

# Automated Exploit Generation For Ethereum Smart Contracts

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering und Internet Computing**

eingereicht von

**Lukas Kösslbacher, BSc**

Matrikelnummer 01615231

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ass.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Monika di Angelo

Wien, 21. Juni 2023

---

Lukas Kösslbacher

---

Monika di Angelo



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Automated Exploit Generation For Ethereum Smart Contracts

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering and Internet Computing**

by

**Lukas Kösslbacher, BSc**

Registration Number 01615231

to the Faculty of Informatics

at the TU Wien

Advisor: Ass.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Monika di Angelo

Vienna, 21<sup>st</sup> June, 2023

---

Lukas Kösslbacher

---

Monika di Angelo



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Erklärung zur Verfassung der Arbeit

Lukas Kösslbacher, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. Juni 2023

---

Lukas Kösslbacher



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Acknowledgements

Thank you to everyone who feels addressed.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Kurzfassung

Mit dem Aufkommen der Ethereum-Blockchain im Juli 2015 wurden Smart Contracts dem Mainstream näher gebracht. Smart Contracts sind Programme, die die zugrunde liegende Blockchain als Ausführungsumgebung nutzen. Ein Hauptargument für sie ist, dass jede:r sie untersuchen und mit ihnen interagieren kann. Allerdings können sie, sobald sie auf der Ethereum-Blockchain installiert sind, nicht mehr verändert werden, was Fehlerbehebungen unmöglich macht.

Da Smart Contracts Milliarden von Euro in verschiedenen Vermögenswerten halten, bieten Exploits hohe Gewinne bei relativ geringen Risiken. Daher ist es wichtig, mögliche Schwachstellen in Smart Contracts zu erkennen, bevor sie eingesetzt werden. Schwachstellen in Smart Contracts erfordern oft bestimmte Laufzeitbedingungen oder sehr spezifische Transaktionssequenzen.

Diese Arbeit konzentriert sich auf Tools, die eine automatisierte Exploit-Generierung verwenden und versuchen, diese Probleme zu lösen. Die Arbeit untersucht zunächst bestehende Techniken zur Erkennung von Schwachstellen sowie zur automatischen Generierung von Exploits. Danach wird ein Überblick und ein theoretischer Vergleich bestehender Tools hinsichtlich der verwendeten Techniken und der Schwachstellen, die sie zu finden vorgeben, gegeben. Schließlich wird ein neuer Prototyp vorgestellt, der auf einem der zuvor untersuchten Tools (*Vultron*[WLL<sup>+</sup>20]) basiert. Der Prototyp wird dann anhand eines Benchmark-Sets getestet und mit *ConFuzzius*[TIGS20] und *sFuzz*[NPS<sup>+</sup>20] verglichen.

Obwohl alle Tools auf Fuzzing basieren, zeigt der Vergleich, dass der Prototyp nicht so gut abschneidet wie die beiden anderen Tools. Dennoch entdeckte der Prototyp Exploits in Smart Contracts, bei denen die anderen Tools nichts fanden, und bietet daher einige neue Erkenntnisse über gefundene Exploits. Um möglichst viele Schwachstellen abzudecken, reicht es außerdem nicht aus, nur ein einziges Tool zu verwenden, sondern vielmehr eine Kombination aus mehreren Tools, die unterschiedliche Analysetechniken implementieren.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Abstract

With the rise of the Ethereum blockchain in July 2015, smart contracts were brought closer to the mainstream. Smart contracts are programs that use the underlying blockchain as their execution environment. A major argument for them is that anyone can investigate and interact with them. However, once deployed to the Ethereum blockchain they cannot be altered anymore, making bugfixes impossible.

Since smart contracts hold billions of Euros in various assets, exploits offer high rewards with relatively small risks. Therefore, detecting possible vulnerabilities in smart contracts before deploying them is essential. Vulnerabilities in smart contracts often require certain run-time conditions or very specific transactions sequences.

This thesis focuses on tools that employ automated exploit generation which try to tackle these problems. The work first researches existing techniques for vulnerability detection as well as automated exploit generation. Afterwards, it gives an overview as well as a theoretical comparison of existing tools regarding the techniques used and which vulnerabilities they claim to find. Finally, the thesis introduces a new prototype based on one of the before-studied tools - *Vultron*[WLL<sup>+</sup>20]. The prototype is then tested against a benchmark set and compared to *ConFuzzius*[TIGS20] and *sFuzz*[NPS<sup>+</sup>20].

Although all tools are based on fuzzing, the comparison shows that the prototype does not perform as well as the other two tools. However, the prototype found exploits in smart contracts that the other tools did not find, and therefore provides some new insights into exploits found. Furthermore, to cover the most vulnerabilities possible it is not sufficient to use only a single tool, but rather a combination of multiple tools using different analysing techniques.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Contents

<b>Kurzfassung</b>	<b>ix</b>
<b>Abstract</b>	<b>xi</b>
<b>Contents</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Definition and Motivation . . . . .	1
1.2 Research Questions . . . . .	2
1.3 Methodological Approach . . . . .	2
<b>2 Fundamentals</b>	<b>5</b>
2.1 Blockchain . . . . .	5
2.2 Cryptocurrencies . . . . .	8
2.3 Smart contracts . . . . .	9
2.4 Ethereum . . . . .	10
<b>3 Definition of vulnerabilities in Ethereum smart contracts</b>	<b>21</b>
3.1 Weaknesses . . . . .	22
3.2 Vulnerabilities . . . . .	23
3.3 Exploits . . . . .	23
3.4 OP-Codes interesting for Smart Contract exploits . . . . .	24
<b>4 Techniques for Automated Exploit Generation</b>	<b>27</b>
4.1 Control Flow Graph (CFG) . . . . .	27
4.2 Symbolic Execution . . . . .	28
4.3 Code property graph (CPG) . . . . .	30
4.4 Fuzzing . . . . .	30
4.5 Program slicing . . . . .	31
4.6 Taint analysis . . . . .	32
<b>5 Related Work</b>	<b>33</b>
5.1 Maian . . . . .	33
5.2 Teether . . . . .	35
	xiii

5.3	Vultron (ContraMaster) . . . . .	37
5.4	EthPloit . . . . .	40
5.5	EthFuzz . . . . .	44
5.6	Comparision . . . . .	46
<b>6</b>	<b>Implementation of a Prototype</b>	<b>49</b>
6.1	Foundation for the prototype . . . . .	49
6.2	Design decisions and implementation details . . . . .	50
<b>7</b>	<b>Definition and evaluation of a Benchmarkset</b>	<b>61</b>
7.1	Need for a benchmark set . . . . .	61
7.2	Selection of contracts for the benchmark set . . . . .	61
7.3	Evaluation of the prototype as well as <i>ConFuzzius</i> and <i>sFuzz</i> on the benchmark set . . . . .	62
7.4	Interpretation of the evaluation results . . . . .	65
<b>8</b>	<b>Conclusio</b>	<b>69</b>
8.1	Summary & Conclusion . . . . .	69
8.2	Limitations & Future work . . . . .	70
<b>A</b>	<b>Voting Contract</b>	<b>71</b>
	<b>Listings</b>	<b>79</b>
	<b>List of Figures</b>	<b>80</b>
	<b>List of Tables</b>	<b>81</b>
	<b>Bibliography</b>	<b>83</b>

# Introduction

## 1.1 Problem Definition and Motivation

In 2008 a paper describing an electronic peer-to-peer cash system was published - Bitcoin. It described a transaction-centred model, where all transactions are recorded on a distributed public ledger - the blockchain[Nak09]. Bitcoin needs no third party to ensure "trust" in the introduced peer-to-peer network, instead, it uses a consensus algorithm that motivates the participants to be honest by awarding them Bitcoins after reaching a consensus. To generate these "new" bitcoins the participants have to solve a cryptographic hash function which is known as Proof-of-Work[CPNX20]. This was the beginning of cryptocurrencies. Since the emergence of Bitcoin, a lot has happened in the space of cryptocurrencies and their popularity is still increasing. Currently, there are around 8700 different cryptocurrencies or token projects and the overall market cap keeps rising. In the last year alone it rose from 769.418.164.617€(on 05.01.2021) to 1.985.313.731.461€(on 04.01.2022)[coi]. One of the largest cryptocurrencies by market value went live in July 2015 and is called Ethereum[eth]. Unlike Bitcoin, Ethereum is based on accounts. Every account consists of:

- a counter to ensure transactions get processed only once (nonce)
- a balance of Ether
- the contract storage
- an optional contract code

There are two types of accounts in Ethereum, externally owned accounts (private accounts) and contract accounts (smart contracts). Externally owned accounts are controlled by

their private keys, while contract accounts are controlled by their contract code. To pay the costs involved in making transactions Ether is used.

Ethereum also introduced a touring complete programming language to develop smart contracts[But14]. Smart contracts hold billions of Euros in various assets and exploits offer high rewards with relatively small risks. For example, the DAO-FORK in July 2016 was the result of an insecure contract that was drained of 3.6 million Ether[eth]. However, the creation of secure smart contracts is a very complicated task and since they are immutable, this gets even harder. Once a contract code is deployed on the Ethereum chain it can not be changed again. On the one hand, this ensures nobody can alter deployed code, but on the other hand, it makes the patching of identified vulnerabilities impossible[Ash21]. Considering this, detecting possible vulnerabilities in smart contracts is all the more important. There are currently a lot of tools available that analyse the underlying code of smart contracts [SSD<sup>+</sup>21, DN18, FTB19, TIGS21], but they essentially all have the same problem. Exploits in smart contracts often depend on specific transactions-sequences or are only exploitable if certain run-time conditions are met. Therefore the existing tools often detect false-positive or miss false-negative vulnerabilities. This is where automated exploit generation comes into play. The goal of automated exploit generation is to identify possible exploitable vulnerabilities and to create a set of transactions or a sample code which directly exploits found vulnerabilities. This gives developers of smart contracts the possibility to verify the results of static analysis and test their contracts before deploying them.

### 1.2 Research Questions

The thesis addresses the following research questions:

1. Which open source tools are currently available for automated exploit generation of Ethereum smart contracts?
2. What are the methods used to detect exploitable vulnerabilities and generate the actual exploit?
3. How well do the selected tools work?

### 1.3 Methodological Approach

The methodological approach in this thesis consists of the following steps:

1. **Semi-Systematic Literature Review:**

A semi-systematic literature review based on [Sny19] to get an overview of current trends in smart contract development, known vulnerabilities of smart contracts and the current state of the art regarding automated exploit generation tools.

Furthermore, the review is used to gather background knowledge - on the Ethereum network, the Ethereum Virtual Machine (EVM) <sup>1</sup> and Solidity <sup>2</sup> - needed for the evaluation of currently available tools and the possible implementation of a new prototype.

## 2. Theoretical Comparison of existing tools:

Although there exist lots of analysis tools for smart contracts only a few of them also offer automatic exploit generation. This thesis will first summarise the existing tools and afterwards compare their approach focusing on:

- a) Analysis techniques used
- b) Vulnerabilities they claim to find

## 3. Prototyping:

Development of a tool to automatically generate exploits for Ethereum smart contracts based on the findings of the conducted literature review.

## 4. Comparative Analysis:

A comparative analysis of different tools against a provided benchmark set<sup>3</sup> will be conducted. It will focus on:

- a) Number of failed contracts
- b) Number of contracts containing an exploit
- c) Number of total generated exploits

---

<sup>1</sup><https://ethereum.org/en/developers/docs/evm/> - The environment in which all Ethereum accounts and smart contracts live. It defines the rules for computing a new valid state for new blocks.

<sup>2</sup><https://soliditylang.org/> - Solidity is an object-oriented, high-level language for implementing smart contracts.

<sup>3</sup>The benchmark set will be provided by the supervisor of this work: *Ass.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Monika di Angelo* (for more information see chapter 7).



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Fundamentals

A short introduction to blockchain and cryptocurrency technologies important for this work is given in this chapter. Since discussing technical implementation details would be beyond the scope of this work, only functional aspects necessary to understand the following thesis are described. In-depth and more technical analysis will be conducted in separate chapters where needed, or sources that offer more comprehensive information are cited.

## 2.1 Blockchain

Transferring assets between two parties often involves some legal contract that has to be certified by a third party. Most of the time, this is a long, tedious and quite expensive process, after which the ownership is officially moved from one party to the other. However, to verify its legitimacy, one must examine some previous transactions related to the transferred asset. Depending on the domain, such a transaction history is kept by a third party (e.g. financial institute, government) in a centralised place (also called a ledger). For this system to work, every party involved in the transaction has to trust the third party and even then, it is impossible to prevent deceitful behaviour completely.[DP17] Furthermore, even if a ledger exists for a specific asset, it is not always feasible to trace the current owner of it. This becomes even more of an issue if the asset in question changes ownership often and on a global scale. For example, in 2008, JP Morgan Chase acquired the US investment bank Bear Stearns, but due to an accounting error, JP Morgan Chase offered the acquirer a larger number of shares than what were outstanding in the books of Bear and Stearns.[NGHS17]

In 2008 Satoshi Nakamoto published a paper which introduced a peer-to-peer payment system. This was the birth hour of a technology we now know as “blockchain”. The intention behind the proposed technology was, amongst others, to solve the above-mentioned problem of trust occurring during an interaction between different parties. To

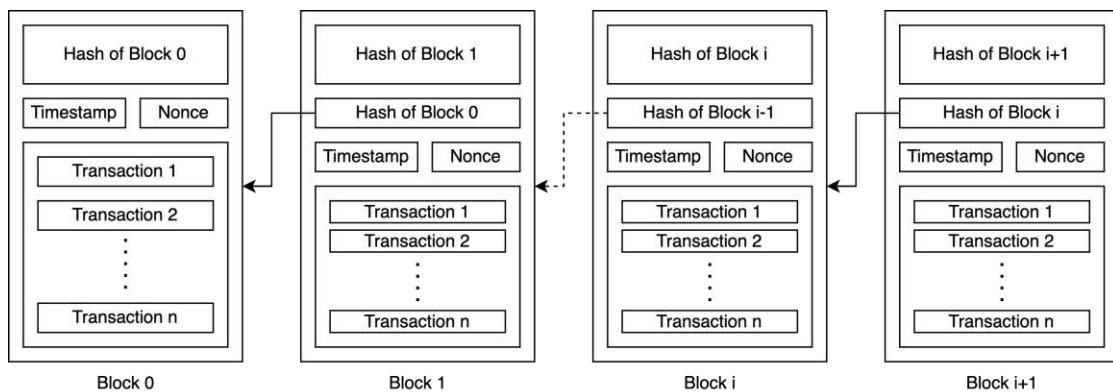


Figure 2.1: Basic schema of a “Blockchain” drawn after [Nak09].

overcome the issues, Nakamoto proposed a system that is based on cryptographic proof instead of trust. Figure 2.1 shows the basic schema of a blockchain.

A bitcoin-like blockchain behaves typically like a linked list where each new block points at its predecessor. Each block includes a certain amount of transactions, a timestamp when it was created, the hash of the previous block, a nonce (= random number needed to compute the hash of the block) and its own hash that is created by hashing all previously mentioned fields.[NGHS17] Every party (= node) that wants to participate stores this chain of data blocks. Since it behaves like a linked list, every transaction ever completed is publicly available for all nodes. For a new block to be created, it has to be validated by the network. This happens by checking the correctness of each transaction inside the block - ensuring that involved assets have not been transferred in previous transactions already - and solving a computationally complex problem. After checking the correctness and solving the problem, the new block is propagated to the network, and a consensus between all nodes is established. The new block gets appended to the blockchain. and the process of gathering transactions and validating them starts again.[YHKC<sup>+</sup>16]

Bitcoin, the most prominent technology based on a blockchain, for example, uses “Proof-of-Work” as consensus mechanism, which relies on computing power to find the hash of a new block. The computed hash value of the block has to start with a certain number of zeros. This requirement can only be brute forced. It means that a mining node has to change the block values (e.g. by incrementing the nonce) until the requirement is met. However, today the nonce is exhausted after a few seconds. Hence other values of the block have to be changed as well. The average work required for computing this hash grows exponentially with the number of zero bits, while it can be verified by simply executing a single hash. Since new blocks always contain the previous hash value (see fig. 2.1), changing a transaction already validated would require the attacker not only to redo the targeted block but also to redo all subsequent blocks. Therefore as long as the majority of computing power is possessed by honest nodes, once validated, blocks can not be tampered with.[Nak09]

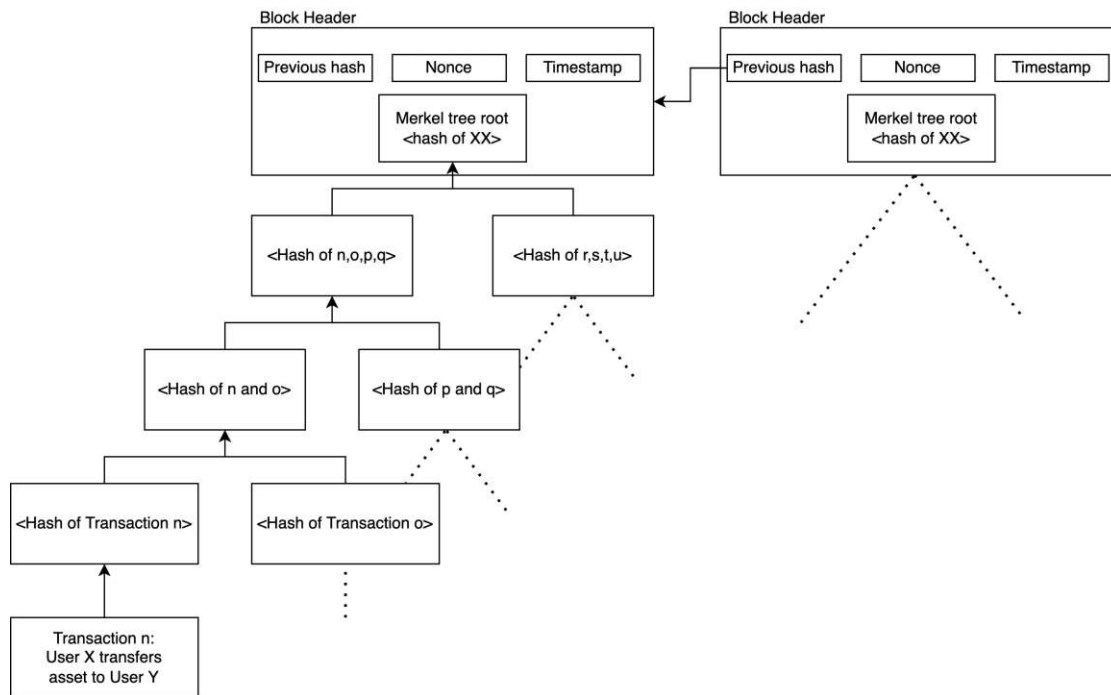


Figure 2.2: Showing the schema of a Merkle tree inspired by [Nak09] and [But14]. Each block header stores only the root hash of the underlying Merkle tree. Modifying something down the tree will change all superordinate hashes and therefore invalidate the whole tree and block.

