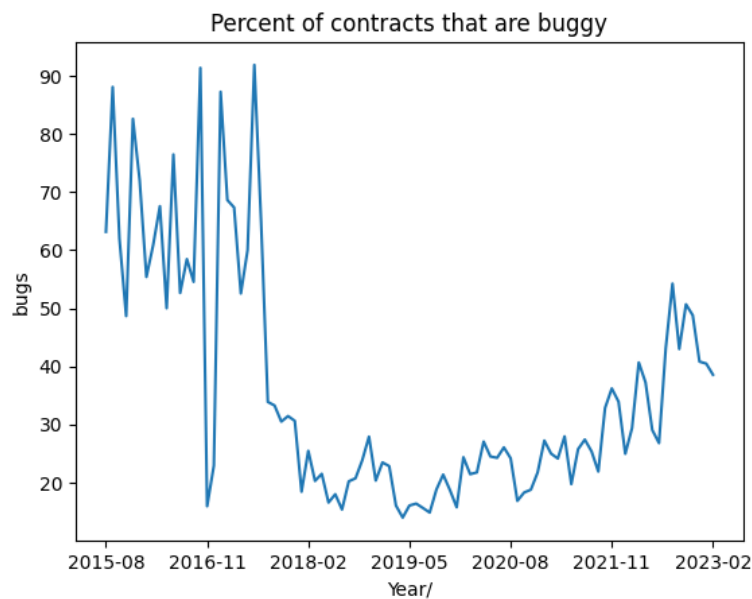
The project has focused on high-severity bugs since they can have the largest negative impact and be the most dangerous. Some medium- and low-severity vulnerabilities have also

been included to see trends in the ecosystem and present more data to track the overall trends.

How many different transactions are sent to unique contracts

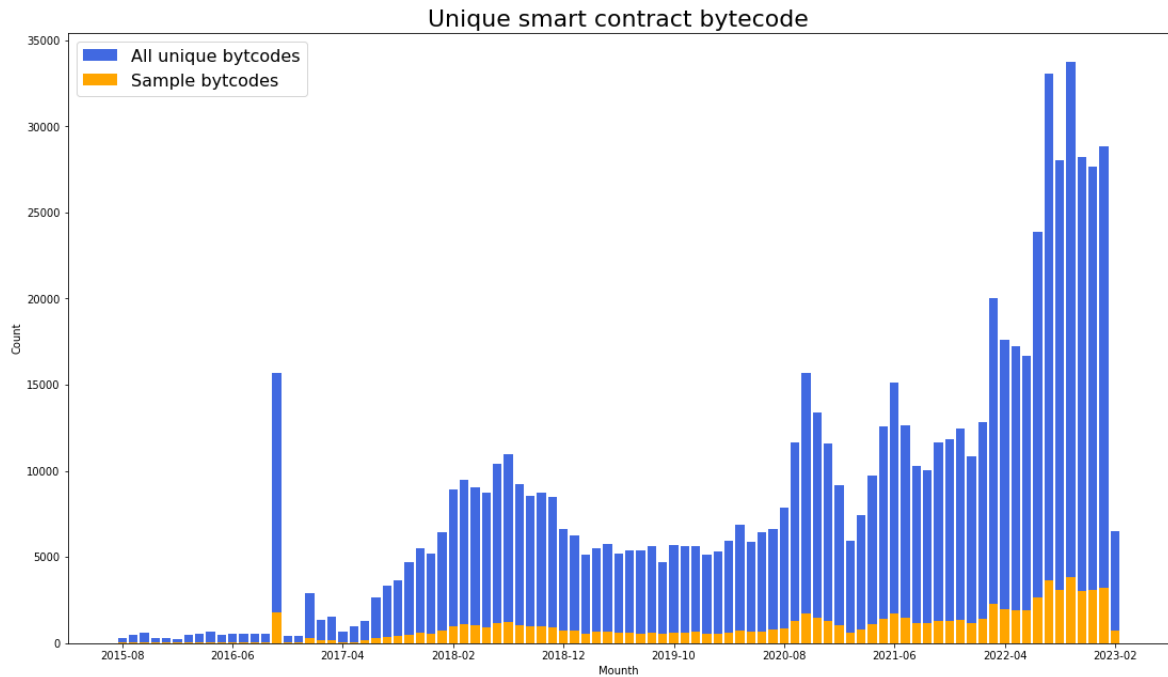


**Figure 3.10:** The number of unique transactions sent to smart contracts that were flagged by Mythril to have a bug. Disclosure this chart does not represent all transactions read more about transactions in the approach.

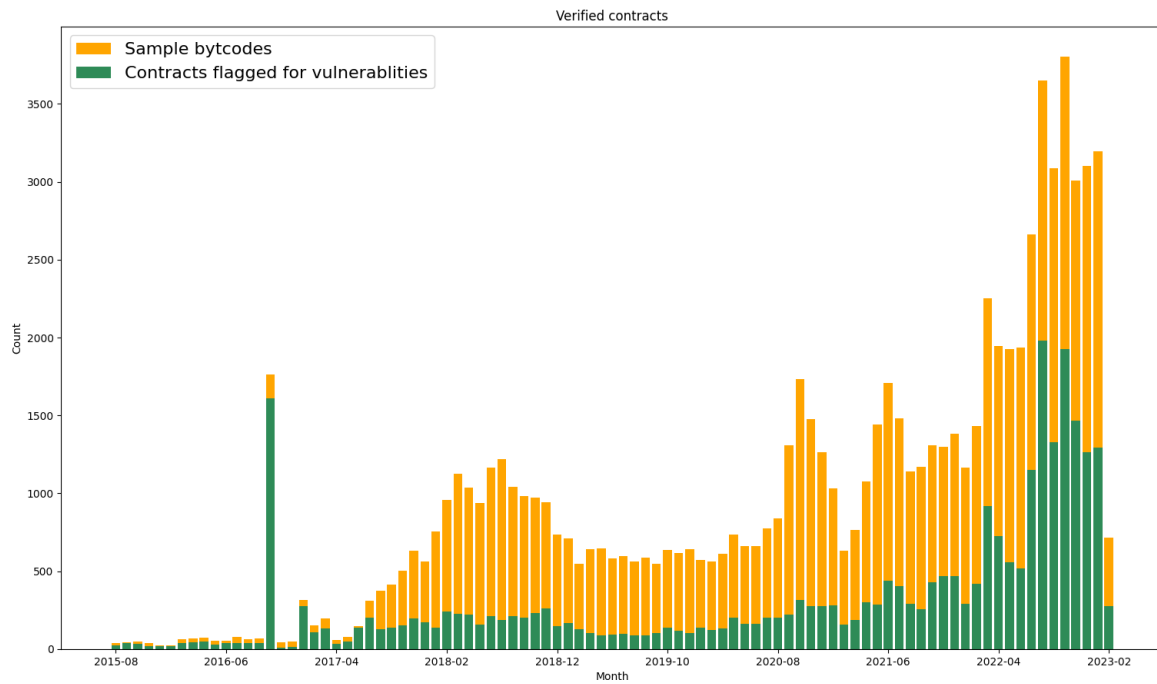


**Figure 3.11:** Showing how many percent of contracts get flagged for vulnerabilities each year, showing a decrease since the launch of the Ethereum blockchain.

As can be seen in 3.14, the number of contracts flagged for vulnerabilities have decreased since the launch of the Ethereum blockchain. This shows that the ecosystem is maturing. It does however showcase a steady increase between 2019 and 2022 which might indicate that the change might only have to do with the updates done to the solidity compilation which prevents a lot of simple bugs that were common early in the development of smart contracts.