This process of creating and validating new blocks is called “mining”. To create interest in the network and motivate the participants to stay honest, an incentive can be given to whichever node proposes the new and valid block fastest. Due to the nature of blockchains, it is not even necessary for every node to keep a copy of the complete history. As long as few nodes do, the network can carry on, and others can join and participate. This is possible because everyone knows the last hash and can verify that nothing has been altered since this would lead to a different and invalid hash.

Storing every block in its entirety (= “full node”) takes up a lot of storage space - as of March 2022, approximately 400 GB<sup>1</sup>. This makes it quite impracticable for occasional users to participate. To conquer the topic of storage size, Satoshi proposed the following solution: The hash of each block is, in fact, only a hash of the blocks header, containing the timestamp, its nonce, the previous hash and the root hash of the blocks Merkle tree (see fig. 2.2). A Merkle tree is a type of binary tree - meaning every node has a maximum of two child nodes. At the bottom of the tree is a large number of leaf nodes containing the actual transaction data. All nodes other than the leaf nodes are only hashes of their children. On top of the tree sits the root node (= also the hash of its two children).

<sup>1</sup>[https://ycharts.com/indicators/bitcoin\\_blockchain\\_size](https://ycharts.com/indicators/bitcoin_blockchain_size) - accessed on 24.03.2022

This setup allows it to deliver data in pieces and from different sources without losing its integrity. Since hashes propagate upward, an attacker trying to change a transaction inside a leaf would cause a change of the node above, which would again change the hash of its parent up to the hash of the root changes. When the root changes, the block hash also changes, which would lead to a new and possibly invalid block[But14].

Due to the introduction of a Merkle tree for storing transaction data, nodes can participate without running a “full node”. So-called “light nodes” only keep a copy of all block headers and can fetch the branch of the Merkle tree corresponding to transactions of interest. Although the transaction itself can not be verified, by linking it to the chain, it can be shown that the network has accepted it. Blocks added to the chain afterwards further confirm the validity of earlier transactions.[Nak09].

### 2.2 Cryptocurrencies

Cryptocurrencies are mediums of exchange, like “money”, that are used to transfer financial assets over the internet. But unlike the Euro or the U.S. Dollar, cryptocurrencies do not rely on central authorities such as governments to maintain their value and verify transactions.

In 1983 David Chaum suggested his concept of “blind signatures”[Cha83]. This was the first advance for payment methods to use cryptography. With blind signatures, Chaum designed a system to keep anonymity and prevent/minimise double-spending. The functionality can be described as follows:

- Person A creates a note saying: “The person giving me back this card gets one dollar.”. Person A then hands this card out to person B.
- Person B thinks about a serial number and writes it on the card.
- Person A now signs the card without seeing the serial number written on it.

Since every serial number can only be redeemed once, person B wants to pick a long and random one that will most likely be unique. Person A does not have to worry about person B picking a serial number that was already picked. It would only mean that person B would not be able to redeem the note. This was the first serious digital cash proposal, but it still needs a centralised trusted entity. New payments/transactions cannot be processed if the server is down.[NBF<sup>+</sup>16][Cha83]

Until the advent of Bitcoin, there were several tries to bring electronic cash to mainstream. Especially throughout the 1990s several companies tried and failed.[BMC<sup>+</sup>15]

With an overall market cap of approximately 1.9 trillion Euro[glo], cryptocurrencies have established themselves by now. Next to the big players like Bitcoin and Ethereum, there exist over 9000 different cryptocurrencies at the moment.[glo] What most of

them share are three common principles: decentralisation, anonymity and transparency. Decentralisation and transparency are achieved because they run on blockchains (see 2.1). Since cryptocurrencies do not use usernames or user-ids but public and private keys to identify users, they are considered to be at least pseudo-anonymous.[TKA<sup>+</sup>22]

Cryptocurrencies on a blockchain come in two different forms. Coins and tokens. A lot of times, these two terms are used as synonyms while, in fact, they are two different things with different purposes.

When talking about a coin, we usually refer to the native currency of a blockchain. Examples of coins are Bitcoin (BTC) for the Bitcoin project and Ether (ETH) for Ethereum. Coins are generally used like traditional money. For example, BTC can be used to buy goods and services over the internet and in some real-life places. You can also just store it for an indefinite time and exchange it later for something of equal value. BTC has no other uses; it is so to say, just digital money. However, other cryptocurrencies can offer more features like ETH. It can be used as money, but it is also necessary to cover transaction costs and deploying new applications to the Ethereum network.[M21]

Opposed to coins, tokens are created on existing blockchains. They are mostly created and administered by smart contracts, which will be briefly described in 2.3. For the creation of tokens, the native currency of the blockchain has to be traded in. The most popular blockchain platform implementing token protocols is Ethereum.[M21] The applications of token systems are diverse and range from forming sub-currencies, representing smart property to unforgeable coupons.[But14] The two main categories of tokens are: fungible (FT) and non-fungible tokens (NFT). FTs are interchangeable - with other units of the same token - and divisible, whereas NFTs are unique - they cannot be changed with other units - and are indivisible.[KCR21] Ethereum uses “Ethereum Improvement Proposals (EIP)” based on “Ethereum Request for Comments (ERC)” to standardise tokens. With over 500,000<sup>2</sup>token contracts on the Ethereum network, the best known and used token standard by the time of writing is the fungible ERC-20[BV15] token.

## 2.3 Smart contracts

The idea of “smart contracts” was first introduced in the 1990s by Nick Szabo[Sza96]. He described them as a set of promises, specified in digital form, that automatically gets executed when predefined conditions are met. Since conventional contracts often rely on trusted third parties, smart contracts offer a reduced execution time by trying to eliminate the need of these third parties. Due to the lack of trusted execution environments at the time of the proposal, the idea of smart contracts stayed theoretically[WSX<sup>+</sup>21]. The emergence of blockchains (see 2.1) changed this.

Generally speaking, smart contracts are programs that use the underlying blockchain as their execution environment. For example, a supply-chain between two parties could work as follows (see also figure 2.3): The supplier deploys a smart contract (supplier

<sup>2</sup><https://etherscan.io/tokens> - accessed on 04.04.2022

contract) where potential buyers can first get and browse through the offered products and afterwards can submit an order containing the desired product. After receiving the order, the deployed smart contract can start the search for a carrier. This could work following the same principles. Different carriers publish their shipping information (price, shipping time, etc.) as smart contracts on the blockchain. The supplier contract can then choose the best-suited carrier. After choosing a carrier the contract sits idle until the ordered product reaches its destination. As soon as this happens and the buyer confirms the integrity of the product the smart contract triggers the payment process, where the buyer pays the supplier and the supplier pays the carrier. All payments are conducted in cryptocurrencies and every action taken during the whole supply-chain process is publicly available on the blockchain.

Taking the mentioned scenario, smart contracts offer several advantages over conventional contracts:

- Reducing administration time and service cost: Since blockchains create trust between mutually distrusting parties, no third party is needed and due to the distributed environment smart contracts can be triggered from anywhere at anytime.
- Reducing risk of fraud: Once deployed on a blockchain the smart contract becomes immutable and can not be altered any more. Furthermore, every execution of a smart contract statement is also recorded as an immutable transaction and is publicly available for every participant.[ZXD<sup>+</sup>20]

A smart contract is normally written in a high-level Turing complete programming language and compiled to bytecode. It gets stored on the blockchain and comprises a unique address, a persistent storage private to the contract, a balance of the underlying cryptocurrency (see 2.2) and some executable code to perform computations and manipulate its storage or balance.[FTB19]

### 2.4 Ethereum

With a market cap of around 360 billion Euro<sup>3</sup>Ethereum is the second largest cryptocurrency. The intent behind Ethereum was to create a blockchain that incorporates a Turing complete programming language, making it possible for everyone to write smart contracts and decentralised applications[But14].

Ethereum achieves this by keeping important information in its “world state”. The world state maps the identifiers of accounts (addresses) to their corresponding state[W<sup>+</sup>14]. This will be discussed further in 2.4.1. As mentioned, Ethereum uses accounts as basic building blocks. There are two types of accounts: externally owned accounts (private accounts) and contract accounts (smart contracts). Every account consists of:

---

<sup>3</sup><https://coinmarketcap.com/> - accessed on 08.04.2022

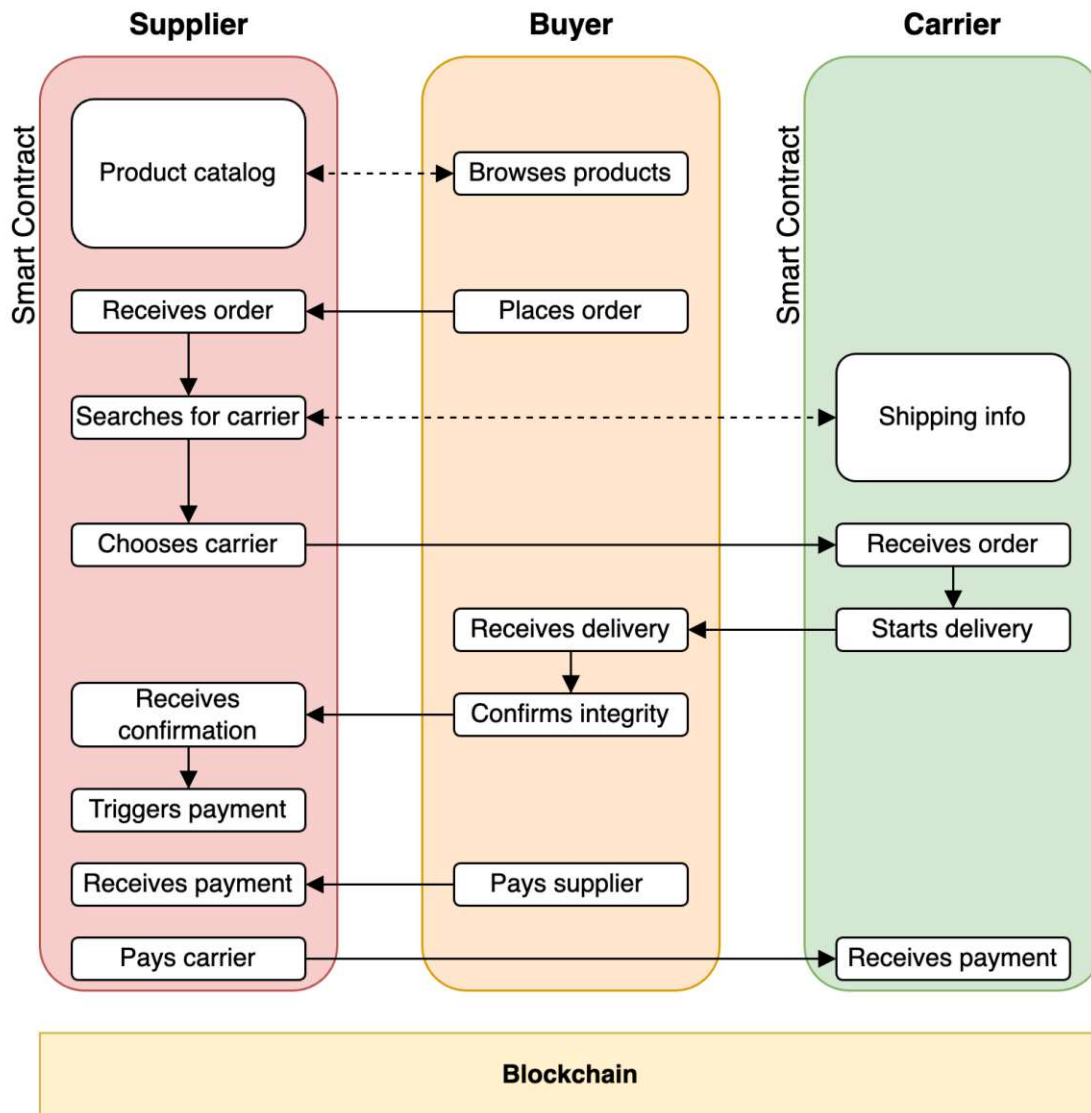


Figure 2.3: Representation of a supply chain using smart contracts. Once the buyer has issued an order, interaction and payment between the different parties works completely automatic and without intervention of intermediaries

- a counter to ensure transactions get processed only once (nonce)
- the current balance of the account, given in *wei*<sup>4</sup>
- a persistent contract storage

Additionally to these, a contract account also includes the underlying contract code. While externally owned accounts are controlled by their private keys, contract accounts are controlled by their code.

In Ethereum the world state can be altered by transactions and messages, which are signed packages storing data to be sent over the blockchain. Transactions are sent by private accounts and messages are created by contracts. While calls to externally owned accounts can only transfer ether, calls to contracts also lead to them executing their code.

One important feature about Ethereum is its model against denial-of-service attacks<sup>5</sup>.

To prevent infinite loops, every computational step comes with a cost that has to be paid by the transaction sender. Computation costs are denoted in “gas”[But14]. Every computational operation has its specific cost described in [W<sup>+</sup>14]. With this system an attacker would need to pay for every action he executes, making it infeasible/impossible to shut down the Ethereum network.

### 2.4.1 World State

As hinted above, Ethereum can be viewed as a transaction based state machine. To achieve this behaviour the network has to keep some sort of mapping between transactions and account states. In Ethereum, unlike Bitcoin, every block header contains not one, but three trees storing:

- the transactions within the block
- receipts of transactions, representing the effects of them
- the state of all accounts

This makes it possible to easily answer questions like: “What is the balance of my account?”, “Does this transaction exist in a specific block?”, “What would happen if this transaction is simulated on this contract?” and many more.

While binary Merkle trees (like the ones used by Bitcoin) are good to store “lists” of transactions and data that is not modified any more once it has been created, they are not well suited to store the state of accounts. The state tree in Ethereum uses key-value

---

<sup>4</sup>Wei is the smallest subunit of *Ether*. One *Ether* is equal to  $10^{18}$  wei.

<sup>5</sup><https://www.cisa.gov/uscert/ncas/tips/ST04-015> offers a description on denial-of-service attacks

mapping, where the key is an address and the value is an “object” containing its nonce, its balance, the code and the root hash of its own storage tree. Furthermore, the state gets changed frequently. Balances are modified very often and every transaction changes the nonce of an account. New accounts are inserted, keys get inserted and deleted. Therefore, a data-structure where quickly calculating the new tree root, without recomputing the whole tree, is needed. To achieve all of the above mentioned, “Merkle-Patricia-trees<sup>6</sup>” turned out to be the best suiting data-structure[?].

### 2.4.2 Ethereum Virtual Machine

While the world state (see 2.4.1) captures the current status-information of accounts, the “Ethereum Virtual Machine (EVM)” specifies how said state can be altered.

Alterations are represented by transactions and a valid state transition is formally described as:

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T) \quad (2.1)$$

where  $\Upsilon$  is the Ethereum state transition function,  $T$  a transaction and  $\sigma$  the current state of the system. The EVM uses a stack-based architecture with a maximum stack size of 1024 items. The utilised memory model is a volatile word-addressed byte array with a word size of 256 bit - meaning it consists of a series of bytes (words), where each word represents a different operation. Additional to the stack, the EVM also implements a persistent independent storage that is represented by a word-addressable word array and stored as part of the system state.

Due to the limitation of the maximum computational steps through gas, the described machine is *quasi*-Turing-complete[W<sup>+</sup>14].

### 2.4.3 Solidity

Although the EVM uses byte-code instructions, smart contracts are normally written in high-level programming languages. One of the most popular high-level languages is “Solidity”<sup>7 8</sup>. It is object-oriented, statically typed, supports libraries, inheritance and custom types. The language is heavily influenced by “C++”<sup>9</sup> and lightly influenced by “JavaScript”<sup>10</sup> and “Python”<sup>11</sup>.

<sup>6</sup><https://eth.wiki/en/fundamentals/patricia-tree> offers an in-depth description on Merkle-Patricia-trees

<sup>7</sup><https://ethereum.org/en/developers/docs/programming-languages/javascript/> - accessed on 11.04.2022

<sup>8</sup><https://soliditylang.org/>

<sup>9</sup><https://isocpp.org/>

<sup>10</sup><https://www.javascript.com/>

<sup>11</sup><https://www.python.org/>

Solidity enables developers to write smart contracts without facing a steep learning curve. It uses the concept of classes to represent contracts, which can implement private and public fields as well as methods. Solidity offers “function modifiers” that let developers modify the behaviour of functions (e.g. check preconditions or locking functions during execution to avoid re-entrancy attacks). Additionally to its standard types, Solidity offers globally available functions and fields that offer information about the blockchain. Two often used examples of these fields are: `msg.sender` for accessing the address of the account calling the method and `block.timestamp` to get the current block timestamp.

Like Python, Solidity offers multiple inheritance and polymorphism. An important fact about inheritance is that only a single contract is created on the blockchain, compiling all base contracts into one.

Interaction with other contracts is also possible but has to be done with caution, since every interaction imposes potential danger, especially if the source code of the contract is not known in advance.

Listing 2.1 shows a simple smart contract written in Solidity. Due to the rapid speed with which new releases are published it is important to always use the latest version.[Fou16b]

```
1 // SPDX-License-Identifier: GPL-3.0
2 pragma solidity >=0.4.16 <0.9.0;
3
4 contract SimpleStorage {
5     uint storedData;
6
7     function set(uint x) public {
8         storedData = x;
9     }
10
11    function get() public view returns (uint) {
12        return storedData;
13    }
14 }
```

Listing 2.1: Example of a simple Solidity contract that offers the functionality to store an integer-value[Pro].

### 2.4.4 Smart contracts in Ethereum

This section will describe smart contracts in Ethereum on the basis of a contract written in Solidity. The used contract implements a simple voting process, where either the creator of the contract can give others the right to vote or interested parties can pay for their right to vote. In this section only excerpts of it are used. The whole contract can be viewed in A.

Listing 2.2 shows a few important points of Ethereum/Solidity smart contracts. Before the actual implementation of the contract starts, so called `pragmas` can be defined. `Pragmas` are used to activate different compiler features and checks.

```

1 {
2   pragma solidity >=0.7.0 <0.9.0;
3   /**
4    * @title SimpleVote
5    * @dev Implements voting process along with vote delegation
6    */
7   contract SimpleVote {
8
9     struct Voter {
10      uint weight; // weight is accumulated by delegation
11      bool voted; // if true, that person already voted
12      address delegate; // person delegated to
13      uint vote; // index of the voted proposal
14    }
15
16    address public chairperson;
17    mapping(address => Voter) public voters;
18    Proposal[] public proposals;
19    ...
20 }

```

Listing 2.2: This listing shows how the Solidity pragma is defined and how custom types (structs) can be created[Fou21].

In this case only the `pragma_solidity` is defined. It ensures, that a contract only compiles with compiler versions that match the pragma and should therefore always be defined. The syntax follows the semantic versioning used by npm<sup>12</sup>. This means the pragma  $\geq 0.7.0 > 0.9.0$  tells the user, that the contract only compiles with compiler versions that are equal or greater than 0.7.0 and smaller than 0.9.0.

Next, the different state variables of a contract are defined. This contract does not only use pre-defined types but also a custom defined struct (`Voter`) is created. A custom defined struct can contain all pre-defined types, as well as other custom defined structs, but can not contain fields of its own type.

```

33 {
34   ...
35
36   constructor(bytes32[] memory proposalNames) {
37     chairperson = msg.sender;
38     voters[chairperson].weight = 1;
39
40     for (uint i = 0; i < proposalNames.length; i++) {
41       // 'Proposal({...})' creates a temporary
42       // Proposal object and 'proposals.push(...)'
43       // appends it to the end of 'proposals'.
44       proposals.push(Proposal({
45         name: proposalNames[i],
46         voteCount: 0

```

<sup>12</sup><https://docs.npmjs.com/cli/v6/using-npm/semver> - offers a in-depth description about semantic versioning used by npm

```

47     }));
48     }
49     }
50
51     ...
52 }

```

Listing 2.3: Example of how the constructor in Solidity is often used to define the owner of a contract and to initialise state variables[Fou21].

Listing 2.3 shows a common approach within smart contracts. The constructor is only called once during the creation of a contract and therefore it is often used to specify the calling account as its “owner”. In the example listed above this is done in line 35 where `chairperson` is set to `msg.owner`, which is the address of the account creating the contract. Furthermore line 36 shows how a new entry in a mapping (`voters`) can be created.

As mentioned in section 2.4.3, Solidity offers the possibility to create function modifiers. They can be used to check pre-conditions or implement arbitrary custom logic. In Solidity a function modifier is always executed before the actual function and has to explicitly return to the main function. This is achieved by the `_` operator which is substituted with the function body during runtime. The `_` operator can be used zero to multiple times within a modifier.

Listing 2.4 shows how a modifier can be defined and how it can be used by a function. In this case the modifier checks if the account calling the function has the same address as the `chairperson` field. If the addresses do not match the `require` statement will fail and the transaction will be reverted.

The `giveRightToVote` function makes use of the before defined `isChairperson` modifier. The function also shows that `require` statements do not have to be defined within a modifier. They can also be defined inside the main function body to check certain conditions.

```

49     ...
50
51     modifier isChairperson () {
52         require(
53             msg.sender == chairperson, ]
54             "Only chairperson can give right to vote."
55         );
56         _;
57     }
58
59     /**
60     * @dev Give 'voter' the right to vote on this ballot. May only be called by
61     *       'chairperson'.
62     * @param voter address of voter
63     */
64     function giveRightToVote(address voter) public isChairperson {

```

```

64     require(
65         !voters[voter].voted,
66         "The voter already voted."
67     );
68     require(voters[voter].weight == 0);
69     voters[voter].weight = 1;
70 }
71 ...
72 }

```

Listing 2.4: This listing shows how modifiers are defined and how they can be applied[Fou21].

A Solidity smart contract can only receive Ether in five ways:

- if it is the target of a `selfdestruct` call
- if it is the target of a *coinbase transaction* (=mining block reward)
- if it contains functions that implement the `payable` modifier
- Ether was transferred to the address before a contract was deployed there
- if multiple deployments - through `create2` - have happen at the same address

Listing 2.5 shows how the `payable` modifier can be used to receive Ether. In this specific case an account that did not get a “voting permission” from the chairman, can “buy” a vote if it sends at least one Ether to the contract (see line 106). If Ether is sent to a function that does not implement the `payable` modifier, the transaction gets automatically reverted and the sent Ether is transferred back to the sender[Fou16b].

```

96 {
97     ...
98
99     /**
100     * @dev Give your vote (including votes delegated to you) to proposal '
101         proposals[proposal].name'.
102     * @param proposal index of proposal in the proposals array
103     */
104     function vote(uint proposal) public payable {
105         Voter storage sender = voters[msg.sender];
106         require(
107             sender.weight != 0 || msg.value >= 1 ether,
108             "Has no right to vote"
109         );
110         require(!sender.voted, "Already voted.");
111         sender.voted = true;
112         sender.vote = proposal;
113
114         // If 'proposal' is out of the range of the array,
115         // this will throw automatically and revert all

```

```
115     // changes.
116     proposals[proposal].voteCount += sender.weight;
117 }
118
119 ...
120 }
```

Listing 2.5: Example of how a voting function - that accepts votes from chosen voters as well as "bought" votes - can be implemented[Fou21].

### 2.4.5 Application Binary Interface

The standard way to interact with a deployed smart contract is through its “Application Binary Interface (ABI)”. The ABI contains an entry for every function, error and event defined inside a smart contract. The single entries are encoded using a particular schema described in [Fou16a].

To make the communication with smart contracts through client-libraries easier, the JSON-format of the ABI can be passed to them. The JSON-ABI is an array of function, event and error descriptions. Since events and errors are not that important when it comes to exploits, this section will only list the fields of a function description.

- type: depending on the function either function, constructor, receive or fallback
- name: the name of the function
- inputs: an array containing a description of every function-parameter consisting of:
  - name: the parameter name
  - type: the canonical type of the parameter
  - components: optional array that is used to describe custom types
- outputs: an array with the same format as inputs, containing information about the return types
- stateMutability: defines how the function can interact with the blockchain state. Possible values:
  - pure - can not read state
  - view - can read but not modify state
  - nonpayable - can modify state but not receive Ether
  - payable - can modify state and receive Ether [Fou16a]

Listing 2.6 shows an excerpt of the ABI created from the voting contract used in section 2.4.4. For the sake of completeness the complete ABI can be viewed in Appendix A

```

1  [
2  {
3    "inputs": [
4      {
5        "internalType": "bytes32[]",
6        "name": "proposalNames",
7        "type": "bytes32[]"
8      }
9    ],
10   "stateMutability": "nonpayable",
11   "type": "constructor"
12 },
13 {
14   "inputs": [
15     {
16       "internalType": "address",
17       "name": "to",
18       "type": "address"
19     }
20   ],
21   "name": "delegate",
22   "outputs": [],
23   "stateMutability": "nonpayable",
24   "type": "function"
25 },
26 {
27   "inputs": [
28     {
29       "internalType": "uint256",
30       "name": "",
31       "type": "uint256"
32     }
33   ],
34   "name": "proposals",
35   "outputs": [
36     {
37       "internalType": "bytes32",
38       "name": "name",
39       "type": "bytes32"
40     },
41     {
42       "internalType": "uint256",
43       "name": "voteCount",
44       "type": "uint256"
45     }
46   ],
47   "stateMutability": "view",
48   "type": "function"
49 },
50 {
51   "inputs": [
52     {
53       "internalType": "uint256",

```

## 2. FUNDAMENTALS

---

```
54     "name": "proposal",
55     "type": "uint256"
56   }
57 ],
58   "name": "vote",
59   "outputs": [],
60   "stateMutability": "payable",
61   "type": "function"
62 },
63 ...
64 ]
```

Listing 2.6: Excerpt of the Application Binary Interface belonging to the voting contract used in 2.4.4

# Definition of vulnerabilities in Ethereum smart contracts

The *Common Weakness Enumeration (CWE)*<sup>1</sup> defines a vulnerability as:

A flaw in a software, firmware, hardware, or service component resulting from a weakness that can be exploited, causing a negative impact to the confidentiality, integrity, or availability of an impacted component or components.[Enuc]

Rainer Böhme, on the other hand, describes vulnerabilities in his paper as follows:

Vulnerabilities are errors in computer systems which can be exploited to breach security mechanisms.[Böh06]

A third category will also be addressed in this work's context - weaknesses:

A condition in a software, firmware, hardware, or service component that, under certain circumstances, could contribute to the introduction of vulnerabilities. [Enud]

The *Smart Contract Weakness Classification (SWC)*<sup>2</sup> defines a well-respected list of vulnerabilities and weaknesses within the community.

---

<sup>1</sup><https://cwe.mitre.org/index.html>

<sup>2</sup><https://swcregistry.io/>

### 3.1 Weaknesses

In the context of this work, parts of smart contracts that are considered bad practices (e.g. minder user experience, lead to bad performance and more) but can not be exploited are considered weaknesses.

Weaknesses in a smart contract can surface in different ways. A weakness can be so-called “irrelevant code” - code that is not essential for execution[Enua][Clab]. Not adhering to given or proven coding standards can also result in weaknesses within a contract[Enub]. However, in smart contracts, it can also mean contracts which are not programmed efficiently enough and produce higher gas costs than necessary.

Another aspect is the intended behaviour of a smart contract. A contract can contain vulnerable parts which still would only count as a weakness as long as it is the intended behaviour of the contract. Listing 3.1 for example shows such a contract.

```

1 // SPDX-License-Identifier: GPL-3.0
2 pragma solidity >=0.7.0 <0.9.0;
3
4 contract BuyToOwn {
5
6     address private owner;
7     bool public sold;
8     uint public priceToPay;
9
10    constructor(uint amount) {
11        require(amount > 1 ether, 'Amount must be bigger than 1 Ether');
12
13        owner = msg.sender;
14        sold = false;
15        priceToPay = amount;
16    }
17
18    function buyContract() external payable {
19        require(msg.value >= priceToPay, 'Not enough funds to pay price of
20            contract');
21        require(!sold, 'Contract was already sold.');
```

Listing 3.1: Shows a contract where a external party can trigger a selfdestruct after it pays a certain amount. Since this is the intended behaviour of the contract it is no vulnerability.

## 3.2 Vulnerabilities

From the aforementioned definitions of vulnerabilities follows, vulnerabilities lead to unintended behaviour that theoretically can be exploited.

Since the underlying code of smart contracts can not be altered once deployed, it is all the more important to prevent them. Unintended behaviour can range from locked or stolen resources to loss of data or integrity[Ram21].

While vulnerabilities imply that a contract is theoretically exploitable, it does not mean that a contract has already been exploited or can be exploited in a real-life scenario. For example, an attacker must be the contract owner to exploit it[PL21].

Another example of a vulnerability would be the smart contract shown in listing 3.2. The intended behaviour of *line 10* is to update the balance of the function caller according to the sent Ether. Instead, the contract only “checks” if the current balance is the same as the sent amount because of the double equal sign. Since the result of this check is not captured, the function continues and exits without error and the sent Ether is deposited into the contract. This could lead to a deviation between the stored balances and the deposited ether, which could theoretically lead to an exploit.

```

1  pragma solidity ^0.5.0;
2
3  contract DepositBox {
4      mapping(address => uint) balance;
5
6      // Accept deposit
7      function deposit(uint amount) public payable {
8          require(msg.value == amount, 'incorrect amount');
9          // Should update user balance
10         balance[msg.sender] == amount;
11     }
12 }

```

Listing 3.2: Smart contract snippet that shows code that has no effect, but leads to unintended behaviour[Clac].

## 3.3 Exploits

This work uses the following definition for exploits/exploitables. A smart contract is exploitable:

- If an attacker can use a vulnerability to transfer funds to an attacker-controlled address.

This definition helps to distinguish between vulnerabilities and exploitable more clearly. For example, following the definition, listing 3.2 does not classify as exploitable but only as vulnerable.

Another example is *SWC-107 - Reentrancy*. Reentrancy is a vulnerability that can occur when calling an external contract. The external contract then calls back into the contract before the original function invocation is finished[Claa]. Reentrancy is always a weakness and not desirable within a contract. As soon as the function executing the callback enables an exploit, reentrancy becomes a vulnerability. However, following the definition of this work, it is only classified as exploitable if the external contract can transfer funds to a controlled address through it.

## 3.4 OP-Codes interesting for Smart Contract exploits

As described in section 2.4.2 the EVM specifies how the Ethereum world state is alterable. All alterations happen through computational operations defined by specific codes - *OP-codes*.

The Ethereum Yellow Paper[W<sup>+</sup>14] provides a list of available codes as well as a list with their corresponding gas costs. OP-codes are described usually either via their hex value or their better human-readable mnemonic (e.g. *0x03* or *SUB* for the subtraction op-code). In this work, the mnemonic identifier will be used when referring to op-codes.

Op-codes are grouped into different categories that range from basic arithmetic and storage operations up to specific “Ethereum” or “Blockchain” information (e.g. *ADDRESS*, *CALLER*, *GASPRICE*).

### 3.4.1 Critical OP-Codes

Regarding the exploits of Ethereum smart contracts, a few op-codes are especially relevant, i.e. critical:

- *CALL* [0xf1]  
calls another account (e.g. sending Ether to another account).
- *DELEGATECALL* [0xf4]  
calls code of another account within its own context (i.e. contract loads code from somewhere else and executes it. The loaded code can access storage and memory of contract loading it).
- *CALLCODE* [0xf2]  
similar to *DELEGATECALL*
- *SELFDESTRUCT* [0xff]  
halts execution of contract and queues it for deletion. Any remaining funds get transferred to the provided address.
- *SSTORE* [0x55]  
saves provided word to the contracts storage

The mentioned op-codes are not the only ones that can be used for exploiting a smart contract. However, regarding the definition from 3.3, at least one of these op-codes is always needed to create an exploit.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Techniques for Automated Exploit Generation

To automatically create smart contracts exploits, said contracts must first undergo program analysis. This chapter gives an overview of different program analysis methods for smart contracts. After covering key aspects of control flow graphs, symbolic execution and code property graphs, the chapter will briefly introduce fuzzing, program slicing and taint analysis as analysis techniques. Their specific use will be described in context to the works using them - see chapter 5

## 4.1 Control Flow Graph (CFG)

A computer program usually follows a semi-predefined path/flow. It consists of a start point, an endpoint and transitions that occur on the path from start to end. As the name suggests, CFGs are visualised using a graph, where vertices are simple statements (e.g. assignments), conditional statements or whole functions. The edges represent possible transfers of control[PC90].

Contro et al. [CCCP21] point out that CFGs for smart contracts need special attention since:

- The EVM knows no concept of functions. Everything is managed through jumps.
- Jump destinations are not op-code parameters, but JUMP assumes its destination is available on the stack.
- No op-code for returning from functions exists. The return address is just pushed onto the stack, and a JUMP is performed.

To construct CFGs, first *basic blocks* - where control flow may change - have to be defined. Contro et al. [CCCP21] define these basic blocks as: Sequences of consecutively executed op-codes between a JUMP instruction and its target without any other instruction that alters the control flow. Using this definition, the beginning of a basic block is marked by the JUMPDEST op-code and the end is marked by JUMP, JUMPI, STOP, REVERT, RETURN, INVALID or SELFDESTRUCT. Once the basic blocks are defined, the edges between them are computed and added as edges. Figure 4.1 shows a CFG derived from a smart contract.