**Figure 3.12:** Unique bytecode of smart contracts been deployed per month, Since October 2015, showcasing the distribution of the random sample taken



**Figure 3.13:** Showcasing how many of the sampled contracts were flagged as vulnerable

Bugs reported by Mythril on sampled contracts				
Error	Severity	Verified	Unverified	SWC-id
Exception State	Medium	411	8516	110
Dependence on predictable environment variable	Low	195	1592	116
Dependence on predictable environment variable	Low	12	163	120
Dependence on tx.origin	Low	25	2109	115
Integer Arithmetic Bugs	High	4	242	101
Unprotected self-destruct	High	2	490	106
Unchecked return value from external call	Medium	1	1025	104
Multiple Calls in a Single Transaction	Low	1	484	113
Delegatecall to user-supplied address	High	0	4	112
State access after external call	Medium	0	2	107
State access after external call	Low	0	4	107
External Call To User-Supplied Address	Low	0	11	107
Write to an arbitrary storage location	High	0	1	124
Jump to an arbitrary instruction	High	0	7	127

**Table 3.3:** Bugs detected by Mythril and their severity from the sample contracts divided on verification status.

A hypothesis of ours was that the verified contracts would have fewer bugs than unverified ones. This holds up for all of the bugs not taking severity into account according to table 3.4. But if we consider only the high-severity bugs we can see a smaller difference between the two. This tells us that there are fewer bugs in the verified contracts but the really dangerous bugs are still present even if they are verified. This can also be concluded by figure 3.14 where reachable self-destruct (106 found) is still very present and its presence even increased in 2021 compared to 2020.

Summary of Mythril bugs			
Type of contract	Contracts in sample	Contracts with bugs	High severity
Verified	11711	651 (5.6%)	6 (0.05%)
Unverified	85004	14650 (17.2%)	744 (0.9%)

**Table 3.4:** Summarizing the results from table 3.3 for comparing verified and unverified smart contracts

Bugs reported by Slither on sampled contracts			
Error	Severity	Verified	SWC-id
Unprotected self-destruct	High	46	106
Delegatecall to user-supplied address	High	201	112
Reentrancy exploit	High	1784	107
Dependence on predictable environment variable	High	295	116
controlled-array-length	-	275	-
shadowing-state	-	229	-
unchecked-transfer	-	1447	-
uninitialized-storage	-	20	-
array-by-reference	-	10	-
name-reused	-	37	-
uninitialized-state	-	485	-
public-mappings-nested	-	1	-
incorrect-shift	-	109	-
delegatecall-loop	-	20	-
controlled-array-length	-	275	-
arbitrary-send	-	1036	-

**Table 3.5:** Bugs detected by Slither in the verified contracts within the sampled smart contracts. Slither only works on source code and therefore unverified column is excluded. Only included bugs with high severity and that where able to be mapped to a SWC-code.

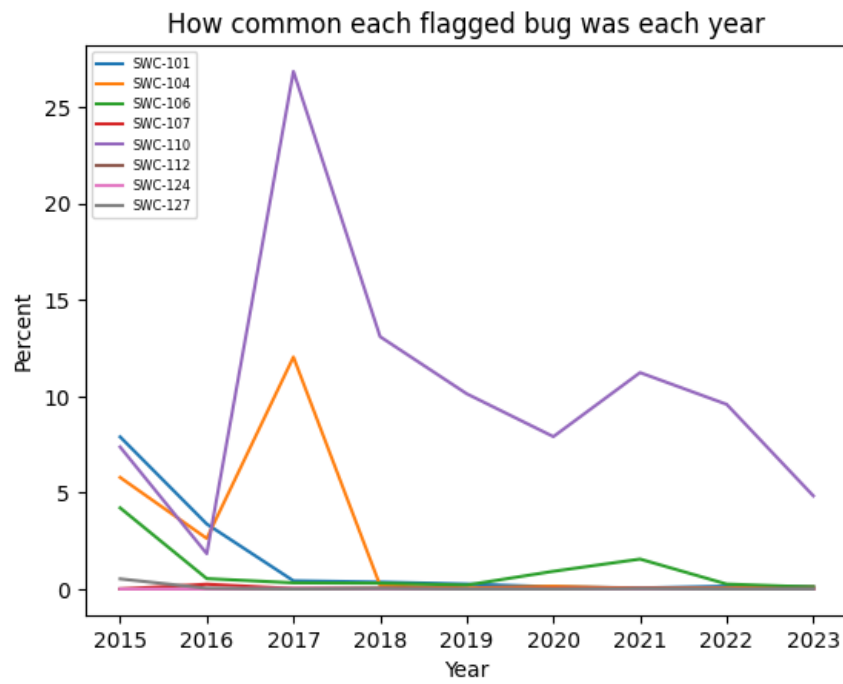
Summary of Slither bugs			
Type of contract	Contracts in sample	Contracts with bugs	High severity
Verified	11711	6270 (53.5%)	2326 (19.9%)

**Table 3.6:** Summarizing the results from table 3.5 for comparing verified smart contracts. This table does only classifies bugs as high severity if they are able to be mapped to a SWC-id. This is because Slither labels a lot as severe, so to make the tables comparable only common SWC-ids where included.

Comparing the numbers from tables 3.6 and 3.4 gives drastically different pictures of these contracts. Slither the static analysis tool flags around 390 times more contracts to have a high severity bug. And to clarify this is only taking into account the bugs that have a SWC-id. If counting more bugs that Slither itself counted as high severity this number would be greater. If excluding reentrancy vulnerabilities this is the one most flagged by Slither then it still flags 50 times more often. According to Slither, see table 3.6, 53% of all verified contracts contain bugs and 20% of them are severe.

The question, which one of these should you trust, still remains. This is a hard question that we are not able to answer just by looking at how contracts are flagged. More specific examples and numbers for accuracy and precision is presented in chapter 4, where we zoom into reachable selfdestruct.





**Figure 3.14:** The prevalence of bugs classified as high or medium severity over time flagged by Mythril

## 3.6 Closing remarks and zooming in on self-destruct

The Ethereum blockchain is continuously evolving with new users and new smart contracts. This process goes in waves and depends on external factors like trends and the general financial markets. As seen in figure 3.11, many early smart contracts have a clear tendency to have bugs in them. Many different things can cause this, but we believe the main reason was the youth of the domain and the Solidity compiler allowing more faulty code to compile. Since the ecosystem still contains all contracts since the launch of Ethereum 9 years ago, all older bugs are still present. When the blockchain launched, many bugs were unknown and used in many contracts. Smart contracts are still in their infancy compared to other technologies, such as the traditional internet. This brings along growing pains that are still prevalent within the ecosystem. Some errors were not classified as mistakes until long after the creation, causing many old contracts to contain those types of vulnerabilities.

As previously mentioned, Maian was the first tool to detect the parity wallet hack mentioned in the background, where a non-authorized user could destroy a smart contract and empty it of its funds. Since then, the type of access control used in the Parity wallet has been deemed unsafe and is no longer commonly used, showing one example of how contracts on the blockchain have evolved.

## Chapter 4

# Self-destruct within the Ethereum blockchain

---

This chapter delves deeper into the results obtained from the previous chapter's analysis, with a focus on assessing the classification performance of different tools for identifying reachable self-destruct instructions, including identifying gray areas such as intended reachable self-destruct. Additionally, an attempt is made to estimate Maian, Slither, and Mythril's false-positive ratio and the reason why contracts are vulnerable to reachable self-destruct. Also, the number of false negatives is tried to be estimated through examining a sample of not flagged smart contracts. A manual review of contracts flagged as being vulnerable to reachable self-destruct was conducted to assess the accuracy of the tools.

The knowledge gained in the manual review creates a small benchmark dataset to discuss where the tools succeed and fall short. The chapter then uses the results from the manual study and benchmark dataset to propose improvements to Slither, the static analysis tool used during the survey.

### 4.1 Definition of reachable self-destruct

Our definition of "reachable self-destruct" refers to any sequence of functions triggered by an arbitrary user that executes a self-destruct instruction. Therefore, contracts with conditional statements that nonauthorized users cannot bypass are deemed safe. The question of identifying reachable self-destruct instructions thus becomes an access control issue, which is discussed further in the next section. We use the term "reachable" rather than "unprotected" self-destruct to account for the possibility that the self-destruct instruction may be protected but circumvented through alternative means.

The only other source we have found discussing reachable self-destruct or, in other terms, composite unprotected self-destruct is Ethainter [10]. They focused on tainted guards that overlap with our work but are more of a general case. They did not have a specific definition focused on self-destruct, and because this thesis focuses on that particular access control case, we chose to introduce the term. In the paper describing Maian, these contracts are called

*suicidal*, which refers to the same type of contracts we refer to as reachable self-destruct, i.e., contracts that can be killed by an arbitrary user[28]. Mythril refers to all of these contracts as unprotected self-destruct[13].