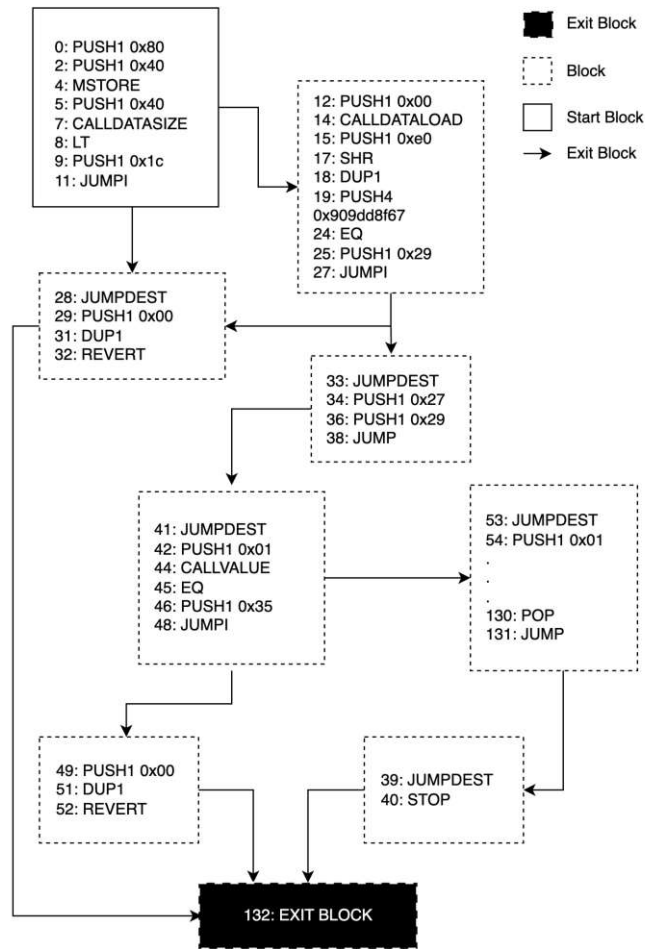


Figure 4.1: Example of a control flow graph derived from a smart contract. Adapted from [CCCP21].

## 4.2 Symbolic Execution

Symbolic execution is used to test if certain properties of a program can be violated (e.g. a division by zero performed, authentication can be bypassed). Not all desired properties

can be automatically decided. Nevertheless, approximate analysis and usage of heuristics often prove useful in practice.

In contrast to a concrete execution, symbolic execution simultaneously explores multiple paths a program could go into using different inputs. This is achieved by using symbolic inputs rather than concrete values. The execution engine gathers information about every explored path, such as:

- A boolean formula describing a “path condition” that has to be satisfied to reach the given point in the branch.
- A memory mapping from symbolic values to variables or values

A model checker evaluates the path conditions and checks every branch for satisfiability. [BCD<sup>+</sup>18]

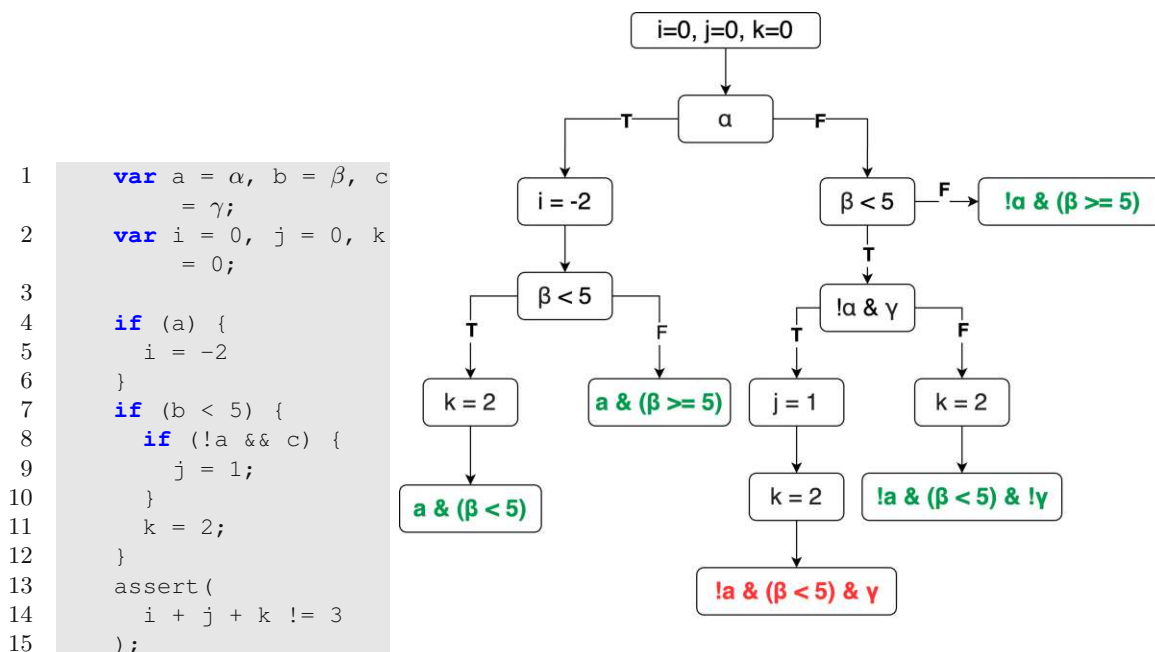


Figure 4.2: A simple program next to a visualisation of possible program paths and their corresponding path conditions at the end. Green-coloured formulas mean that they are satisfiable. The red ones are not satisfactory [Cho15].

Figure 4.2 shows the overall functioning of symbolic execution with a simple program. Line 4, 7 and 8 show conditional statements instead of using concrete values; symbolic execution uses a symbol such as  $\alpha$ . This symbol is then used as a constraint for the given path (e.g.  $a = \text{true}$  for the *true* branch). When the following conditional statement is reached, a new symbol (i.e. a constraint) is added to all previous path constraints

- resulting in a boolean formula. At any given point in a path, the model checker can evaluate the stored boolean formula, compute all requirements for reaching this point and check if it is even possible.

### 4.3 Code property graph (CPG)

A code property graph is created by combining the “Abstract syntax tree”[Wil97, FLWvG07], the “Program dependence graph”[HR92] and the CFG (see 4.1) of a program [YGAR14].

A CPG is a directed graph that consists of nodes representing source code and edges representing the relationship between them. It can easily be traversed with algorithms to analyse it and for example, discover vulnerabilities. Due to the combination of the different representations, the CPG can combine syntax, control flow and data-flow information and is, therefore able to find a large variety of vulnerabilities.[YGAR14]

Figure 4.3 shows a property graph. For a better understanding, the different relationships use different coloured edges.

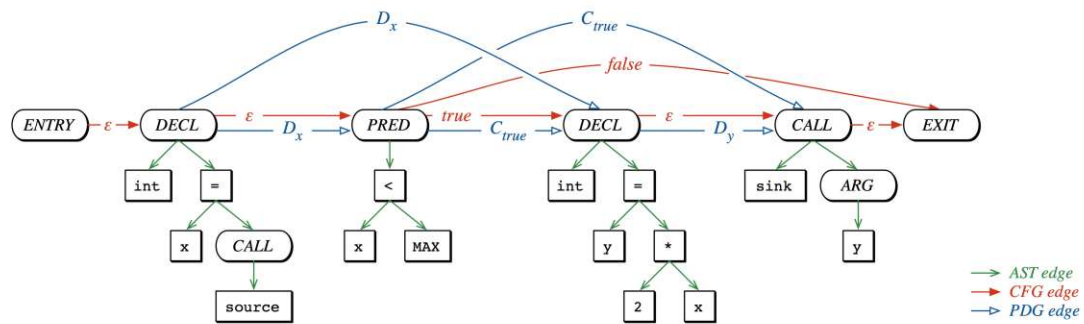


Figure 4.3: A simple control property graph. Depending on the origin of the relation to edges use different colours.[YGAR14]

### 4.4 Fuzzing

Fuzzing was introduced in the early 1990s by Miller et al.[MFS90]. It is an automatic technique to detect vulnerabilities and bugs inside pieces of software by continuously executing it with generated inputs. The generated inputs may be malformed and lead to a crash of the program. Inputs that crash a program are usually reported and used for further investigations.

Fuzzing can be divided into three main categories:

- **Blackbox-Fuzzing:** Inputs get “created” without any knowledge of the program. The input creation mainly happens following two approaches: mutation and gen-

eration. Generational blackbox fuzzing creates new inputs from scratch, while mutational blackbox fuzzing uses so-called seeds that are provided and creates new inputs by randomly mutating these seeds. In both approaches, it is usually possible to provide a grammar that inputs have to meet. The fuzzing process continues until a time or resource limit is reached[BCR21].

- **Greybox-Fuzzing:** Greybox-Fuzzing uses lightweight program instrumentation to get information about the program structure. “Coverage-based greybox fuzzing (CGF)”, which uses instrumentation to gain coverage information (see [AFL17] and [Lib17]), is the most used greybox-fuzzing technique. CGF uses seeds and mutates them to create new inputs. The gained coverage information is used to decide if inputs are either added (i.e. increase code coverage) or removed (i.e. decrease code coverage) from the seed set and which input is fuzzed next. This helps to reach deeper into the code continuously[BCR21][BPNR17].
- **Whitebox-Fuzzing:** Whitebox-Fuzzing uses symbolic execution (see section 4.2) to calculate path conditions. Instead of concrete inputs, the calculated path conditions are mutated and sent to a constraint solver for verification. If the path conditions can be satisfied, they are used as new inputs. In theory, this technique provides complete program path coverage. However, due to the large number of possible execution paths in real-world programs whitebox-fuzzing usually remains incomplete. To tackle this problem and remain effective whitebox-fuzzing also uses a predefined seed set[BCR21][God20].

## 4.5 Program slicing

Program slicing is a technique to analyse computer programs. It decomposes the program into slices containing all statements that possibly influenced a variable’s value at a certain point of the program. Program slices are executable subsets of program statements. They preserve the behaviour of the original program but only regarding a defined subset at a certain point in the program[SRH11][Kam95].

The computation of a single slice can roughly be separated into two categories:

- **Static-slicing:** gathers program statements and control predicates by traversing the CFG (see section 4.1) backwards, starting at the defined slicing criterion. Such slices are called backward slices. They contain statements that possibly affect the slicing criterion and help developers to identify bugs. Figure 4.4 shows a simple program side by side with its generated backward slice.
- **Dynamic-slicing:** takes the inputs supplied to the program during execution. It uses dynamic analysis of the program to identify statements that affect the relevant variables. Furthermore, only executions that cause an error or an anomaly are used for computation[SRH11].

```

1   DataInputStream d = new
      DataInputStream (System.in);
2   var terminat_var = Integer.
      parseInt(d.readlnt());
3   int product = 1;
4   int sum = 1;
5   for (int counter = 1; counter <=
      terminate_var; counter++) {
6       sum += counter;
7       product *= counter;
8   }
9   int average = (sum - 1) /
      terminate_var;
10  System.out.println("The Sum is :
      " + sum);
11  System.out.println("The Product
      is : " + product);
12  System.out.println("The Average
      is : " + average);

```

```

      int sum=1;
1   for (int counter = 1;
2       counter <=
      terminate_var;
      counter++) {
3       sum += counter;
4   }
5   System.out.println("The
      Sum is : " + sum);

```

Figure 4.4: A simple program next to a generated backward program slice. Taken from [SRH11]

## 4.6 Taint analysis

The idea behind taint analysis is to track values that originate from untrusted sources like function parameters. In theory taint analysis makes it possible to check if a variable is tainted at any given point in a program. This means one can check if the value of a variable is depending on any untrusted external input. Dependencies can either be “data flow dependencies” or “control flow dependencies”.[BBGM12]

- Data flow dependency:  $i$  is data flow dependent on  $j$  if  $i = j + 10$ .
- Control flow dependency:  $i$  is control flow dependent on  $j$  if the expression changing  $i$  is only reached as long as  $j$  meets certain criteria.

Taint analysis can be approached in two different ways:

- **Static analysis:** This theoretically deals with all possible runtime cases (i.e. all possible paths are covered), but due to abstract interpretation that is necessary to achieve decidability often leads to over-approximation of certain behaviours and therefore to false positives.[BBGM12]
- **Dynamic analysis:** This approach analyses code and user inputs during runtime. It is very complex to implement and since a program has to be executed with specific input values not all possible paths can be covered.[BBGM12]

# Related Work

This chapter aims to describe a selection of tools that offer support for automated exploit generation for Ethereum smart contracts. Since it was not possible to get access to the source code of every tool, they are described concerning their theoretical approach to the problem. In the end, a comparison between the pictured works is conducted.

## 5.1 Maian

Maian tries to find execution traces within contracts that violate one of the three below-defined categories.

1. Greedy contracts - contracts that lock their funds indefinitely.
2. Prodigal contracts - contracts that leak their funds.
3. Suicidal contracts - contracts that others can destroy.

Maian takes a contract's bytecode and its state at a concrete block as input. It then tries to show the existence or the absence of vulnerable transaction sequences according to the found bug.[FTB19]

### 5.1.1 Symbolic Analysis

Maian uses static symbolic analysis to find properties of execution traces that involve multiple contract invocations over multiple blocks. For this, it takes values from transactions and block parameters and tries to create a set of all possible values those inputs can take. After that, the relationship of computed contract variables is symbolically interpreted as symbolic expressions. Maian maintains two memory mappings. One maps

the symbolic variables to their symbolic expression, the other maps variables to concrete values.[FTB19]

**Execution path search:** In a first step, Maian's symbolic interpreter starts a depth-first search over all execution paths to increase coverage. The symbolic execution starts at the first bytecode instruction. It proceeds until the execution path ends either on a valid (STOP, RETURN) or invalid (e.g. invalid jump destinations) instruction. If a valid clause is encountered, symbolic execution stops and starts again from the first instruction to simulate a new invocation. If an invalid clause is encountered, the search along this path is terminated, and with the help of backtracking, another path gets selected. When reaching a conditional expression, Maian adds all branches for which the expression can be satisfied. If Maian cannot determine if the expression is satisfactory, the search down this path is terminated. Maian's depth-first search is inter-procedural but not inter-contract, meaning call instructions that have a target outside the given contract are not simulated.[FTB19]

**Handling data access:** Maian uses two memory mappings (one with concrete, one with symbolic values) that record the complete contract and blockchain storage. To ensure that reads and writes are always kept local to the current path, Maian saves concrete values read for the first time during a path into local mappings. When Maian detects variables of dynamic length (e.g. arrays), it uses an SMT solver to generate  $k$  different values, assuming the length of the variable is between zero and  $k$ . Whenever a symbolic address is accessed through memory, the search is terminated, and another branch is chosen with backtracking.[FTB19]

**Flagging Violations:** When a state is reached that violates a safety property, the contract is flagged as a buggy candidate. The path constraints, together with the property conditions, are passed to the SMT solver. Maian uses Z3[dMB08] as SMT solver.[FTB19]

**Bounding the path search space:** To avoid path explosion, Maian employs different approaches.

- Limiting the call depth.
- Limiting the total number of jumps or control transfers per path.
- Setting a timeout for every call to the SMT solver.
- Setting a maximum time that can be spent on one contract.

**Pruning:** Maian memorises already encountered configurations (= contract storage, memory and stack). A path is not explored further if symbolic execution encounters a point that matches a saved configuration.[FTB19]

### 5.1.2 Concrete Validation

To validate if a buggy candidate also holds in concrete execution, a private fork of the Ethereum chain is created, and the concrete transactions are generated. The validation framework checks the contract against the three above-defined categories:[FTB19]

- **Prodigal contracts:** To check if a contract is prodigal, it is checked if the contract sends Ether to an account that previously sent a transaction.
- **Suicidal contracts:** In this case, the framework checks if the contract bytecode is reset to “0x” after executing the provided transactions.
- **Greedy contracts:** To check if a contract locks Ether, the framework tries to confirm that the contract does not contain any `CALL`, `CALLCODE`, `DELEGATECALL` or `SUICIDE` instructions.