A smart contract might be intentionally designed in a way that it is deemed vulnerable to reachable self-destruct. In our understanding, separating intentional and non-intentional is very hard. Therefore, intentional and non-intentional are considered true positives. It is a critical perspective when analyzing contracts that may appear vulnerable but are intentionally designed to be transparent for legitimate reasons, which will be discussed in a later subsection.

## 4.2 Access Control

Access control encompasses various methods of regulating access to resources, with the overarching goal of determining who can access what and how access is enforced [2]. Access control works differently on the Ethereum blockchain than on your computer or the web, where you usually identify yourself with a username and password. Because everything is public on the blockchain, you can not send a password or username without revealing it to everyone. The unique identifier used for access control on the blockchain is your address.

Access modifiers in solidity provide a fundamental mechanism for imposing restrictions on the usage of a specific function. As illustrated in figure 4.1 and documented in [3], the access control logic is encapsulated in a separate modifier function rather than implemented using an if statement. In practice, using `if` statements as access control are frequently replaced by the `require()` function. Among the manually reviewed contracts, the most common modifier used was `onlyOwner`, as exemplified in figure 4.1, where it is shown how `onlyOwner` is often used in real-world contracts.

```

1 contract Purchase {
2     address public owner;
3     mapping (address => bool) public Sellers;
4
5     constructor() payable public {
6         owner= msg.sender;
7         // msg.sender represents address that is being called
8     }
9
10    modifier onlyOwner() { // Modifier
11        require(
12            msg.sender == owner,
13            "Only owner can call this."
14        );
15        _;
16    }
17
18    function assign_seller(address addr) public view onlyOwner {
19        // Modifier usage
20        Sellers[addr] = true;
21    }
22 }

```

**Figure 4.1:** Example of how a modifier can be used to secure a smart contract. See how the modifier `onlyOwner` on row 10 is used in row 18. [21]

Role-based access control (RBAC) is an alternative approach to managing access. It involves assigning distinct roles to different users that can have distinct access rights. This method may be implemented using modifiers, as outlined in the following source [2].

The `require()` function operates in a similar manner to a database transaction. If the specified requirement is not met, the contract is reverted to its prior state, as explained in the Solidity documentation [3].

Subsequent subsections discuss potential weaknesses in access control mechanisms.

## 4.2.1 Flawed Access Control

The first type of vulnerability is called "Unprotected self-destruct." It occurs when the self-destruct function is not protected with any `require()`, modifiers, or if statements. This vulnerability can be easily identified by Mythril and Slither.

The second type of vulnerability is a composite vulnerability [10]. This vulnerability arises when user privileges can be altered to circumvent security requirements through multiple invocations of the contract, making the contract susceptible to attacks. See figure 4.2 how the owner can be changed. We have defined unprotected and composite self-destruct under one concept, reachable self-destruct.

Another example of composite self-destruct can be made with "Write to arbitrary storage location" (SWC:124) from Chapter 3, where the owner in the case of figure 4.2 could be changed to circumvent the access control, of course, needs the logic for writing to an arbitrary memory location, seen in figure 3.4.

"Jump to an arbitrary instruction" (SWC: 127) is also a way to circumvent access control where a user can execute an arbitrary instruction without fulfilling a function's requirements. For example, see Chapter 3, figure 3.7.

```
1 contract CompositeSuicide {
2     address owner;
3
4     constructor() public {
5         owner = msg.sender;
6     }
7
8     modifier onlyOwner {
9         require (owner == msg.sender);
10        _;
11    }
12
13    function changeOwner(address newOwner) public {
14        owner = newOwner;
15    }
16
17    function sudicideByOwner() public onlyOwner {
18        selfdestruct(msg.sender);
19    }
20 }
```

**Figure 4.2:** Example of a composite self-destruct, where anyone can change the owner of the contract making it possible the destroy the contract and steal the funds.

## 4.2.2 Intended reachable self-destruct

Determining whether a feature represents a vulnerability or legitimate functionality requires a nuanced evaluation that the solidity security tools in this paper are not able to do. An example of this can be seen in figure 4.3 where if a person guesses the right number, the person shall be able to destroy the smart contract and receive the funds currently belonging to it. These kinds of contracts have been chosen to call true-positives since an arbitrary user can kill them under the right conditions.

```

1 contract Lottery{
2     // creates random number between 0 and 1 on contract creation
3     uint256 private randomNumber = uint256( keccak256(now) ) % 2;
4     ....
5     function play(uint256 _number) public payable {
6
7         // if player guesses correctly, transfer contract balance
8         // else transfer to owner
9         if (_number == randomNumber){
10             selfdestruct(msg.sender);
11         }else{
12             selfdestruct(owner);
13         }
14     }
15     ....
16 }
```

**Figure 4.3:** Example of an intended reachable self-destruct in the context of a lottery. Side note this contract contains another vulnerability when it uses a predictable environment variable; see Chapter 3 "Dependence on predictable environment variable" for further details.

## 4.3 Slither's static analysis of reachable self-destruct

The analysis performed by Slither for determining whether a program is vulnerable to self-destruct involves two hard-coded steps.

Firstly, it examines if the modifier's name is "*onlyOwner*" or if *msg.sender* is employed directly in the modifier's `require()` statement. Relying on a function name is not advisable since it is an unreliable source of information. For instance, the *modifier\_faulty\_onlyOwner* in the benchmark table 4.5 is incorrectly classified as safe when it is protecting its self-destruct with an always true statement, `require(1 == (2/2))`. Slither classifies functions using this modifier as safe, but the team acknowledges that this type of method results in false positives and false negatives.

Secondly, Slither checks if *msg.sender* is utilized directly in the modifier's `require()` statement. As a result, the test case *modifier\_faulty\_requirement* from the benchmark dataset

4.5 passes even though the requirement is always true, as seen in *require(msg.sender == msg.sender)*.

Currently, Slither does not perform any analysis to determine if external users can alter permissions. Consequently, it cannot handle composite vulnerabilities where the owner of the contract can be changed [18].

## 4.4 Detecting reachable self-destruct with Mythril's symbolic execution

Symbolic execution allows Mythril to create a comprehensive model of the smart contract's behavior, including its possible states, inputs, and transitions. By analyzing this model, Mythril can identify instances where a self-destruct instruction can be executed, thus potentially destroying the contract and causing a loss of funds or other undesired consequences.

The process of detecting reachable self-destruct vulnerabilities with Mythril's symbolic execution can be outlined as follows:

- Create a control flow graph of the smart contract's different states. This graph encompasses all possible states and inputs, as well as the resulting transitions.
- Analyze the graph to identify any instances where the self-destruct instruction can be executed. This includes checking for conditions that allow any user to trigger the self-destruct instruction, such as unprotected function calls or faulty access control mechanisms.
- Generate a detailed report of the identified vulnerabilities, along with the corresponding transaction traces that can lead to the execution of self-destruct. This report assists developers in understanding the root cause of the vulnerability and provides guidance on how to remediate the issue.

By employing symbolic execution, Mythril can detect reachable self-destruct vulnerabilities in Ethereum smart contracts, enabling developers to address these security risks before deploying their contracts on the blockchain [13].

## 4.5 Method

To evaluate the results from the smart contract security tools, we needed to be able to read the contracts ourselves. Therefore we chose to evaluate only verified contracts for which we had the corresponding source code. After obtaining the verified contracts, we filtered out the smart contracts that contained at least one self-destruct instruction to minimize the time for analysis, see table 4.1. After filtering out the relevant contracts, we ran the tools on them and got their results. Every flagged contract was manually reviewed to find false positives; see figures 4.4 4.6 to see results. Some of the true positive contracts were also cross-referenced. We tried to find out why the other tools did not flag contracts containing a vulnerability. To find out the ratio of false negatives, a random sample of 10% of the contracts with a self-destruct that had not been flagged were selected and manually reviewed to find



any vulnerabilities that were not detected; see figures 4.5 4.8 to see results. Finally, we created a benchmark dataset with smart contracts that summarized the weaknesses of the different security tools, see table 4.5.