## 5.2 Teether

Teether defines a vulnerable contract as follows: A vulnerable contract allows an attacker to transfer Ether from it to an attacker-controlled address. Furthermore, potential attackers do not require any special capabilities to perform attacks. They only need the corresponding bytecode and must be able to submit transactions to the Ethereum network.[KR18]

Since a value transfer from one contract to another can only happen under certain conditions, Krupp & Rossow try to find paths within a contract’s control flow graph (CFG) that satisfy said conditions. To transfer Ether, the execution of one of the following four low-level EVM instructions is unavoidable `CALL`, `SELFDESTRUCT`, `CALLCODE`, `DELEGATECALL`. The first two can be used to transfer value directly. The second two can be used to inject arbitrary bytecode.[KR18]

Nevertheless, instructions are only vulnerable if the attackers can control their arguments. Since this is not always trivial to check, Teether also looks for state-changing instructions. With `SSTORE` being the only one in Ethereum. Teether uses symbolic execution to turn the path into a set of constraints if a critical path is found. With the help of constraint solving, Teether can recreate necessary transactions to trigger the critical path.[KR18]

### 5.2.1 Creation of a control flow graph

First, the EVM bytecode is used to extract the CFG. Since the EVM only utilises indirect jumps - meaning `JUMPI` and `JUMP` read their target from the stack - the target often cannot be trivially inferred. Because of this Teether uses backward slicing to reconstruct the CFG. This works as follows: In the first iteration, all `JUMP` and `JUMPI` instructions that cannot be trivially inferred are marked as unresolved. Next, an unresolved instruction is selected, and the backward slices of its jump target are

computed. If a complete backward slice can be found, the target is computed, and the instruction is marked resolved. Since a new edge in the CFG can lead to new backward slices, all JUMP and JUMPI instructions in the corresponding subtree are again marked as unresolved. This continues as long as new edges can be added to the CFG.[KR18]

### 5.2.2 Path Creation

The created CFG is then scanned for one of the four defined critical instructions. Since Teether requires that the arguments of these instructions must be controllable by attackers, backward slicing is used again, but this time on the arguments. The computed set of backward slices is then filtered for those that contain instructions which can be either directly or indirectly controlled. Since only pursuing critical paths makes sense, Teether checks after every step if all instructions of at least one critical slice can still be reached.[KR18]

### 5.2.3 Constraint Creation

Once a path is found, symbolic execution of it is started to find a set of path constraints. Teether uses an execution engine based on Z3[dMB08]. For conditional jumps, the jump target and the fall-through target are compared to the next address given by the path, and a respective constraint is added. To prevent the investigation of paths that do not correspond to feasible execution traces, Teether considers a path infeasible as soon as the program counter differs from the constructed path. Furthermore, Teether extracts a minimal infeasible sub-path to avoid the investigation of other paths containing the same conditions. In some special cases, the symbolic translation of EVM instructions is quite complicated. For example, the SHA3 instruction takes a memory region, defined by an address and a size, as input. From this input, it computes the Keccak-256 hash. Solidity uses this instruction for its mapping<sup>1</sup> data structure. The instruction is generally often used within smart contracts to hash specific values. Since both cases are very common in the Solidity world, Teether came up with a solution for symbolic modelling of SHA3. Every time a SHA3 instruction is encountered, a new 256-bit variable that models the result of the hash gets introduced. Together with the relation of it to the input data, it is later used to construct constraints. Another problem encountered during symbolic execution is instructions that can copy from or to variable-length elements. Teether uses two approaches to deal with this. First, every time when data is copied to the memory, the IF expression from Z3 is used to model conditional assignment to cover instructions of different lengths. The second approach is used when reading from memory. Here the same approach as with SHA3 is used. A new symbolic variable is created, and a mapping to the corresponding input data, the address, its length and the memory state during the read is saved. This information is later used to resolve the value of the new symbolic variable.[KR18]

---

<sup>1</sup><https://docs.soliditylang.org/en/v0.8.17/types.html?#mapping-types> - accessed 21.11.2022

### 5.2.4 Exploit Generation

As the last step, the found path constraints are checked for satisfiability. If a satisfiable path sequence is found, a list of transactions able to exploit the contract is created. If the given path sequence is not satisfiable the next path sequence is tested.[KR18]

Before checking satisfiability, additional constraints representing the attackers' goals are added. The first constraint checks if funds are transferred to an attacker-controlled address or if any arbitrary code is injected. This is done by checking if `CALLCODE`, `DELEGATECALL`, `CALL` or `SELFDESTRUCT` are called with an attacker-controlled address as the target. The second constraint added checks if an attacker is able to increase their own balance. Exploits using the `CALL` instructions are the only ones which need an additional constraint that checks if the amount of transferred Ether in the final `CALL` instruction is larger than the amount sent to the contract before.[KR18]

After the path constraints are completed, Teether tries to find a satisfying assignment using `Z3`. Since Teether uses special logic to handle `KECCAK-256` hashes and variable-length elements, the created constraints can not be passed into `Z3` directly. An iterative approach is used to gradually resolve those special variables ( $=Q$ ) First, a satisfying assignment for a subset that is not dependent on any variable of  $Q$  is computed. After an assignment is found, the next step is to resolve unresolved variables from  $Q$ . An unresolved variable can be resolved if it only depends on already resolved variables. After a variable is resolved, it gets removed from  $Q$ . This process is repeated as long as  $Q$  is not empty.[KR18]

Teether uses the following logic to resolve complex variables like `KECCAK-256` hashes: First, the input data gets evaluated to get a concrete value. This value is then used to compute a `KECCAK-256` hash. Finally, the input data and the computed hash get bound to the concrete value and to the `KECCAK-256` variable.[KR18]

## 5.3 Vultron (ContraMaster)

### 5.3.1 Semantic test oracle

Smart contracts do not crash on failure, they often just silently fail, and everything related to the execution gets reverted. Due to this, no general-purpose test oracle exists to detect vulnerabilities. The Authors of ContraMaster[WLL<sup>+</sup>20] propose a semantic test oracle in their paper that works upon two types of invariants which every transaction has to comply with:

1. **Balance invariant:** Smart contracts often manage different assets and keep track of the individual balances of participants in a dedicated variable. The *balance invariant* ensures that the relation between the general balance of the contract and the sum of the individual balances always remains the same between transactions.

2. **Transaction invariant:** The *transaction invariant* ensures that every time one of the individual balances is reduced, the same amount gets transferred to the corresponding address.[WLL<sup>+</sup>20]

Checking the invariants is done via runtime checks which are conducted before and after every transaction. Since identifying the dedicated variable holding all individual balances (=bookkeeping variable) is the biggest challenge, ContraMaster uses a specially designed algorithm to find such variables within a smart contract. The algorithm makes use of the fact that bookkeeping variables often show certain characteristics like:

1. Its datatype is a Mapping from addresses to unsigned integers.
2. It gets updated when payable functions are called/executed.
3. When Ether is received from another account, the individual balance of this account is increased by the same amount.[WLL<sup>+</sup>20]

For contracts that implement a token interface (see sections 2.2), the booking variable usually does not refer to Ether but token. Valid token contracts must implement standardised APIs to get the total contract balance as well as individual balances. These APIs can be used to implement runtime checks.[WLL<sup>+</sup>20]

### 5.3.2 Oracle-supported fuzzing

From a set of initial seeds, ContraMaster creates a collection of random transaction sequences. For each fuzzing iteration, it picks one of the sequences and runs it against the contract in an instrumented EVM. After each transaction, the contract state is checked against a semantic test oracle. If any violation is found, a vulnerability is reported, and an attacking contract with the necessary transaction sequence is created. If no violation is found, runtime information is collected and used to improve the next iteration of transaction sequences. Additional to the collected information ContraMaster uses new mutation operators tailored for smart contracts (e.g. gas limits, fallback functions) alongside traditional ones.[WLL<sup>+</sup>20]

ContraMaster is driven by a grey-box fuzzing loop.<sup>2</sup> Its goal is to generate transaction sequences that violate the test oracle. Conceptual this works as follows:

- From a contract code and an initial seed set ContraMaster creates transaction sequences.
- At the start of each fuzzing iteration, a transaction sequence is selected, and the contract state dictionary, as well as the execution trace sequence, is initialised newly and empty.

---

<sup>2</sup><https://lcamtuf.coredump.cx/afl/>

- Next, every transaction of the chosen sequence is executed, and the execution trace is stored, forming a transaction sequence trace. Additionally, the different contract states are also saved inside the state dictionary. This is later used to generate new function inputs.
- After execution is finished and ContraMaster detects a test oracle violation, the current transaction sequence is added to the output exploiting script.
- ContraMaster executes different mutation strategies. They are either based on single transactions or the whole sequence. Furthermore, different strategies are employed based on the execution results:
  - If new coverage is achieved, mutation on function inputs and gas limit is triggered.
  - If a fallback function is called inside a transaction, mutations on the fallback function are performed.
  - If new data dependencies are detected within the transaction sequence trace, the order of the transactions is mutated.
  - Last, regardless of beforehand executed mutations, ContraMaster uses contract state mutations where the contract state is randomly reset.[WLL<sup>+</sup>20]

Since a lot of vulnerabilities can only be triggered by multiple coherent transactions, this approach is necessary.

As mentioned, ContraMaster uses five different mutation strategies which are tailored for smart contracts. They will be shortly summarised below:

- Gaslimit: If the gas costs of a transaction exceed the *gaslimit*, an out-of-gas exception gets thrown. ContraMaster simulates this by mutating the gas limits for transactions.
- Fallback function: The fallback function is often used together with reentrancy and exception disorder exploits. Therefore ContraMaster mutates the function to simulate different behaviours when receiving Ether.
- Transaction sequence: The correct transaction order is essential for certain vulnerabilities to be discovered. ContraMaster uses two approaches to mutate a given transaction sequence.
  - It randomly removes or adds a transaction from/to the sequence.
  - If transactions use the same state variables, their order is changed.
- Contract state: Transaction effects heavily depend on the contract state. To simulate this behaviour, the contract state is not reset after every transaction sequence but after a predefined number of transactions.

- Function inputs: ContraMaster distinguishes between two different types of function inputs: primitive and array types. For primitive types, multiple *mutation ranges* are generated by picking values from previously encountered state variables. Additionally to the picked values, new ones are created within the ranges by randomly negating bits in these values. For array types, the procedure is quite similar. If the length is fixed ContraMaster uses the same technique as for primitive types. If the length is arbitrary, ContraMaster first generates a random positive number as array length. It then again uses the same technique as it uses for primitive types. [WLL<sup>+</sup>20]

In addition to custom mutations, ContraMaster also uses an extensive feedback system divided into control-driven, data-driven and contract-state feedback.

- Control-driven feedback: Helps to discover more edges within the CFG, favouring not yet explored edges.
- Data-driven feedback: Different transactions do not necessarily cover new CFG edges. Therefore, a data-driven feedback approach is needed for transaction sequence mutation. The goal is to switch the order of transactions using the same state variables to discover new data dependencies.
- Contract-state feedback: Since transactions heavily rely on contract state, it is used to generate function inputs. For this, the contract state is exported as a dynamic dictionary. [WLL<sup>+</sup>20]

## 5.4 EthPloit

Unlike Teether (5.2), EthPloit uses fuzzing to generate transaction sequences and tries to exploit smart contracts. With this approach, EthPloit tries to solve the problem of unsolvable constraints, which arises when using symbolic execution. To do so, EthPloit utilises a dynamic seed strategy using runtime feedback. This feedback is used to find the solutions to cryptographic functions from execution histories. Additionally, the authors of EthPloit claim that it is easier to simulate blockchain effects such as modification of timestamps and block numbers. For these simulations, EthPloit uses a modified EVM to provide custom configurations. Furthermore, taint analysis is used to rule out invaluable test candidates beforehand. [ZWLM20]

EthPloit focuses on vulnerabilities on contract-level [JLC18], based on [KR18] and [NKS<sup>+</sup>18] three categories of vulnerabilities are defined: Balance increment, self-destruction and code injection. A successful exploit depends on two requirements. First: at least one execution path must trigger one of the critical commands mentioned in 3.4.1. Second: before the critical instructions are executed one or more modifications of the contract state have to be executed. [ZWLM20]

EthPloit's workflow is divided into the following five steps:

### 5.4.1 Static analysis

**Control flow graph creation:** After compiling the Solidity source code EthPloit starts its static taint analysis, which is built upon Slither[FGG19]. In the first step, the taint analyser creates a CFG for each function. Every edge in the CFG connects two nodes (= a solidity expression) that are executed sequentially. Each node comprises written and read variables. For every conditional branch within the control flow a branch in the CFG is created and the execution flow of the branch is pictured.[ZWLM20]

**Taint source and sink labelling:** Variables that can be directly or indirectly controlled by the sender of a transaction are marked as taint sources. Such variables are state fields, function arguments and blockchain properties. Additionally, external calls and state variables are labelled as sinks. After labelling, the marked nodes are used for taint propagation.[ZWLM20]

**Taint propagation:** To see how taint values propagate through functions, EthPloit extracts variable dependencies for taint labels and sinks. For this the authors define two types of dependencies:

1. Variable-data dependency.  
This dependency means, that the value of one variable is controlled by another one (e.g..  $i = j + 10$ ).
2. Variable-control dependency.  
Here a variable  $i$  is control-dependent if the expression changing  $i$  is only reached if variable  $j$  meets certain conditions.[ZWLM20]

The taint analyser iteratively traverses the CFG. During each iteration it propagates the taint onto nodes, this happens in three stages.

1. First for every node that has no predecessors a flow set for each taint source is created. The set is defined as  $Taint(src) \leftarrow \{src\}$ . If predecessors for the node exist, a new *Taint* is created by taking the union of all predecessors *Taint*. The so-created *Taint* shows the status of taint propagation at the current node.
2. Next dependencies are extracted. For example, in a node, written variables are variable-data dependent on read-variables or written variables are variable-control dependent on read-variables if they belong to control-dependent nodes.
3. In the last step, the taint status is updated. This is done by adding all dependencies to the taint sources or sinks.[ZWLM20]

### 5.4.2 Test case generation

Testcases are created as a set of transactions, which are based on the given contract and its variable dependencies. For optimisation, a fuzzing approach is used. The creation process happens in four steps.

1. **Depicting dependencies with a Taint relation graph (TRG).**  
To optimise the creation of test cases a Taint relation graph is constructed. Each node in the TRG is either a taint source or a sink. Since external calls are crucial for exploits, only dependencies to such are analysed (dependencies come from taint analysis). The TRG is constructed as follows: The TRG generation starts at the last transaction of the given test case. For each taint source related to external calls, an edge from the source to the call is added to the TRG. If a source contains state variables those are put into *target sinks* set. In the second step, all previous transactions are traversed until the first transaction of the test case is reached. For all these transactions taint sources that are related to state variables in *target sinks* are extracted and for each such source another edge from it to the target sinks is added. Furthermore, all state variables of the sources are put into the set *target sinks* again. After the TRG is created, a test case is considered to be a candidate for fuzzing, if there is a path from a taint source to any external call inside the last transaction.
2. **Function selection.**  
For every transaction of the given test case, a function of the contract has to be selected. Suitable functions have to be public, callable and have to contain external calls or modify the state in some way. For each transaction, possible candidates are collected on the basis of the TRG, from all suitable functions. Of the candidates, one is chosen randomly with a probability distribution.
3. **Argument generation.**  
After all functions are selected, function arguments are created. Arguments can be created from two different sources. The first is a pseudo-random generator and the second one is dynamic seeds that are filled and modified through runtime feedback. Which source is chosen is decided by a predefined probability  $p$ . The chances for the pseudo-random approach are  $p$  and the probability for the dynamic seed approach is  $1-p$ .
4. **Property generation.**  
The generated message properties are *msg.sender* and *msg.value*. The message sender is chosen from a predefined set of attacker accounts, while the value is a randomly generated *uint256* value. To rule out exceptions a message value is only created if the called function is payable. The generated blockchain properties are *block.number* and *block.timestamp*. [ZWLM20]

### 5.4.3 Test case execution

EthPloit uses an instrumented EVM which is based on the remix-debugger<sup>3</sup>. This makes it possible to extract the stack and memory at each opcode for a transaction. For simulating blockchain effects EthPloit offers an API to modify block properties for each test case execution. Furthermore, the instrumented EVM can forcefully trigger exceptions to simulate failing external calls. EthPloit makes use of this by executing every transaction with an external call twice. Once with a successful call and the other time an exception gets thrown.[ZWLM20]

### 5.4.4 Trace analysis

The analysis of the execution trace created during the test case execution is done by two modules - the *coverage guider* and the *exploit detector*.

1. **Coverage guider:** The coverage guider uses *critical instruction coverage*, where it only focuses on critical instructions (see 3.4.1), to measure test case coverage. If a test case reaches a not yet covered critical instruction it gets tagged as a coverage increment and triggers two reward effects: First, some runtime values will be used as new seeds for argument generation later on and second the probability of functions, in a coverage-increasing sequence, for being chosen during the function selection is increased.
2. **Exploit detector:** EthPloit uses three oracles to check if an exploit occurred in a test case. One oracle checks if the balance of any attacker account has increased, if so a successful “Balance increment exploit” is reported. The second oracle checks if the SELFDESTRUCT opcode appears within the execution traces of a test case. If the opcode is found then the test case is marked as successful “Self-destruction” exploit. Finally the third oracle checks for code injection. For this, it searches for all CALLCODE and DELGATE opcodes and then checks if the destination of the calls is controlled by an attacker. This is done by accessing the program stack after execution and checking the destination.[ZWLM20]

### 5.4.5 Dynamic seed strategy

EthPloit uses feedback to create suitable function arguments and choose proper functions for test case generation. A dynamic seed strategy, where new seeds are added based on trace analysis after every transaction execution, is applied to create function arguments. The authors of [ZWLM20] differentiate between two types of seeds. Local and global. Global seeds persist throughout the fuzzing of a complete contract. They get updated for example if a coverage increment is found. In this case, all arguments used during that execution are added to the seed. On the other hand, local seeds are only for single

<sup>3</sup><https://remix-ide.readthedocs.io/en/latest/debugger.html> - accessed on 23.06.2022

transaction sequences (=test case). They get updated after every transaction and are used to generate the following transaction arguments. Local seeds use four sources for their argument creation:

- Previous arguments of transactions get labelled as local seeds.
- State variables. The latest state variables after each transaction become local seeds.
- In- and output of certain calls. In- and output values of functions that are hard to predict (e.g. hash-functions) get stored as seeds.
- Constant values. Literal values and constant values of each function body are used for local seeds.