Type	Amount
Verified smart contracts	115852
Verified smart contracts with self-destruct Instruction	2231

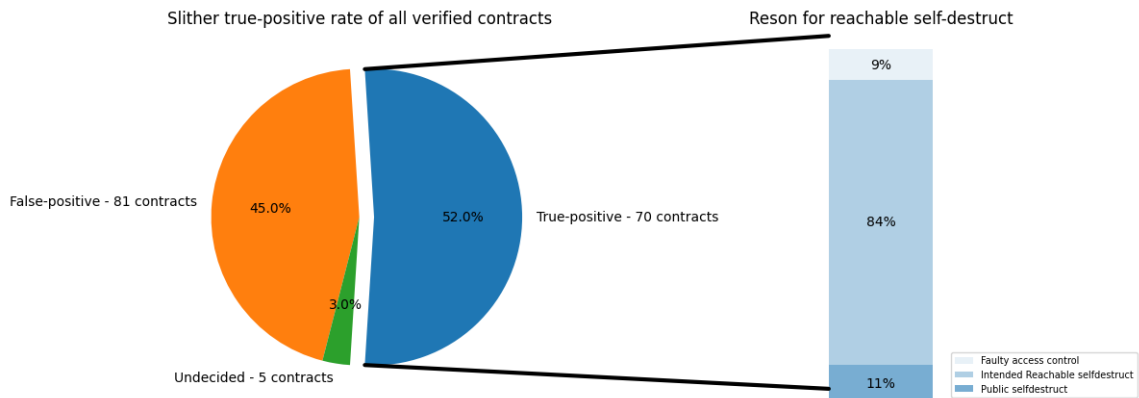
**Table 4.1:** Showing the amount of

## 4.6 Results and Discussion

This section presents true/false negatives and true/false positives, and the reason for them is discussed. An overview of the overlap of the different tools is shown to visualize how much the tools agree. Lastly, we present our benchmark dataset with some of the lessons learned from this deep dive. The results of Maian will be briefly discussed but no tables are presented since it did not flag enough contracts for the result to be significant.

### 4.6.1 Manual review

#### Slither

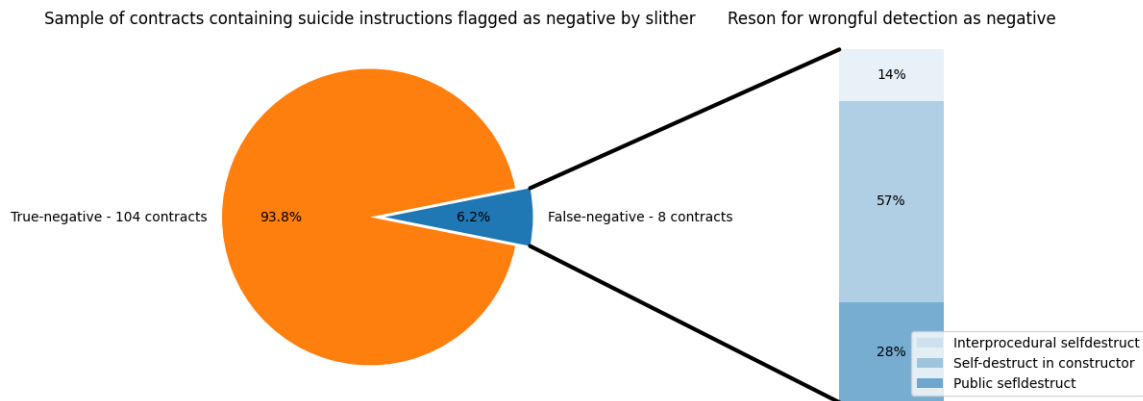


**Figure 4.4:** Accuracy of Slither on verified contracts with reasons for contracts being vulnerable of reachable self-destruct

As can be seen in figure 4.4, Slither has a 52 percent true positive rate of the flagged contracts in the verified dataset. This number is reached through manual review and is the ground truth we use in this thesis. More contracts could have been used and more people could have been involved in the labeling to get a better ground truth. As can be seen in figure 4.4, most contracts that contain reachable self-destruct does so on purpose. One example of this can be a lottery contract which will be killed when a winner has been found. Out of the 155 flagged contracts flagged by Slither, 95 still existed on chain in January 2023, which shows that contracts containing self-destruct are often destroyed. However, from our manual review, we

deem it highly likely that this is due to their owners destroying them. The manual review concluded that the true positive rate for the destroyed contracts was 64 percent, slightly higher than for the alive contracts.

You could argue that smart contracts with intentional reachable self-destruct would be a false positive but we count them as true positives because of the difficulty for smart contract security tools to understand the intent of the author. An example of a true positive with reachable self-destruct can be seen in the example code in figure 4.3. Further examples of such contracts include those that are created for one-time use.



**Figure 4.5:** Manual review on a sample of 5 percent of verified contracts containing self-destruct instructions flagged as negative by Slither

As seen in figure 4.5, Slither mislabeled some contracts flagged as safe too. The majority of the false negatives come from self-destruct in the constructor of the contract, which Slither is programmed to label as safe. All the contracts with self-destruct in the constructor are reviewed to be put there on purpose. That approach can be considered good practice implemented by Slither, but that counts on programmers knowing what they are doing, which comes down to the tool's usability. These contracts are still considered unprotected self-destruct, but we understand why these contracts are not flagged.

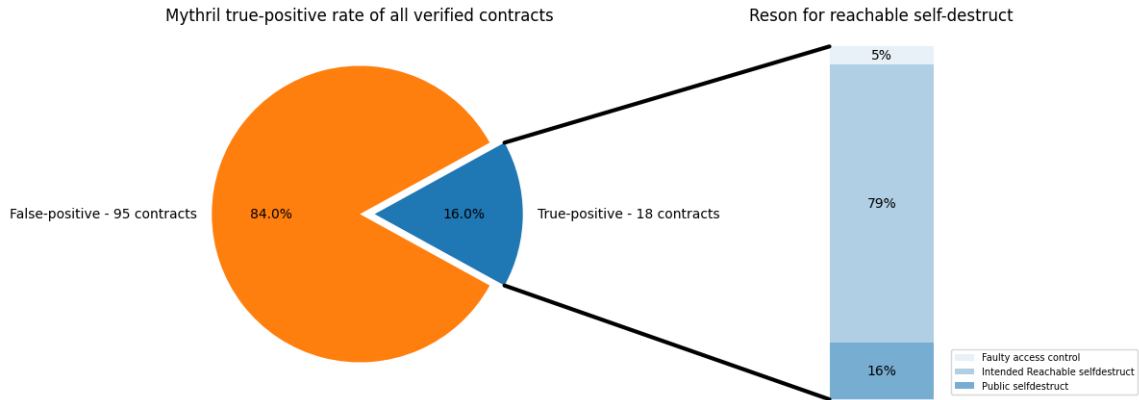
One not flagged contract has an internal (private) function containing self-destruct reached from a public external function, which is still flagged as safe. This refers to Slither's problem in handling inter-procedural functionality that needs modifiers. This example can be found in the benchmark dataset 4.5 named *internal\_function*.

## Mythril

With increasing the call depth limit for the symbolic execution in Mythril an increasing amount of contracts are flagged for reachable self-destruct. This made us do a small measure of how much call depths affected Mythril's ability to find reachable self-destruct vulnerabilities; see table 4.2. This was done so that a relevant call depth could be defined before running contracts to find the intersection between Slither, Maian, and Mythril. The conclusion was that a depth of 15 and a timeout of 240 seconds would find most of the contract deemed vulnerable by Mythril in the verified dataset.

Depth evaluation										
Timeout (seconds)	120	120	120	120	120	240	240	240	240	240
Call Depth	3	5	10	15	20	3	5	10	15	20
Detected	1	3	9	16	14	1	3	9	16	16
TimedOut	0	0	4	7	14	0	0	0	0	0

**Table 4.2:** Running Mythril with different call depths and different timeouts on 95 verified contracts containing bytecode, flagged by Slither



**Figure 4.6:** Accuracy of Mythril on verified contracts with reasons for contracts being vulnerable of reachable self-destruct

Figure 4.6 divides the reasons for flagging the true positives. The false positives shown in figure 4.6 were all the same kind of contract; therefore, no division was needed. The problem with the contract was that it used the initialization of other contracts in function logic. Similar to other object-oriented programs. Initialized smart contracts made the symbolic execution unable to understand `adminVault.admin == msg.sender` seen in figure 4.7, and interpret the modifier as unsafe even if it is not.

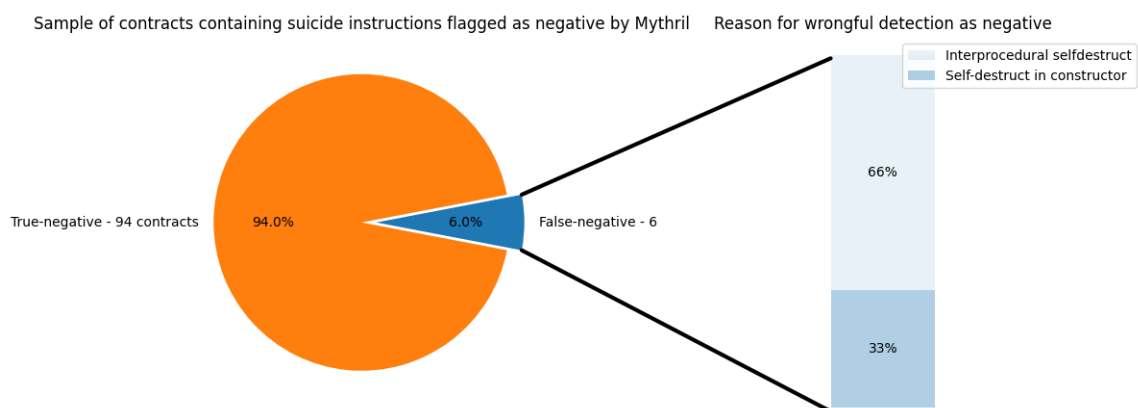
The true positives had the same trend as Slither; with it, many of the smart contracts flagged were intentionally programmed to be self-destructed.

```

1 contract AdminVault {
2     address public owner;
3     address public admin;
4
5     constructor() {
6         owner = msg.sender;
7         admin = 0x25eFA336886C74eA8E282ac466BdCd0199f85BB9;
8     }
9 }
10
11 contract AdminAuth {
12     AdminVault public constant adminVault = AdminVault(
13     ADMINVAULT_ADDRESS);
14     error SenderNotAdmin();
15
16     modifier onlyAdmin() {
17         if (adminVault.admin != msg.sender){
18             revert SenderNotAdmin();
19         }
20     }
21
22     function kill() public onlyAdmin {
23         selfdestruct(payable(msg.sender));
24     }
25 }

```

**Figure 4.7:** Boiled down code of all the false positives in figure 4.6. Showing a weakness of Mythril not being able to handle cross-contract objects. The problem accrues on row 16, where an object of AdminVault is used to access the admin address; Mythril can not handle the interprocedural analysis.



**Figure 4.8:** Manual review on a sample of 5 percent of verified contracts containing self-destruct instructions flagged as negative by Slither

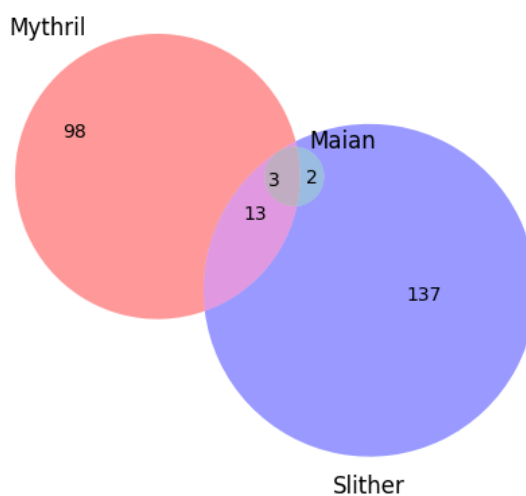
When going through a sample of 5% of the smart contracts not labeled positive by Mythril,

see figure 4.8 to approximate its false negatives, 6 false negatives could be found. The majority of them had initialized smart contracts inside of them that made the symbolic execution not able to process the whole control flow of the program and therefore break and classify the contract as safe even if that was not the case.

Tool	Reachable self-destruct found in 480 000 smart contracts currently on chain	Tool	Vulnerabilities found in 117 000 verified smart contracts
Maian	674	Maian	6
Mythril	480	Mythril	114
		Slither	155

**Table 4.3:** Comparision between reachable self-destruct found in verified contracts and all contracts

In the left table of figure 4.3, all unique contracts on chain until October 2021 were analyzed to get a general picture of how many would be flagged for reachable self-destruct. Only Mythril and Maian were used since Slither does not take bytecode, and we did not find any good decompiler to convert the bytecode into source code. Interestingly, Maian finds so many contracts with reachable self-destruct since it finds the least by far in the verified dataset and does not detect any out of the benchmark. This indicates that Maian was good at detecting reachable self-destruct when it was released but would need maintenance to keep up with the development of the EVM optimizations of bytecode.



**Figure 4.9:** Overlap of tools on the verified contracts on-chain untill of July 2022

Tool	Amount	False Negatives	False Positives
Mythril	114	6	95
Slither	160	8	43
Maian	6	-	-

**Table 4.4:** Summarized numbers for the different security tools show in figure 4.9 and their corresponding false positive/negative

## 4.6.2 Closing remarks on zooming in on reachable selfdestruct

In smart contract security, a high level of certainty is required because of the stakes of losing funds. That risk makes the completeness of a security analysis critical. Symbolic execution flags real vulnerabilities but misses some contracts classed as unprotected self-destruct. Static analysis over flags but also misses some of the more obvious true positives seen in figure 4.9. The figure shows that both symbolic execution tools miss so many true positives that it becomes redundant to look for this vulnerability through their use of them. Even though both Maian and Mythril had an accuracy of 100 percent, they both missed 88 of the contracts flagged by Slither, of which 45 percent were true positives. The imperfect completeness also gives a picture of how immature the field of analysis of smart contracts is. Maian claims to be able to flag composite vulnerabilities, which it indeed does, but it does not flag enough vulnerable contracts compared to other tools. Mythril has the same dilemma regarding contracts susceptible to reachable self-destruct; at least, it does flag more access control problems that Maian does not consider. The symbolic execution tools can handle more complex vulnerabilities, but some of their complexity makes them miss simple cases that Slither, with its simple rules, can detect.

The contracts flagged for reachable self-destruct often contain vulnerabilities that can be applied to more than self-destruct. More bugs in the access control category have the same type of problems within the code. One simple example is being able to change the owner of a contract since if an attacker can become the owner, many vulnerable functions become available to the attacker. Other vulnerabilities this can be applied to are *Jump to an arbitrary instruction* and *Delegatecall to user-supplied address* as can be seen in figures 3.3 and 3.7.

## 4.7 Benchmark dataset

We derived the benchmark dataset from our prior expertise in program analysis, coupled with the insight from the study in Chapter 3 and the manual review of the contracts flagged for having reachable self-destruct by the Maian, Slither, and Mythril. Some contracts were also taken from the Smartbugs benchmark dataset for access control vulnerabilities.[6] The benchmark dataset can be accessed on [https://github.com/Frallan97/reachable\\_self\\_destruct\\_benchmark](https://github.com/Frallan97/reachable_self_destruct_benchmark). [8]

Tools on created contracts					
( <i>x detected bug</i> )	Slither	Mythril	Maian	Inspired by manual review	Taken from smartbugs dataset
<code>composite_selfdestruct_password</code>					
<code>composite_selfdestruct_recursion</code>					
<code>composite_selfdestruct_transfer_ownership</code>		X	X		
<code>modifier_faulty_onlyOwner</code>		X			
<code>modifier_faulty_requirement</code>		X			
<code>double_modifier</code>	X			X	
<code>cross_contract_variable</code>	X			X	
<code>cross_contract_selfdestruct</code>	X			X	
<code>forloop_selfdestruct</code>	X	X			
<code>password_selfdestruct</code>	X	X			
<code>recursive_selfdestruct</code>	X	X			
<code>simple_selfdestruct</code>	X	X		X	X
<code>suicidal_fake_guard</code>	X	X		X	
<code>arbitrary_jump</code>	X	X			
<code>arbitrary_write</code>	X	X			
<code>internal_selfdestruct_function</code>		X		X	
<code>delegatecall_arbitrary_address</code>		X		X	
<code>parity_wallet_hack_1</code>					X
<code>parity_wallet_hack_2</code>					X

**Table 4.5:** A table of what cases of reachable self-destruct Slither, Mythril, and Maian can catch. A hint of what kind of flaw is present can be seen in the name. The Solidity code can be found on [here](#). "Inspired by manual review" column represents boiled-down smart contracts that were inspired by our manual review.