To choose a fitting seed for each argument, potential seeds must have the same type as the target argument. Furthermore, the seed and the argument must taint the same taint sink. This is ensured by checking the paths of the TRG.[ZWLM20]

### 5.5 EthFuzz

EthFuzz uses a hybrid approach that combines “static call graph analysis”, “dynamic execution” and “symbolic execution” to find vulnerabilities and generate exploits. It uses EVM bytecode as input and makes use of the “Parity Ethereum Client”<sup>4</sup>. EthFuzz defines three main stages:

1. Code property graph analysis.
2. Dynamic execution.
3. Symbolic testing.[Ash21]

#### 5.5.1 Call graph analysis

In the first step, a graph model of the target contracts bytecode is created to identify potentially exploitable execution paths. EthFuzz bases its graph models on Code property graphs (CPGs) [YGAR14]. CPSs make it possible to find data dependency paths corresponding to critical instructions. These paths are later used for symbolic execution.[Ash21]

To generate contracts call graphs EthFuzz uses “Porosity”<sup>5</sup>, an unmaintained decompiler that contains many bugs that have to be addressed.[Ash21]

---

<sup>4</sup><https://github.com/openethereum/parity-ethereum>

<sup>5</sup><https://github.com/msuiche/porosity>

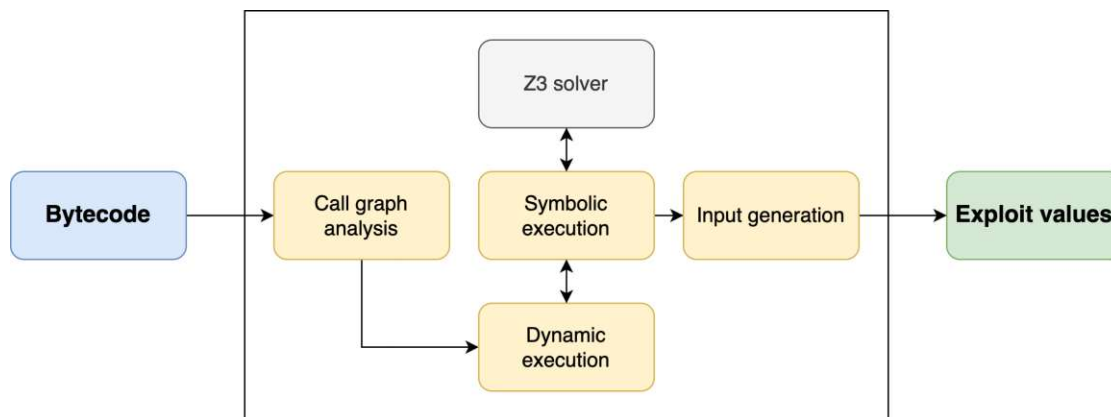


Figure 5.1: Architectural overview of EthFuzz. Adapted from [Ash21]

Backward slicing:

In order for exploits to be able to reach critical instructions, the correct entry point is necessary. Finding an entry point is done by backward slicing of critical instructions. EthFuzz does this the following way:

1. From the before generated graph critical instructions are extracted.
2. For every critical instruction all incoming data dependency links are investigated until either an entry point or a function argument is found.
3. If a function argument is found step 2 is repeated until a source is found.[Ash21]

Since executing non-exploitable paths causes overhead and can lead to false-positive results, such paths get pruned

### 5.5.2 Dynamic execution

Including external libraries in smart contracts is very common. To reduce false-positive findings it is important to detect the presence of such external libraries within execution paths. EthFuzz uses dynamic execution to evaluate target paths with runtime data. During the execution, a symbolic variable (=“Taint label”) is added to the path. It is initially set to zero. The taint label propagates through the path without changing the result. With this approach, it is possible to detect the sources of tainted data at a critical point.[Ash21]

### 5.5.3 Symbolic testing

The exploitable paths are modelled as logical formulas and passed into a Z3 SMT solver. The solver executes the given formula symbolically and produces a set of path constraints

and actual values. To avoid infinite loops and infeasible paths during symbolic execution, both the collected actual and runtime data from dynamic execution are used. To keep overhead small EthFuzz first tries to exploit a single path. If this does not lead to success, more extensive path sequences are tried.[Ash21]

## 5.6 Comparision

This section gives a short comparison of the mentioned tools. Table 5.1 visualises the different techniques used by the tools as well as the vulnerabilities they claim to detect. The techniques as well as the vulnerabilities are abbreviated with two-letter codes to retain readability.

- **Static code graph analysis (SA):** This includes techniques that create a “graph-representation” of a computer program. For example control-flow-graphs (see 4.1) or code-property-graphs (see 4.3).
- **Symbolic execution (SE):** This indicates that the paper uses some sort of SE to create path constraints (see 4.2).
- **Taint analysis (TA):** Taint analysis finds sensitive operations that are dependent on values that possibly can be controlled by external parties (see 4.6).
- **Backward slicing (BS):** Backward slicing decomposes a program into slices containing all statements that possibly influenced a variable’s value at a certain point of the program (see 4.5)
- **Fuzzing (FZ):** Fuzzing is an automatic technique to detect bugs inside a program by continuously executing it with generated inputs (see 4.4).
- **Practical tests (PT):** This category indicates if the given tool uses a real or test blockchain to generate or verify their exploits.
- **Etherleak (EL):** Exploits that transfer Ether from a contract to an attacker-controlled address are classified as Etherleaks.
- **Etherlock (EO):** An etherlock describes a contract that exists and is alive on the blockchain but any Ether that gets deployed there can not be retrieved anymore by anyone.
- **Reachable Selfdestruct (RS):** A contract contains a reachable selfdestruct if it can be triggered by an arbitrary address.
- **Integer Over-/Underflow (IO):** Over-/Underflows can occur when using high-level programming languages such as Solidity. Since Solidity only introduced

language support for over- and underflows in version 0.8, it is important to check against them.<sup>6</sup>

- **Code injection (CI):** Op-codes like CALLCODE and DELEGATECALL (see 3.4.1) allow importing external code. Arbitrary contracts are so to say able to inject code into the execution.
- **Transaction order dependency (TO):** If the order of two or more transactions calling the same contract changes the outcome, malicious parties can exploit this behaviour.
- **Exception disorder (ED):** Again when using high-level programming languages, there are some low-level operations (e.g send ()) that do not throw an exception when failing.

Tools	Analysis Techniques						Vulnerabilities						
	SA	SE	TA	BS	FZ	PT	EL	EO	RS	IO	CI	TO	ED
Maian		✓				✓	✓	✓	✓				
Teether	✓	✓		✓			✓						
EthPloit	✓		✓		✓	✓	✓			✓	✓		
EthFuzz	✓	✓		✓			✓	✓	✓	✓	✓	✓	✓
Vultron					✓	✓	✓	✓		✓			✓

Table 5.1: This table shows which analysis techniques the tools use and which vulnerabilities they claim to detect. To keep the table more readable the techniques as well as the vulnerabilities are abbreviated to two-letter identifiers.

Table 5.1 shows that the tools published earliest (Tether/2018 and Maian/2019) focus on fewer vulnerabilities, but they serve as building blocks for the other later published tools.

Vultron and EthPloit were both published in 2020. While Vultron focuses solely on a fuzzing approach EthPloit implements a combination of static analysis and fuzzing.

Comparing Vultron and EthPloit it seems that concentrating on only one technique is superior in comparison to merging them (i.e. Vultron finds more vulnerabilities than EthPloit), but upon closer inspection, one can see that EthPloit finds code injection vulnerabilities (code injections are part of the most repeated instructions within public exploit databases) and Vultron does not. Another point not in favour of Vultron is, that it runs only on-chain with no option to run it off-chain. Furthermore, so-called attacking contracts have to be created. They are contracts containing a reference to the vulnerable contract and implement a wrapper for all public non-view functions. Since the fallback function also always calls one of the public functions (or reverts), the number of attacking

<sup>6</sup>Changelogs for Solidity version 0.8: <https://docs.soliditylang.org/en/v0.8.17/080-breaking-changes.html>

contracts has to be *number of public functions* + 1. This can result in a considerable amount of attacking contracts when testing larger contracts.

EthFuzz is the newest of the listed tools. It was published in 2021. Same as Teether it uses static code graph analysis as well as symbolic execution and program slicing. Instead of using the CFG however, it uses a CPG which offers more information. Additionally, it offers the option to specify new vulnerabilities in a dedicated data catalogue, making it the only tool that accepts “updates”. The only argument against EthFuzz is that it does not implement an automatic practical test (i.e. it does not test its exploit against a real blockchain). Verification of generated exploits has to be done manually which makes testing against larger data sets harder and tiresome.

# Implementation of a Prototype

This chapter describes the approach to implementing the prototype used in this thesis. It aims to clarify the reasons behind specific design decisions, discuss the problems encountered during the implementation process, and explain the corresponding solutions that were employed.

## 6.1 Foundation for the prototype

After comparing the currently available exploit tools (see chapter 5 ) it was decided to base the implementation of the prototype on one of the analysed tools. Since the source code is only accessible for *Tether*, *Maian* and *Vultron*, *Vultron* was chosen because it is the most recent of these three.

### 6.1.1 Problems encountered while using Vultron

When investigating the source code of *Vultron*<sup>1</sup> some minor and major problems came up.

The first problem that surfaced was the existence of different branches with different development statuses. To determine which branch should be used the creators of *Vultron* were contacted. Based on their reply the `ContraMaster-cmdline-version`<sup>2</sup> branch was chosen as the foundation.

Unfortunately, screening and running the selected branch brought up new problems:

- The command-line branch still uses some kind of `silent-server` architecture (the web-UI used by the `default` branch was removed but `http-request` were still

<sup>1</sup>Link to Vultron source code: <https://github.com/ntu-SRSLab/vultron> - accessed 10.05.2023

<sup>2</sup>Link to Vultron-branch used as foundation for prototype: <https://github.com/ntu-SRSLab/vultron/tree/ContraMaster-cmdline-version> - accessed 10.05.2023

used under the hood). Because of the nature of this setup, it was based on eventEmitters to trigger new contract deployments and fuzzing rounds.

- Since *Vultron* performs its tests on a real blockchain and uses the execution traces for creating meaningful fuzzing inputs, communication between the fuzzer and the running blockchain is necessary. For this *Vultron* uses a modified *Aleth* client<sup>3</sup> that should communicate with the fuzzer via calling an endpoint of the `silent-server` after every execution cycle. While testing *Vultron* it became apparent that this connection was never established and the execution information was never passed to the fuzzer. Unfortunately, the creators of *Vultron* did not give an explanation or solution for this problem.
- Another problem occurred when trying to run different contracts than the ones provided by the *Vultron*-benchmark. *Vultron* uses so-called attack contracts to conduct the fuzzing. These contracts wrap all public-callable functions of the target contract and each contract calls one of the public functions from its own fallback function. The provided script to create said attack contracts, unfortunately, had some shortcomings - e.g. the solidity version for the created constructor was hard coded to `^0.4.19`. This made the script not feasible when working with contracts that use newer solidity versions.
- The created exploits revealed another problem. Exploits for a given contract are saved into a `exploits.txt` file. This file then contains a list of all the different executed call sequences but does not include any information about the parameters used when calling the different functions.
- The creators of *Vultron* decided on using Truffle<sup>4</sup> for compiling and deploying contracts. While running *Vultron* it was not possible to make this setup work. For this reason, it was necessary to replace Truffle with a custom solution for compiling and deploying contracts.

The following section (see 6.2) deals with solutions to the above-described problems as well as design decisions that were made when implementing the prototype.

### 6.2 Design decisions and implementation details

On one hand, the design decisions and implementation details described in this section were made to reduce overhead or simplify certain aspects of the prototype, on the other hand, they sometimes were driven by the problems mentioned in 6.1.1.

The list below tries to give a short overview of the different design and implementation decisions. The individual decisions will be described more accurately in dedicated sections.

---

<sup>3</sup><https://github.com/ethereum/aleth> - accessed 09.05.2023

<sup>4</sup><https://trufflesuite.com/docs/truffle/> - accessed 09.05.2023

- Replacing script used for creating attacking contracts
- Replacing *Truffle* with custom compile and deployment scripts.
- Replacing the *silent-server* approach with a custom *Javascript* wrapper.
- Replacing *Aleth* with *go-ethereum (geth)*<sup>5</sup> as blockchain client.
- Replacing logic to extract information of call sequences that led to exploits.
- Skipping of contracts that do not specify a solidity version.
- Skipping of contracts that encounter an error while compiling.
- Skipping of contracts that do not have an ABI or do not expose any public functions.
- Skipping of contracts that cannot be deployed within 60 seconds.
- Creation of a Docker setup to enable parallel test runs.

### 6.2.1 Replacing script used for creating attacking contracts

In order to utilize Vultron, it is essential to create attacker contracts (refer to sections 5.3 and 6.1.1) based on the target contract prior to conducting fuzzing. However, the provided script did not consider the Solidity version of the target contract and instead generated attack contracts hardcoded to version <sup>^</sup>0.4.19. In order to achieve optimal test results, the attack contracts should use the same Solidity version as the target contract. Therefore, a new script was developed to create attack contracts that not only account for the Solidity version of the target contract but also handle any breaking changes across different Solidity versions.

For instance, listings 6.1 and 6.2 demonstrate the evolution of a simple constructor implementation over time. Prior to Solidity version *0.5.0*<sup>6</sup>, the constructor was implemented as a function named after the contract and had to be declared as either `public` or `internal`. Between Solidity versions *0.5.0* and *0.7.0*<sup>7</sup>, the constructor had to be implemented as shown in listing 6.2, with the additional requirement of specifying the visibility using either `public` or `internal` alongside the `constructor` keyword. Since version *0.7.0*, the specification of visibility is not necessary anymore.

<sup>5</sup>Link to the documentation of go-ethereum: <https://geth.ethereum.org/> - accessed 09.05.2023

<sup>6</sup>Link to the Solidity documentation of version 0.4.21: <https://docs.soliditylang.org/en/v0.4.21/contracts.html#constructors> - accessed 11.05.2023

<sup>7</sup>Link to the Solidity documentation of version 0.7.0: <https://docs.soliditylang.org/en/v0.7.0/contracts.html#constructors> - accessed 11.05.2023

```

1 pragma solidity ^0.4.19;
2 contract ContractOldVersion {
3     function ContractOldVersion() public {
4         ...
5     }
6     ...
7 }

```

Listing 6.1: Example showing how a constructor could be implemented prior to Solidity 0.5.0

```

1 pragma solidity =0.8.2;
2 contract ContractNewVersion {
3     constructor () {
4         ...
5     }
6 }

```

Listing 6.2: Example showing how a constructor can be implemented since Solidity 0.7.0

Another important aspect when creating the attack contracts is choosing the right Solidity version. Since Solidity follows the semantic versioning of npm<sup>8 9</sup> the specified version may be defined as a semantic range (e.g. ^0.4.3). In cases where the specified version is not a concrete value but a range, the smallest possible version is used for the attack contract.

### 6.2.2 Replacing *Truffle* with custom compile and deployment scripts.

For compiling and deploying contracts, *Vultron* originally uses *Truffle*. However, due to difficulties in getting this setup to function properly, a custom compilation and deployment script were implemented instead.

The custom compile script is used to prepare the provided contracts for the fuzzing process while the deployment script is integrated into the script managing the fuzzing.

When implementing the compile script the two biggest issues were:

- Extracting and choosing the right Solidity version.
- Merging the present information into a usable format.

To compile a contract `solc-js:0.8.17`<sup>10</sup> is used. For this, the contract first is read into memory as a string. Afterwards, the Solidity version is extracted and the source

<sup>8</sup>Link to the version-pragma section of the Solidity documentation: <https://docs.soliditylang.org/en/latest/layout-of-source-files.html#version-pragma> - accessed 11.05.2023

<sup>9</sup>Link to the documentation of the semantic versioning used by npm: <https://github.com/npm/node-semver> - accessed 11.05.2023

<sup>10</sup>Link to Github repository of solc-js: <https://github.com/ethereum/solc-js> - accessed 11.05.2023

string is passed into solc-js for compilation. Extracting the Solidity version is done via the code snippet shown in listing 6.3: Same as in 6.2.1 the smallest possible Solidity version is chosen to compile the contract.

```

1 const getSemverExpression = (source) => {
2   const result = source.match(/pragma solidity(.*)/);
3   return result && result[1] ? result[1].trim() : undefined;
4 };

```

Listing 6.3: Code snippet showing logic to extract the Solidity version out of a contract.

In order to perform fuzzing and create attack contracts, it is necessary to obtain specific information for each contract. To accomplish this, the compilation options displayed in Listing 6.4 are used as input for solc-js. The resulting compiled output is then enhanced with additional metadata, such as the Solidity version and whether the experimental abiEncoder is enabled. This additional metadata is needed for the creation of attack contracts, as illustrated in listing 6.5.

```

1 {
2   language: "Solidity",
3   sources: {
4     [<contractName>]: {
5       content: <source string>,
6     },
7   },
8   settings: {
9     outputSelection: {
10      "*": {
11        "*": [
12          "metadata",
13          "abi",
14          "evm.bytecode.object",
15          "evm.bytecode.sourceMap",
16          "evm.deployedBytecode.object",
17          "evm.deployedBytecode.sourceMap",
18        ],
19        "": [
20          "ast", // Enable the AST output of every single file.
21        ],
22      },
23    },
24  },
25 };

```

Listing 6.4: Compile options passed to solc-js.

```

1 {
2   contracts: {
3     [<contract name>]: {
4       abi: <application binary interface>
5       bin: <deployed bytecode>,
6       "bin-runtime": <runtime bytecode>,
7       srcmap: <deployed sourcemap>,

```

```

8     "srcmap-runtime": <runtime sourcemap>,
9     version: <solidity version extended format>,
10    versionShort: <solidity version short format>,
11    abiEncoderEnabled: <boolean>
12  },
13 },
14 };

```

Listing 6.5: Final output used for creating and deploying contracts.