- `composite_selfdestruct_password` - contains a change owner function that can be unlocked with a password. Similar to the code in figure 3.8 but with a separate change owner method.
- `composite_selfdestruct_recursion` - contains a change owner function with recursion limited to a depth of two.
- `composite_selfdestruct_transfer_ownership` -
- `modifier_faulty_onlyOwner` - a modifier named only owner but will always return true.
- `modifier_faulty_requirement` - modifier have a requirement containing `msg.sender == msg.sender`.
- `double_modifier` - Containing multiple modifiers that are dependent on each other.

- *cross\_contract\_variable* - instantiating a contract inside another one, similar to instantiating a class in java. Checking if tools can handle cross-contract analysis.
- *cross\_contract\_selfdestruct* - Instantiating a smart contract and using a function with unprotected self-destruct.
- *forloop\_selfdestruct* - checks if the tool can analyze for loop correctly.
- *recursive\_selfdestruct* - self-destruct inside recursion
- *simple\_selfdestruct* - open public self-destruct instruction
- *suicidal\_fake\_guard* - a guard that is always true based on hard-coded values.
- *arbitrary\_jump* - user is able to execute arbitrary smart contract instructions and skip requirements.
- *arbitrary\_write* - Can overwrite arbitrary memory, for example making a contract vulnerable to reachable self-destruct.
- *internal\_selfdestruct\_function* - exposing self-destruct using an internal (private) function.
- *delegatecall\_arbitrary\_address* - Might expose the smart contract to selfdestruct through logic from other smart contracts.
- *parity\_wallet\_hack\_1* & *2* - Are example contracts of the parity wallet bug that occurred in 2018. It is explained in more depth in the text below.

Looking at the results we can see that Slither and Mythril flag a lot of the benchmark dataset, see table 4.5 but Maian can not find any of the built-in vulnerabilities, which may not be that strange because it is not maintained anymore, with its last update in 2018 [23].

This dataset focuses on ways to create false-negatives for the Solidity security tools it might not always be the case that many of their kind are present on the Ethereum blockchain.

A study conducted in 2019 about the treewidth of Solidity control flow graphs examined the existing verified smart contracts and concluded that none of them contained instances of recursion [11]. It is important to note that this conclusion only pertains to verified contracts. Actors with bad intent seeking to defraud users would not want to have their smart contracts verified and, therefore composite vulnerabilities with recursion matters. Our own research did not specifically focus on the presence of recursion, so it is unclear how frequently such smart contracts are deployed today or are currently present on the Ethereum blockchain. However, it is worth noting that if recursion was combined with the composite vulnerability of allowing the owner of a contract to change, a malicious user could conceal the vulnerability from both static and symbolic execution analysis tools. It is meant to be a collection of contracts to evaluate current and future tools' ability to detect reachable self-destruct.

To be noted is that two versions of the previously mentioned parity-wallet hack are included in the data-set. This bug is, to this day, the case where reachable self-destruct has had the largest monetary impact. The vulnerability is an access control issue where the number of owners can be set to 0 so that anyone is allowed to call on protected functions.



Note that Slither cannot detect this since it is a composite vulnerability. Maian does not detect this either, but it has another version of the same bug that it detects included in its own example dataset. This can be explained through that it is an older bug that Maian can handle, as well as that Maian was built to find composite vulnerabilities like the parity wallet hack. The examples of the Smartbugs dataset were taken from the smart bugs repository. [6]

## 4.8 Limits of tools and possible improvements

### 4.8.1 Can you detect intended reachable self-destruct

Whether to label intended reachable self-destruct as a true positive or a false positive comes down to what expectations to put on tools. We decided to label the intended reachable self-destruct as a true positive since it is hard for both symbolic execution and static analysis to distinguish if the contract developers put the reachable self-destruct as a feature or if it should be considered a bug. Many of these contracts depend on hard-coded variables such as time or a random number generator for organizing a lottery. To label these contracts correctly, the tools would need to understand the purpose of the analyzed contract, which is outside the current state of static analysis and symbolic execution and therefore intended reachable self-destruct should be labeled as a true positive. This labeling can even be seen as a tool feature since developers intending to implement a reachable self can see that they succeeded. This can also ensure detection in cases where developers have a reachable self-destruct in a contract which might seem intentional but is not.

One way to approach this problem would be to use a machine learning model with a neural network trained to find code. This has not been done yet. Machine learning to find vulnerabilities in smart contracts has been attempted, with the model being based on results produced by Slither and using the byte code from these [30]. The study tried using both models of CNN (convolutional neural networks) and one model of LSTM(Long term short memory neural networks). The results showed enough accuracy to prove that this approach can be viable when using CNNs. However, the training of this neural network was based on the results of Slither and is therefore limited to the tool's capacity. To be able to outperform tools, a dataset with more reliable ground truth is needed [30].

### 4.8.2 Slither

Slither is the tool that is the quickest and most reliably flags vulnerable contracts. This does not mean that Slither does not have flaws or possible improvements. It does not handle interprocedural vulnerabilities where the contract calls its own

**internal** (private) functions. Slither fully relies on its check if a function is protected or not, and if a function is protected it is classified as safe. However, it does not check for internal calls (calls to private functions), which makes this an issue. We propose that the tool implement some pointer-to or taint analysis to solve this. This would make the tool interprocedural and able to handle more complex programs. We acknowledge that this would create new false negatives and increase the run time of the analysis, but for the severe vulnerabilities it could prohibit, it would be worth the trade-off. This type of analysis could also reduce the total amount of false negatives if it were implemented instead of the current modifier

detection to determine if a function is protected by the access control implemented in the contract.][17]

As mentioned in Slither's paper it uses taint analysis for some of its analyses [18], but not in the case of finding unprotected self-destruct. For Slither to be able to find the composite vulnerabilities in the benchmark dataset, taint analysis could be the solution. Taint analysis is a program analysis technique that, in summary, contains three steps.

1. Identify the tainted data, in other words, the data that can be set or manipulated by an external user.
2. Identify sensitive sinks like a requirement in a self-destruct function.
3. Lastly, ensure no tainted data is used in the sensitive sinks.

This type of analysis can be automated with points to analysis that builds a graph representing how data flows through a program. An illustration of the taint analysis using pointer analysis could be implemented, which can be seen in the figures 4.10 4.11 4.12.

```
contract CompositeSuicide {  
    address owner;  
  
    constructor() public {  
        owner = msg.sender;  
    }  
  
    modifier onlyOwner {  
        require (owner == msg.sender);  
        _;  
    }  
  
    function changeOwner(address newOwner) public {  
        owner = newOwner;  
    }  
  
    function suicideByOwner() public onlyOwner {  
        selfdestruct(msg.sender);  
    }  
}
```

**Figure 4.10:** Identifying tainted data, data that external users can change

```
contract CompositeSuicide {
    address owner;

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

    modifier onlyOwner {
        require (owner == msg.sender);
    }

    function changeOwner(address newOwner) public {
        owner = newOwner;
    }

    function sudicideByOwner() public onlyOwner {
        selfdestruct(msg.sender);
    }
}
```

**Figure 4.11:** Identifying sinks where tainted data should not be able to reach

```
contract CompositeSuicide {
    address owner;

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

    modifier onlyOwner {
        require (owner == msg.sender);
    }

    function changeOwner(address newOwner) public {
        owner = newOwner;
    }

    function sudicideByOwner() public onlyOwner {
        selfdestruct(msg.sender);
    }
}
```

**Figure 4.12:** Doing a pointer analysis to see what different values, in this case owner can take and check if the sink and tainted data are connected. The result is that they are, and the program contains reachable self-destruct

### 4.8.3 Mythril

Mythril outperforms Slither and Maian in the benchmark dataset, detecting 11/17 reachable self-destruct. Mythril may find a lot of different kinds of self-destructs planted inside, but it is harder for it to handle larger, more advanced smart contracts. Even though Mythril flags 114 contracts, 95 of them are the same type of false negative, as can be seen in figures 4.4 4.6 for reference. After analyzing the different contracts that were not labeled by Mythril but by Slither, it was clear that Mythril had a hard time when smart contracts were being used with each other. Also, all its false positives were contracts initialized inside contracts.

Using the default settings of recursion depth of three for Mythril, only 1 of the 95 flagged (49 true positives) contracts by Slither was flagged to contain reachable self-destruct; this can be seen in table 4.2. Because when the recursion depth was increased to 15, 18 smart contracts of all verified smart contracts could be detected with 95 false positives.

The low default value of the recursion depth makes it very likely that Mythril has missed many reachable self-destructs during its lifetime.

Mythril has the ability to analyze a program and create a transaction trace to follow the vulnerability. But if you increase the complexity of a smart contract, for example, *composite\_selfdestruct\_password* that has a 2-step vulnerability and is not detected by Mythril. We are unsure why this is the case and can not understand it from the source code in Mythril.



# Chapter 5

## Related work

---

Here we present articles and studies related to ours that we either took inspiration from or investigate similar topics to our own.

### 5.1 Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts

The most similar study to our was done by the developers behind Smartbugs. [6]. The study is mentioned in section 2.3.4 and is a paper that inspired us to write this thesis. It goes through the results of running nine automated tools on two data sets, one consisting of 47 587 verified smart contracts and one created by the team at Smartbugs with 69 annotated smart contracts containing known vulnerabilities.

The access control part of the annotated contracts contains 18 contracts, and out of those, four are directly related to reachable self-destruct, and 6 contain bugs that can lead to a contract being vulnerable to reachable self-destruct. Three of the contracts from our benchmark dataset were taken from these contracts, while some others were inspired by it but modified to include reachable self-destruct. Both Mythril and Slither label four contracts from this set of contracts consistent with our results. Maian labels zero, which is also consistent with the results achieved in our study. The only reachable self-destruct that all tools detect in this dataset is the `simple_selfdestruct`, which is included in our benchmark dataset. The authors of the Smartbugs paper propose running a combination of Slither and Mythril to find the most bugs compared to execution time and memory requirements, which is the combination we have analyzed in this thesis. This combination finds 31% of the bugs in their benchmark dataset, which they acknowledge is a low number and similar to the conclusions we drew.

The Smartbugs study presents metrics similar to ours regarding activity within smart contracts on the Ethereum blockchain, such as different categories of vulnerabilities over time. The graph presented for this makes it hard to compare to our results seen in figure 3.14, and no numbers were presented to back the graph. This graph mainly shows a heavy correlation between the number of contracts deployed and the number of bugs present. The nine tools found possible vulnerabilities in 93% of the analyzed contracts, which the authors acknowledge points to many false positives. This has much to do with the symbolic execution tool Oyente labeling 73% of all contracts as vulnerable. [16]

There are some differences to our own study. When describing the overall state of the blockchain they did not investigate the amount of transactions conducted by flagged contracts. We also went into more depth when analyzing the precision and accuracy of the tools, by manually reviewing a sample, while they solely used their benchmark dataset. By taking this sample we were also able to estimate the number of false negatives with numbers that reflect more on how contracts on the actual chain look.

## **5.2 A critical comparison on six static analysis tools: Detection, agreement, and precision**

A similar study was conducted on 47 large open-source Java projects in 2023[26], where the authors ran the projects on the bytecode of the Java classes and then analyzed the results. Six different static analysis tools were used, and 13 554 762 issues were found. These tools investigate code quality in general and not only security, like the tools used in our survey, which can explain the many issues found. After the analysis, the authors took a sample just like we did and compared the results of the tools. They defined a manual ground truth for each issue to find the accuracy of each tool. The accuracy of the tools ranged from 19% to 86%. This shows that static analysis can be highly accurate when analyzing faulty code. However, these results might need to be more accurate because the tool with the highest accuracy had a limited scope of issues that it detected. It only checked for design errors and wrong syntax. For the tools that did take in more complex issues, such as security and bugs, the best tool had an accuracy of 52%. The authors conclude that many of the simpler issues flagged could be solved through re-factorization tools and therefore automated with the static analysis tools. The authors took a sample similar to our own method by randomly selecting between 300 and 400 issues and then manually reviewing these to get accuracy numbers. However, they did not sample for false negatives as we did.

The study showcases similar results to our own when analyzing the capabilities of current static analysis tools. The most comparable one in our survey would be Slither, which runs on source code and uses static analysis techniques. The study shows that tools succeed in finding simpler issues while struggling to find more complex ones, similar to our own conclusion of Slither.



## 5.3 Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities

Another tool we wish we could have included in our study is Ethainter[10], a static analysis tool that can run on byte-code. Ethainter can detect five types of vulnerabilities, classified by the authors:

- *Tainted Owner Variable*
- *Tainted delegatecall*
- *Accessible self-destruct*
- *Tainted self-destruct*
- *Unchecked Tainted staticcall*

Our definition of reachable self-destruct can be seen as a combination between reachable self-destruct and tainted self-destruct since they lead to unauthorized access to self-destruct instructions. Unfortunately, Ethainter is not open source, so we could not include it in our study.

When launched in 2020, Ethainter was presented with a paper that compared Ethainter to other program analysis tools based on the results obtained when testing the tool. During these tests, it was run on all 240 000 contracts with unique bytecodes on the Ethereum blockchain. A small random sample of these contracts was taken to evaluate the tool's accuracy. In the sample, the tool has an accuracy of 100% on both tainted self-destruct and accessible self-destruct. A comparison is also made to two other tools, teEther, and Securify, which can detect similar things that Ethainter can. On a sample of 6 094 contracts, Ethainter has an accuracy of 69% across all bugs and of 73% when it comes to self-destruct. This shows potential for Ethainter to be the best available tool for finding reachable self-destruct, but since we cannot run it and the data sets used are not available, we cannot investigate this further. Compared to our numbers, where Slither achieved an accuracy of 52% and Mythril of only 16%. To keep in mind, is that Mythril has an accuracy of 61% on our benchmark dataset and the low percentage on the verified dataset was due to Mythril mislabeling one common type of contract many times. Slither had an accuracy of 56% on our benchmark dataset, which is more in line with the results on the entire verified dataset and Ethainter's results.



# Chapter 6

## Conclusion

---

Program analysis is a complex task, and it takes a lot to cover everything. In the case of Solidity, it is hard to tell if a smart contract has a security flaw or if it is a feature. From writing this thesis, we can conclude that manual review of smart contracts will be around for a while with the extensive range of different security vulnerabilities, where no tool can detect all vulnerabilities. Smart contract security tools have their different specialized strengths and, with them, their corresponding weaknesses.

The ecosystem of smart contracts on the Ethereum blockchain has quickly emerged as a large holder of monetary value. The potential financial gain has attracted many malicious users and people trying to develop serious, smart contract projects. As seen in 3.14, the number of vulnerable contracts has decreased as the space has matured. Even though the number of bugs in newly deployed contracts has decreased, a significant percentage of contracts are still vulnerable. The need for program analysis tools still exists to make sure smart contracts are developed without bugs. The tools analyzed in our thesis and T.Durieux Et al. (Section 5.1) find less than 50% of the relevant security threats. In the current state, manual review is still heavily relied upon to ensure contract safety.

Maian is a reasonably old tool that is no longer maintained, and in a fast-moving area such as smart contracts, it can no longer be used to find bugs. Maian finds the more reachable self-destructs than the other symbolic execution tool, Mythril, on the unverified smart contracts, as seen in section 4.3. These contracts do not have available source code, and it is hard to conclude the accuracy and precision of Maian. The fact that it finds more than Mythril might be because many of the contracts present in the unverified dataset are old, and Maian can handle them, which shows that it was state-of-the-art when developed; the lack of maintenance has prevented it from keeping this status.