### 6.2.3 Replacing the *silent-server* approach with a custom *Javascript* wrapper.

As briefly mentioned in 6.1.1 *Vultron* uses a *silent-server* approach for their command line version. This means they spin up a server and expose certain end-points without rendering any user interface (UI).

```

1  app.listen(port, () => {
2    // myEmitter.emit("eventCopyBenchmark");
3    test(...parameters)
4    .then(answer=>{
5      console.log(answer)
6    }).catch(err=>{
7      console.error(err);
8      console.trace("show testing error");
9    });
10 });

```

Listing 6.6: Snippet showing how *Vultron* starts its *silent-server* and starts the fuzzing process. [WLL<sup>+</sup>20]

Listing 6.6 shows how *Vultron* starts its silent-server and fuzzing process. It starts the server and immediately calls the `test` method which then interacts with the actual fuzzer. The `test` method loads the provided contracts into the fuzzer and starts the seeding process. After that, an event listener waits for and reacts to emitted events. The actual fuzzing is initiated only when the `/fuzz` endpoint of the silent-server is invoked, which is not performed by the fuzzer itself but should be triggered within the custom *Aleth* client that *Vultron* utilizes. Throughout my testing, I encountered several issues with this configuration. These problems included events failing to trigger, events not being sent at all, or the fuzzing endpoint not being called. Because of this, the decision was made to remove the silent-server approach and wrap the fuzzer with a custom *Javascript* script.

The custom script traverses all provided contracts, sets the needed solidity version, deploys them to the blockchain and after that starts the fuzzing. The fuzzing is conducted on a “per folder” basis. The script first deploys the target contract, saves its allocated address and then deploys the attacking contracts. After all contracts of one folder are deployed the fuzzing is carried for it.

With this approach, it is easy to make changes or add new functionality to the single steps. The new script is also no longer dependent on the correct handling/triggering of

events. Furthermore, no custom blockchain client is needed since the communication with it is now triggered by the script and not the other way round (see 6.2.4).

#### 6.2.4 Replacing Aleth with geth as blockchain client

As briefly mentioned in 6.1.1 *Vultron* uses a custom *Aleth* client. The custom client<sup>11</sup> is needed to establish a connection between the fuzzer and the running blockchain. The idea is to record the execution trace for every transaction of the blockchain and subsequently send it back to the fuzzer by calling the `/fuzz` endpoint of the silent-server.

This approach introduces various problems:

- Since this approach extends the source code of the original project it gets hard to update the client to newer versions without losing or manually adding the extension again.
- By hardcoding the server address (see listing 6.7 line 55) the silent-server is restricted to only use this address.
- In a blockchain environment, many different activities take place, such as transactions and the creation of new blocks. As a result, a large number of log messages are generated, which can make it easy to miss updates from the Tracerecorder. This, in turn, can make debugging more difficult.

Additionally to the above-mentioned problems, the *Aleth* client was deprecated by Ethereum on October 6<sup>th</sup> 2021.<sup>12</sup>

All these points led to the conclusion to replace the *Aleth* client with the newer well-maintained *geth* client. With this switch, it is also possible to gather execution traces for all relevant transactions using the *Javascript* wrapper instead of extending the *geth* source code. *Geth* offers a few predefined tracers that can be used, but to keep the format exactly the same as defined before a custom tracer-config was used to extract execution traces from *geth* (see listing 6.8).

```

45 void TraceRecorder::sendTrace() {
46     std::string instructions;
47     join(this->instructions, ',', instructions);
48     instructions = "[" + instructions + "];";
49
50     ...
51
52     std::string json = "{\"hash\": \"0x\"+this->txHash+\"\", \"address
53         \": \"\"+\"0x\"+this->contractAddress+\"\", \"
                    \"trace\": \"+instructions+\"}";

```

<sup>11</sup>Link to the Github repo of the custom Aleth client: <https://github.com/ntu-SRSLab/AlethWithTraceRecorder> - accessed 12.05.2023

<sup>12</sup>Link to deprecated Ethereum software: <https://ethereum.org/en/deprecated-software/#aleth> - accessed 12.05.2023

```

54     LOGMSG("will send trace to vultron server");
55     httpplib::Client cli("localhost", 3000);
56     auto res = cli.Get("/");
57     if (res && res->status == 200) {
58         LOGMSG("connected to vultron server");
59         res = cli.Post("/fuzz", json, "application/json");
60         if (res && res->status == 200) {
61             std::cout << "[" << __FILE__ << ":" << __LINE__ << "]" <<
                res->body << std::endl;
62         } else {
63             std::cerr << "[" << __FILE__ << ":" << __LINE__ << "]" << "
                cannot transfer trace info to vultron server" << std::
                endl;
64         }
65     } else {
66         std::cerr << "[" << __FILE__ << ":" << __LINE__ << "]" << "
                cannot connect to vultron server" << std::endl;
67     }
68 }

```

Listing 6.7: Snippet showing how *Vultron* extends the default Tracerecorder of *Aleth* to transport the recorded execution trace to the silent-server.<sup>13</sup>

```

1  {
2      id: 1,
3      method: "debug_traceTransaction",
4      params: [
5          <hash of relevant transaction>,
6          {
7              tracer: `{
8                  retVal: [],
9                  step: function (log, db) {
10                     this.retVal.push(log.getPC()+log.op.toString());
11                 },
12                 fault: function (log, db) {
13                     this.retVal .push("FAULT: " + JSON.stringify(log));
14                 },
15                 result: function (ctx, db) {
16                     return this.retVal;
17                 },
18             }`,
19         ],
20     ],
21 }

```

Listing 6.8: Tracer config used to extract execution traces from geth in the same format as previously defined by *Aleth*

<sup>13</sup>Link to the extended Tracerecorder used by *Vultron*: <https://github.com/ntu-SRSLab/AlethWithTraceRecorder/blob/master/libevm/TraceRecorder.cpp> - accessed 12.05.2023

### 6.2.5 Replacing logic to extract information of call sequence that led to exploits

The goal of *Vultron* is to generate reusable/reproducible exploits. Listing 6.9 shows an excerpt of a so-called *exploit.txt* file generated by *Vultron*.

```

1  ...
2  512: #vultron_grantBounty#transferBounty#claimBounty
3  1495: #vultron_grantBounty#transferBounty#claimBounty
4  6575: #vultron_grantBounty#transferBounty#claimBounty#vultron_grantBounty
5  15509: #vultron_grantBounty#transferBounty#claimBounty
6  19061: #vultron_grantBounty#transferBounty#claimBounty#grantBounty
7  22889: #vultron_grantBounty#transferBounty#claimBounty
8  540: #vultron_grantBounty#grantBounty
9  ...

```

Listing 6.9: Excerpt of an exploit file generate by *Vultron*[WLL<sup>+</sup>20].

This file contains a list of **all executed** transactions sequences formatted in the following way:

*<time since fuzzing start in ms> : <name of first function in sequence># ... #<name of n<sup>th</sup> function in sequence>*

This approach faces two major problems:

- The file stores every executed transaction sequence without any indication if an exploit was found.
- Only the names of the functions are stored. For instance, there is a lack of information regarding whether the function accepts input parameters and what specific inputs were utilised.

To provide reproducible exploits the provided logic to extract exploits had to be overwritten. Listing 6.10 displays the output generated by the new approach.

- Every transaction sequence that gets classified as an exploit is written into its own JSON file.
- The generated file contains an array of all transactions that led to the exploit.
- Every transaction contains the following information:
  - “from”: the address of the caller contract
  - “to”: the address of the called contract
  - “abi”: the ABI of the called function. This contains information about the expected inputs, the name of the function and the four-byte signature of the function.

- “gas”: the specified gas for the transaction.
- “param”: An array containing the inputs used with this transaction.

Storing the found exploits in this new format gives the user the possibility to execute every transaction that led to the exploit again using the same parameters as the fuzzer.

```

1  [
2    {
3      "from": "0x6042987EB57Bf4d16a63e68d8f8018E2c12e8FC5",
4      "to": "0x22C526E7874dc352010Bd2033af49550529e8eaC",
5      "abi": {
6        "inputs": [],
7        "name": "claim",
8        "outputs": [],
9        "stateMutability": "nonpayable",
10       "type": "function",
11       "signature": "0x4e71d92d"
12     },
13     "gas": 4000000,
14     "param": []
15   },
16   {
17     "from": "0x6042987EB57Bf4d16a63e68d8f8018E2c12e8FC5",
18     "to": "0x22C526E7874dc352010Bd2033af49550529e8eaC",
19     "abi": {
20       "inputs": [
21         {
22           "internalType": "uint256",
23           "name": "amount",
24           "type": "uint256"
25         }
26       ],
27       "name": "migrate",
28       "outputs": [],
29       "stateMutability": "nonpayable",
30       "type": "function",
31       "signature": "0x454b0608"
32     },
33     "gas": 4000000,
34     "param": [
35       "0x87f056ec015c4e70"
36     ]
37   }
38 ]

```

Listing 6.10: Output file generated by the new logic. Showing all necessary information to reproduce the found exploits.

### 6.2.6 Skipping of contracts that do not specify a Solidity version

When facing contracts that do not specify a Solidity version there are two possibilities:

- Trial and error compiling: Since no version is specified the only possibility regarding compiling is trying a random Solidity version and see if the contract compiles. If it does not compile another version has to be tried.
- Skipping said contract

Potentially compiling every contract that does specify a Solidity version multiple times costs computational resources and time; making it not feasible for our case, therefore approach two was chosen and contracts that do not specify a version are skipped.

### 6.2.7 Skipping contracts that encounter an error while compiling

Contracts that fail during the compilation process do not produce any `bytecode` or `ABI` that can be utilised for generating attack contracts or facilitating deployment. As a result, these contracts are unusable and consequently skipped in the subsequent steps.

### 6.2.8 Skipping of contracts that do not have an ABI or do not expose any public functions

As previously discussed (see section 5.3 & 6.1.1) *Vultron* utilises attack contracts to conduct fuzzing. However, if the target contract lacks an `ABI` or does not expose any public functions, the resulting attack contract lacks the necessary means to interact with the target contract. Deploying and fuzzing such contracts would needlessly consume resources, making it more sensible to skip them in order to optimise the utilisation of available resources.

### 6.2.9 Skipping contracts that cannot be deployed within 60 seconds

While deploying the target contract and all its attack contracts the fuzzing is on hold because it needs the addresses of the contracts. To minimise fuzzing time and prevent deathlocks, contracts that do not return an address within 60 seconds (i.e. contracts that are stuck inside deployment) are skipped. If this happens while deploying a target contract all corresponding attack contracts are also skipped. If the target contract is successfully deployed but all attack contracts are skipped no fuzzing is conducted.

### 6.2.10 Creation of a Docker setup to enable parallel test runs

During the first few parallel test runs with multiple instances using the prototype a handful of problems occurred:

- By changing the system Solidity version for contract compilation (see 6.2.2) multiple fuzzer instances start to interfere with each other. This results in compile errors due wrong compiler version being used.

## 6. IMPLEMENTATION OF A PROTOTYPE

---

- Running only one instance of the test blockchain can lead to wrong account and contract states, causing *false positive* and *false negative* results.

While the second problem could be fixed by either using a different set of accounts or a dedicated blockchain instance per fuzzer instance, there is no really satisfying solution to the first problem when using a local setup. Therefore the decision was made to use Docker<sup>14</sup> ensuring a containerised, stable and shareable execution environment making parallel test runs possible regardless of the platform or device running the fuzzer.

---

<sup>14</sup>Link to the Docker documentation: <https://docs.docker.com/> - accessed on 16.05.2023

# Definition and evaluation of a Benchmarkset

In order to assess the prototype, a benchmark set was created. The set was then used to compare the results of the prototype to two other fuzzing tools: *ConFuzzius*[TIGS20] and *sFuzz*[NPS<sup>+</sup>20].

In this chapter first, an overview of how the benchmark set was formed is offered. Afterwards, the results of the three tools are evaluated and compared.

## 7.1 Need for a benchmark set

One part of this thesis is to evaluate the created prototype as well as *sFuzz* and *ConFuzzius* on a set of selected contracts and compare their findings. The benchmark set was provided by the supervisor of this work and colleagues from her research team.

## 7.2 Selection of contracts for the benchmark set

The selection of the different contracts was conducted as follows:

- First, all “verified contracts”<sup>1</sup> from *Etherscan.io*<sup>2</sup> where selected.
- From the selected contracts, a subset was chosen randomly with constraints on the block number and code length in order to have some code variety.

The final benchmark set contained 889 different contracts.

<sup>1</sup>This means that someone provided the source code to Etherscan, where a check was performed on if this source compiles to the bytecode that is deployed on the Ethereum main chain.

<sup>2</sup><https://etherscan.io/> - accessed 26.05.2023

### 7.2.1 Relevant vulnerability classes

*Vultron* claims to detect etherleaks, etherlocks, inter over-/underflows and exception disorders (see table 5.1 and section 5.6). These vulnerabilities roughly translate to the SWC categories: 101, 105, 106, 107, 112 and 115 making them the relevant vulnerability classes for choosing tools to compare with *Vultron*.

## 7.3 Evaluation of the prototype as well as *ConFuzzius* and *sFuzz* on the benchmark set

This section describes which tools were chosen for comparison with *Vultron*, the technical setup that was used to execute the tools and last but not least compares the result of the tools on the given benchmark set.

### 7.3.1 Selection criteria for *ConFuzzius* and *sFuzz*

*ConFuzzius*[TIGS20] and *sFuzz*[NPS<sup>+</sup>20] were chosen for comparison since they cover most of the vulnerability classes listed in 7.2.1, use fuzzing as main analysis technique and both tools also create exploits for found vulnerabilities.

### 7.3.2 Evaluation of *ConFuzzius* and *sFuzz* on the benchmark set

Since both, *ConFuzzius* and *sFuzz* are supported by the analysing framework SmartBugs<sup>3</sup> it was used for the evaluation of these two tools.

The following command was used to create the results for *ConFuzzius* and *sFuzz*:

```
python3 smartBugs.py -file ./contracts/  
  
--processes 16 --cpu-quota 30000 --mem-limit 48g  
  
--tools confuzzius sfuzz
```

### 7.3.3 Evaluation of the prototype on the benchmark set

The prototype was evaluated by running the following command:

```
docker run -v $(pwd)/output/logs:/App/vultron/logs  
  
-v $(pwd)/output/exploits:/App/vultron/exploits  
  
-v $contractPath:/App/vultron/contracts
```

---

<sup>3</sup>Link to the Github repo of SmartBugs: <https://github.com/smartbugs/smartbugs> - accessed 29.05.2023

```
-it vultron-local /App/vultron/contracts
```

This command spins up a docker container defined by the created Dockerfile (see 6.2.10). This file executes the following steps inside the spawned container:

- Setting up *Ubuntu 22.04*
- Installing *python3* and *python3-pip*
- Setting up *nodeJs 16*
- Installing *geth*
- Installing *Silther* (needed by *Vultron*)
- Installing *solc-select*
- Setting up all the dependencies needed by *Vultron* and the prototype
- Copying the benchmark set
- Executing the prototype on the benchmark set

#### 7.3.4 Results of the tool evaluations

Table 7.1 shows the result of the evaluation of the different tools on the benchmark set. The table is divided into five columns/categories.

- Contracts with error: This category contains contracts that were eliminated before any interaction with the tool happened (only relevant for tools used with *SmartBugs*).
- Failed or skipped contracts: Contracts that caused a failure of the tool or were skipped for arbitrary reasons fall into this category.
- Contracts with no findings: Contracts that finished processing but did not reveal any vulnerability/exploit are sorted into this category.
- Contracts with findings: Contracts that finished processing and revealed at least one vulnerability/exploit do qualify for this category.
- Total number of findings: This category was introduced because one contract can contain more than one vulnerability. This column displays the total number of vulnerabilities unique per contract (i.e. each vulnerability is only counted once per contract but is counted for every contract in which it occurs).

## 7. DEFINITION AND EVALUATION OF A BENCHMARKSET

---

Tool	Contracts with errors	Failed or skipped contracts	Contracts with no findings	Contracts with findings	Total number of findings
Prototype	0	288	587	14	148
Confuzzius	53	281	394	307	436
sFuzz	53	427	239	170	248

Table 7.1: Results per tool. Classified into “Contracts with errors”, “Failed or skipped contracts”, “Contracts with no findings”, “Contracts with findings” and “Total number of findings”

Both *sFuzz* and *ConFuzzius* encountered 53 contracts with errors while the *prototype* encountered zero contracts with errors.

All three tools have quite a high number of skipped or failed contracts. However, *sFuzz*(427) has nearly a third more “skipped or failed” contracts than *ConFuzzius*(281) and the *Vultron-based prototype*(288).

Again, all three tools encountered a high number of contracts where they did not find any vulnerability. Nevertheless, the *Vultron-based prototype*(587) has the highest number of contracts without findings, while *sFuzz*(239) has the lowest and *ConFuzzius*(394) finds itself in the middle.

*ConFuzzius*(307) has by far the most contracts with findings. *sFuzz*(170) comes second while the *Vultron-based prototype*(14) found the least amount of “contracts with findings” by far.

While the *Vultron-based prototype* only found findings in exactly 14 contracts, it found 148 different transaction sequences leading to these. *sFuzz* was able to find 248 findings in 170 different contracts. And again, with 436 findings *ConFuzzius* has discovered the most out of the three tested tools.

## 7.4 Interpretation of the evaluation results

This section aims to discuss and interpret the findings from sections 7.3.4.

### 7.4.1 Errors, failed and skipped contracts

As shown in Table 7.1 all three tools have a high ratio of failed contracts. That a contract fails, produces an error or is skipped can have a multitude of reasons. For example, the contract has no Solidity version specified, the deployment of the contracts fails or a contract uses unsupported data types (e.g. *ConFuzzius* does not support `tuple[]`). A high rate of non-successful contracts, however, does not give any clues about the number of potential findings. The *Vultron-based prototype* is a good example of this. Its zero erroneous contracts and 288 failed or skipped contracts mean a 68% success rate while testing the 889 contracts of the benchmark set (*ConFuzzius*: 62% success rate; *sFuzz*: 46% success rate). Making it the best tool regarding this statistic. Nonetheless, it did find the least amount of contracts with vulnerabilities as well as the least amount of total findings.