## 6.1 Future work

### 6.1.1 Surveying more

To estimate what the current state of the art security tool, more tools would need to be included in the research. The tools in this paper were chosen based on which tools could find reachable self-destruct and because T. Diroux Et Al. [16] suggested the combination of Slither and Mythril as the most complete way of finding bugs while still limiting time and memory used. In particular, Ethainter should be included in a study that the author does not write of the tool. In the case of reachable self-destruct, this would be very interesting since the accuracy numbers claimed in the Ethainter paper, [10] are way above the accuracy of Slither, the most accurate tool we evaluated.

### 6.1.2 Implementing improvement of tools

This thesis studies three tools, out of which we consider two to be state-of-the-art, Mythril and Slither. However, to be state of the art for analyzing smart contracts does not require a high level of accuracy since the tools analyzed still have a high rate of false positives, as seen in figures 4.6 and 4.4. Section 4.7 identifies several improvements for Slither and Mythril that could improve the tool's detection rate of reachable self-destruct and other access control-related problems. If implemented, these improvements could push both tools into an accuracy that is above current standards.

Slither is a static analysis tool with a wide range of detection modules that is easy to expand upon. Restarting this project that could have been a project to implement taint analysis with the points-to analysis described in Chapter 4. Slither's specific "unprotected self-destruct" module is very primitive, as described in Chapter 4, but still quite effective. For example, it handles cross-contract vulnerabilities better than Mythril, where it reads the source code and checks for open self-destruct functions. It is simple yet effective. The trouble comes when you are aware of how the tool works. It is super easy to trick if you know what to do because of its hardcoded parts. For example, name the modifier `onlyOwner`, and your function will always pass as safe.

Mythril can generally find the most reachable self-destruct inside one smart contract; the problem appears when introducing recursion, multiple modifiers, or using logic from smart contracts instantiated inside another. Mythril is harder to improve for anyone because of the use of symbolic execution and the extra complexity it brings. Mythril needs better documentation to help with the process of improving the tool. Improving Mythril's ability to handle cross-contract analysis would remove 100% of its false positives because all of them were of the same type and were secured by values in another smart contract.

### 6.1.3 Detecting vulnerabilities using neural networks

In M.Rossinis's paper about using neural networks to find vulnerabilities in solidity code, it is deemed feasible to do so successfully.[30] However, her results are based on the results of Slither, meaning that it depends on the result of only one tool. Suppose a ground truth were to be set for vulnerabilities; a neural network model could be used with success. An

approach that also has been suggested is to use the neural network on the control flow graphs of a contract and train it to identify dangerous patterns. This we see as a potential way of identifying intended reachable self-destruct or other intended vulnerable smart contracts. As mentioned in Section 4.2.2, we do not see any straightforward ways to separate intended reachable self-destruct from unintended. However, with machine learning, this could be possible and, in this way, remove a lot of irrelevant (false positives).



# References

---

- [1] Etherlite whitepaper. <https://etherlite.org/assets/pdf/EtherLite-Whitepaper.pdf>.
- [2] Access control, openzeppelin. *Openzeppelin.com*, 2017.
- [3] Solidity documentation. <https://docs.soliditylang.org/en/v0.8.11/structure-of-a-contract.html?highlight=modifier%20functionsfunctions>, 2021.
- [4] Common weakness enumeration. <https://cwe.mitre.org/index.html>, 2023.
- [5] Ethereum whitepaper. <https://ethereum.org/en/whitepaper/>, 2023.
- [6] Smartbugs: A framework to analyze ethereum smart contracts. <https://github.com/smartbugs/smartbugs>, Mar 2023.
- [7] Jonathan Aldrich and Claire Le Goues. *Lecture Notes: Symbolic Execution 1 Symbolic Execution Overview*.
- [8] Edward Axlund and Frans Sjöström. Reachable self-destruct benchmark dataset, 2023.
- [9] Roberto Baldoni, Emilio Coppa, Camil Cono D’elia, Daniele Demetrescu, and Irene Finocchi. “a survey of symbolic execution techniques.”. In *ACM Comput. Surv.* 51, 3, 2018.
- [10] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 454–469, 2020.
- [11] Krishnendu Chatterjee, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. The treewidth of smart contracts. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC ’19*, page 400–408, New York, NY, USA, 2019. Association for Computing Machinery.
- [12] Jiachi Chen, Xin Xia, David Lo, and John Grundy. Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum. *ACM Trans. Softw. Eng. Methodol.*, 31(2), dec 2021.

- [13] ConsenSys. `mythril/suicide.py` at `develop` · `consensys/mythril`. <https://github.com/ConsenSys/mythril/blob/develop/mythril/analysis/module/modules/suicide.py>, 2023.
- [14] Phil Daian. Gastoken.io - cheaper ethereum transactions, today, 2017.
- [15] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, page 337–340, 2008.
- [16] Thomas Diurex, Joao Ferreira, Pedro Cruz, and Rui Abreu. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: 2020*, 2020.
- [17] Nik Roby and Karen Scarfone Dylan Yaga, Peter Mell. Blockchain technology overview. Oct 2018.
- [18] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2019.
- [19] Anjana Gosain and Ganga Sharma. Static analysis: A survey of techniques and tools. In *Intelligent Computing and Applications: Proceedings of the International Conference on ICA, 22-24 December 2014*, pages 581–591. Springer, 2015.
- [20] Stephen Graves. 13 biggest defi hacks and heists. <https://decrypt.co/93874/biggest-defi-hacks-heists>, Apr 2022.
- [21] Hao Guo, Ehsan Meamari, and Chien-Chung Shen. Multi-authority attribute-based access control with smart contract. In *Proceedings of the 2019 International Conference on Blockchain Technology, ICBCCT 2019, New York, NY, USA, 2019*. Association for Computing Machinery.
- [22] Alice Henshaw. Using ethereum’s create2. <https://hackernoon.com/using-ethereums-create2-nw2137q7>, Jan 2020.
- [23] Prateek Saxena Ivica Nikolic, Aashish Kolluri and Aquinas Hobor. `Maian`. <https://github.com/ivicanikolicsg/MAIAN/tree/master/tool>, 2023.
- [24] Daniel Jackson and Martin Rinard. Software analysis: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 133–145, 2000.
- [25] Satpal Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Ethereum smart contract analysis tools: A systematic review. *IEEE Access*, pages 1–1, 04 2022.
- [26] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimäki, Savanna Lujan, and Fabio Palomba. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software*, 198:111575–111575, Nov 2022.
- [27] Satoshi Nakamoto. Bitcoin whitepaper. URL: <https://bitcoin.org/bitcoin.pdf>-(: 17.07. 2019), 2008.



- [28] Ivica Nikolić, Aashish Kolluri, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. 2018.
- [29] Martina Rossini. Github - mwritescode/smart-contracts-vulnerabilities: [blocksys 2022] exploring deep learning techniques for ethereum smart contract vulnerability detection., Jun 2022.
- [30] Martina Rossini, Mirko Zichichi, and Stefano Ferretti. Smart contracts vulnerability classification through deep learning. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, SenSys '22, page 1229–1230, New York, NY, USA, 2023. Association for Computing Machinery.
- [31] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. Smart contract: Attacks and protections. *IEEE Access*, 8:24416–24427, 2020.
- [32] Nipun Sharma and Swati Sharma. A survey of mythril, a smart contract security analysis tool for evm bytecode, Dec 2022.
- [33] SmartContractSecurity. Github - smartcontractsecurity/swc-registry: Smart contract weakness classification and test cases. <https://github.com/SmartContractSecurity/SWC-registry/creating-a-new-swc-entry>, 2023.
- [34] Maddipati Varun, Balaji Palanisamy, and Shamik Sural. Mitigating frontrunning attacks in ethereum. BSCI '22, page 115–124, New York, NY, USA, 2022. Association for Computing Machinery.
- [35] Gavin Wood. Ethereum yellow paper. <https://ethereum.github.io/yellowpaper/paper.pdf>, 2022.