### 7.4.2 *ConFuzzius* and *sFuzz*

*ConFuzzius* and *sFuzz* classify their results using the SWC.<sup>4</sup> While *ConFuzzius* provides a vulnerability description as well as the SWC-code, *sFuzz* only provides a description

<sup>4</sup><https://swcregistry.io/> - accessed on 01.06.2023

SWC	Matching descriptions
101	Integer Overflow, Integer Underflow
104	Unhandled Exception, Exception Disorder
105	Leaking Ether
106	Unprotected Selfdestruct
107	Reentrancy
110	Assertion Failure
112	Unsafe Delegatecall
114	Transaction Order Dependency
116	Timestamp Dependency
120	Block Dependency, Block Number Dependency
126	Insufficient Gas Griefing, Gasless Send
124	Arbitrary Memory Access
132	Locking Ether

Table 7.2: Mapping from SWC-categories to their matching descriptions used by *sFuzz* and *ConFuzzius*

without a code. To make these two tools comparable a mapping from descriptions to SWC-codes was conducted. The chosen mapping is displayed in table 7.2.

Table 7.3 shows all findings of *ConFuzzius* and *sFuzz* after applying the aforementioned mapping. One can see, that *ConFuzzius* produces findings for 11 out of the 13 defined categories while *sFuzz* only produces findings for 6 categories. In every category both tools share *ConFuzzius* discovers more vulnerabilities than *sFuzz*.

So generally speaking *ConFuzzius* can be regarded as the better all-purpose tool. However, *sFuzz* has its biggest numbers in categories where *ConFuzzius* does not find any vulnerability: 116 Timestamp dependency - 56 findings and 126 Gasless send - 119 findings. Making it the better tool when one wants to target specifically those two categories.

### 7.4.3 Vultron-based prototype

Unfortunately, the *Vultron-based prototype* does not provide any category or description for its findings. Therefore making it impossible to compare it to the other tools regarding the SWC categories. Therefore, the *Vultron-based prototype* is compared to the other tools by looking at their findings and finding overlapping contracts.

As stated in 7.1 the *Vultron-based prototype* found 148 findings in 14 different contracts.

SWC	<i>ConFuzzius</i>	<i>sFuzz</i>
101	82	33
104	26	11
105	18	-
106	22	-
107	9	6
110	140	-
112	4	-
114	30	-
116	-	56
120	84	23
126	-	119
124	2	-
132	19	-

Table 7.3: Findings per SWC-category for *ConFuzzius* and *sFuzz*

Out of the fourteen contracts *ConFuzzius* finds vulnerabilities in 9 of them, *sFuzz* finds vulnerabilities in 6 of them and in 3 contracts only the *Vultron-based prototype* discovered any findings (see table 7.4). For the three contracts, the *Vultron-based prototype* found 41 different transaction sequences that lead to exploits. This means the *Vultron-based prototype* found 27,8% of its findings in 21,4% of the relevant contracts.

Overlaps with			
<i>ConFuzzius</i>	<i>sFuzz</i>	both	no overlaps
4	2	5	3

Table 7.4: This table lists the number of overlapping findings for the *Vultron-based prototype* compared to *sFuzz* and *ConFuzzius*

With 148 findings in 14 contracts, the *Vultron-based prototype* did perform the worst out of the three tools tested. Nevertheless, it did find exploits in three contracts where neither *ConFuzzius* nor *sFuzz* found anything.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Conclusio

## 8.1 Summary & Conclusion

This work first gave an overview of techniques used for finding vulnerabilities in smart contracts and automatically creating exploits for said vulnerabilities. Afterwards, existing “Exploit Generation Tools” for Ethereum smart contracts using these techniques were evaluated and compared on a theoretical level.

Later on, this work introduced a new prototype for exploit generation based on *Vultron*[WLL<sup>+</sup>20]. The new prototype was tested against a benchmark consisting of randomly chosen “verified contracts”<sup>1</sup> from “etherscan.io”. The results were then compared with *ConFuzzius*[TIGS20] and *sFuzz*[NPS<sup>+</sup>20] two tools that are also based on fuzzing.

Although all three tools are based on fuzzing, the evaluation of the benchmark set showed significant differences between them. *ConFuzzius* discovered findings in 11 different SWC-categories making it the “best” all-purpose tool. *sFuzz* found vulnerabilities in 6 different categories with most findings in the SWC-categories 116 and 126. Both categories that *ConFuzzius* does not cover at all. Additionally, the *Vultron-based prototype* (with 148 findings in 14 contracts the worst of the tested tools) discovered findings in three contracts where neither *ConFuzzius* nor *sFuzz* found anything.

With this insight, it can be concluded that both, the used analysis techniques and the quality of the implementation does make a huge difference regarding the effectiveness of a tool. Furthermore, to cover the most vulnerabilities possible it is not sufficient to use only a single tool, but rather a combination of multiple tools using different analysing techniques.

---

<sup>1</sup>See 7.1 for more information

Concerning the big amount of real-world value held by smart contracts and their quasi-immutable nature, serious analysis should be conducted by combining multiple tools that employ different analysing techniques.

### 8.2 Limitations & Future work

Since the tool introduced in this work was created as a prototype it still has some shortcomings that could be tackled by future works. For one thing, the prototype does not support fuzzing of functions or contracts that accept parameters of type `struct`, excluding contracts that use more complicated data structures.

Since the prototype is based on *Vultron*[WLL<sup>+</sup>20] it uses so-called attack contracts to interact with the contracts in question (see chapter 6). A possible improvement for the future would be to not only use these contracts for interacting with the target contracts but also to use private accounts (see 2.4 for reference).

Last but not least, this work used a benchmark set for the evaluation of the different tools. To get more expressive results the benchmark set could be replaced by an already reviewed ground truth.

# Voting Contract

## Solidity Sourcecode

```
1 // SPDX-License-Identifier: GPL-3.0
2 pragma solidity >=0.7.0 <0.9.0;
3
4 /**
5  * @title Ballot
6  * @dev Implements voting process along with vote delegation
7  */
8 contract Ballot {
9
10     struct Voter {
11         uint weight; // weight is accumulated by delegation
12         bool voted; // if true, that person already voted
13         address delegate; // person delegated to
14         uint vote; // index of the voted proposal
15     }
16
17     struct Proposal {
18         // If you can limit the length to a certain number of bytes,
19         // always use one of bytes1 to bytes32 because they are much cheaper
20         bytes32 name; // short name (up to 32 bytes)
21         uint voteCount; // number of accumulated votes
22     }
23
24     address public chairperson;
25
26     mapping(address => Voter) public voters;
27
28     Proposal[] public proposals;
29
30     /**
31     * @dev Create a new ballot to choose one of 'proposalNames'.
```

## A. VOTING CONTRACT

```
32 * @param proposalNames names of proposals
33 */
34 constructor(bytes32[] memory proposalNames) {
35     chairperson = msg.sender;
36     voters[chairperson].weight = 1;
37
38     for (uint i = 0; i < proposalNames.length; i++) {
39         // 'Proposal({...})' creates a temporary
40         // Proposal object and 'proposals.push(...)'
41         // appends it to the end of 'proposals'.
42         proposals.push(Proposal({
43             name: proposalNames[i],
44             voteCount: 0
45         }));
46     }
47 }
48
49 modifier isChairperson () {
50     require(
51         msg.sender == chairperson,
52         "Only chairperson can give right to vote."
53     );
54     _;
55 }
56
57 /**
58 * @dev Give 'voter' the right to vote on this ballot. May only be called
59 *     by 'chairperson'.
60 * @param voter address of voter
61 */
62 function giveRightToVote(address voter) public isChairperson {
63     require(
64         !voters[voter].voted,
65         "The voter already voted."
66     );
67     require(voters[voter].weight == 0);
68     voters[voter].weight = 1;
69 }
70
71 /**
72 * @dev Delegate your vote to the voter 'to'.
73 * @param to address to which vote is delegated
74 */
75 function delegate(address to) public {
76     Voter storage sender = voters[msg.sender];
77     require(!sender.voted, "You already voted.");
78     require(to != msg.sender, "Self-delegation is disallowed.");
79
80     while (voters[to].delegate != address(0)) {
81         to = voters[to].delegate;
82
83         // We found a loop in the delegation, not allowed.
84         require(to != msg.sender, "Found loop in delegation.");
85     }
86 }
```

```
84     }
85     sender.voted = true;
86     sender.delegate = to;
87     Voter storage delegate_ = voters[to];
88     if (delegate_.voted) {
89         // If the delegate already voted,
90         // directly add to the number of votes
91         proposals[delegate_.vote].voteCount += sender.weight;
92     } else {
93         // If the delegate did not vote yet,
94         // add to her weight.
95         delegate_.weight += sender.weight;
96     }
97 }
98
99 /**
100  * @dev Give your vote (including votes delegated to you) to proposal '
101     proposals[proposal].name'.
102  * @param proposal index of proposal in the proposals array
103  */
104 function vote(uint proposal) public payable {
105     Voter storage sender = voters[msg.sender];
106     require(sender.weight != 0 || msg.value >= 1 ether, "Has no right to
107         vote");
108     require(!sender.voted, "Already voted.");
109     sender.voted = true;
110     sender.vote = proposal;
111
112     // If 'proposal' is out of the range of the array,
113     // this will throw automatically and revert all
114     // changes.
115     proposals[proposal].voteCount += sender.weight;
116 }
117
118 /**
119  * @dev Computes the winning proposal taking all previous votes into
120     account.
121  * @return winningProposal_ index of winning proposal in the proposals
122     array
123  */
124 function winningProposal() public view
125 returns (uint winningProposal_)
126 {
127     uint winningVoteCount = 0;
128     for (uint p = 0; p < proposals.length; p++) {
129         if (proposals[p].voteCount > winningVoteCount) {
130             winningVoteCount = proposals[p].voteCount;
131             winningProposal_ = p;
132         }
133     }
134 }
```

## A. VOTING CONTRACT

---

```
133 * @dev Calls winningProposal() function to get the index of the winner
134 * @return winnerName_ the name of the winner
135 */
136 function winnerName() public view
137 returns (bytes32 winnerName_)
138 {
139     winnerName_ = proposals[winningProposal()].name;
140 }
141 }
```

### Application Binary Interface

```
1 [
2 {
3     "inputs": [
4         {
5             "internalType": "bytes32[]",
6             "name": "proposalNames",
7             "type": "bytes32[]"
8         }
9     ],
10    "stateMutability": "nonpayable",
11    "type": "constructor"
12 },
13 {
14    "inputs": [],
15    "name": "chairperson",
16    "outputs": [
17        {
18            "internalType": "address",
19            "name": "",
20            "type": "address"
21        }
22    ],
23    "stateMutability": "view",
24    "type": "function"
25 },
26 {
27    "inputs": [
28        {
29            "internalType": "address",
30            "name": "to",
31            "type": "address"
32        }
33    ],
34    "name": "delegate",
35    "outputs": [],
36    "stateMutability": "nonpayable",
37    "type": "function"
38 },
39 {
```

```
40     "inputs": [  
41     {  
42         "internalType": "address",  
43         "name": "voter",  
44         "type": "address"  
45     }  
46     ],  
47     "name": "giveRightToVote",  
48     "outputs": [],  
49     "stateMutability": "nonpayable",  
50     "type": "function"  
51 },  
52 {  
53     "inputs": [  
54     {  
55         "internalType": "uint256",  
56         "name": "",  
57         "type": "uint256"  
58     }  
59     ],  
60     "name": "proposals",  
61     "outputs": [  
62     {  
63         "internalType": "bytes32",  
64         "name": "name",  
65         "type": "bytes32"  
66     },  
67     {  
68         "internalType": "uint256",  
69         "name": "voteCount",  
70         "type": "uint256"  
71     }  
72     ],  
73     "stateMutability": "view",  
74     "type": "function"  
75 },  
76 {  
77     "inputs": [  
78     {  
79         "internalType": "uint256",  
80         "name": "proposal",  
81         "type": "uint256"  
82     }  
83     ],  
84     "name": "vote",  
85     "outputs": [],  
86     "stateMutability": "payable",  
87     "type": "function"  
88 },  
89 {  
90     "inputs": [  
91     {  
92         "internalType": "address",
```

## A. VOTING CONTRACT

---

```
93     "name": "",
94     "type": "address"
95   }
96 ],
97 "name": "voters",
98 "outputs": [
99   {
100     "internalType": "uint256",
101     "name": "weight",
102     "type": "uint256"
103   },
104   {
105     "internalType": "bool",
106     "name": "voted",
107     "type": "bool"
108   },
109   {
110     "internalType": "address",
111     "name": "delegate",
112     "type": "address"
113   },
114   {
115     "internalType": "uint256",
116     "name": "vote",
117     "type": "uint256"
118   }
119 ],
120 "stateMutability": "view",
121 "type": "function"
122 },
123 {
124   "inputs": [],
125   "name": "winnerName",
126   "outputs": [
127     {
128       "internalType": "bytes32",
129       "name": "winnerName_",
130       "type": "bytes32"
131     }
132   ],
133   "stateMutability": "view",
134   "type": "function"
135 },
136 {
137   "inputs": [],
138   "name": "winningProposal",
139   "outputs": [
140     {
141       "internalType": "uint256",
142       "name": "winningProposal_",
143       "type": "uint256"
144     }
145   ],
```

---

```
146     "stateMutability": "view",  
147     "type": "function"  
148   }  
149 ]
```



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Listings

2.1	Example of a simple Solidity contract that offers the functionality to store an integer-value[Pro]. . . . .	14
2.2	This listing shows how the Solidity pragma is defined and how custom types (structs) can be created[Fou21]. . . . .	15
2.3	Example of how the constructor in Solidity is often used to define the owner of a contract and to initialise state variables[Fou21]. . . . .	15
2.4	This listing shows how modifiers are defined and how they can be applied[Fou21].	16
2.5	Example of how a voting function - that accepts votes from chosen voters as well as "bought" votes - can be implemented[Fou21]. . . . .	17
2.6	Excerpt of the Application Binary Interface belonging to the voting contract used in 2.4.4 . . . . .	19
3.1	Shows a contract where a external party can trigger a selfdestruct after it pays a certain amount. Since this is the intended behaviour of the contract it is no vulnerability. . . . .	22
3.2	Smart contract snippet that shows code that has no effect, but leads to unintended behaviour[Clac]. . . . .	23
6.1	Example showing how a constructor could be implemented prior to Solidity 0.5.0 . . . . .	52
6.2	Example showing how a constructor can be implemented since Solidity 0.7.0 . . . . .	52
6.3	Code snippet showing logic to extract the Solidity version out of a contract.	53
6.4	Compile options passed to solc-js. . . . .	53
6.5	Final output used for creating and deploying contracts. . . . .	53
6.6	Snippet showing how <i>Vultron</i> starts its <i>silent-server</i> and starts the fuzzing process.[WLL <sup>+</sup> 20] . . . . .	54
6.7	Snippet showing how <i>Vultron</i> extends the default Tracerecorder of <i>Aleth</i> to transport the recorded execution trace to the silent-server. <sup>1</sup> . . . . .	55
6.8	Tracer config used to extract execution traces from geth in the same format as previously defined by <i>Aleth</i> . . . . .	56
6.9	Excerpt of an exploit file generate by <i>Vultron</i> [WLL <sup>+</sup> 20]. . . . .	57
		79

6.10	Output file generated by the new logic. Showing all necessary information to reproduce the found exploits. . . . .	58
------	--	----

## List of Figures

2.1	Basic schema of a “Blockchain” drawn after [Nak09]. . . . .	6
2.2	Showing the schema of a Merkle tree inspired by [Nak09] and [But14]. Each block header stores only the root hash of the underlying Merkle tree. Modifying something down the tree will change all superordinate hashes and therefore invalidate the whole tree and block. . . . .	7
2.3	Representation of a supply chain using smart contracts. Once the buyer has issued an order, interaction and payment between the different parties works completely automatic and without intervention of intermediaries . . . . .	11
4.1	Example of a control flow graph derived from a smart contract. Adapted from [CCCP21]. . . . .	28
4.2	A simple program next to a visualisation of possible program paths and their corresponding path conditions at the end. Green-coloured formulas mean that they are satisfiable. The red ones are not satisfactory[Cho15]. . . . .	29
4.3	A simple control property graph. Depending on the origin of the relation to edges use different colours.[YGAR14] . . . . .	30
4.4	A simple program next to a generated backward program slice. Taken from [SRH11] . . . . .	32
5.1	Architectural overview of EthFuzz. Adapted from [Ash21] . . . . .	45

# List of Tables

5.1	This table shows which analysis techniques the tools use and which vulnerabilities they claim to detect. To keep the table more readable the techniques as well as the vulnerabilities are abbreviated to two-letter identifiers. . . .	47
7.1	Results per tool. Classified into “Contracts with errors”, “Failed or skipped contracts”, “Contracts with no findings”, “Contracts with findings” and “Total number of findings” . . . . .	64
7.2	Mapping from SWC-categories to their matching descriptions used by <i>sFuzz</i> and <i>ConFuzzius</i> . . . . .	66
7.3	Findings per SWC-category for <i>ConFuzzius</i> and <i>sFuzz</i> . . . . .	67
7.4	This table lists the number of overlapping findings for the <i>Vultron-based prototype</i> compared to <i>sFuzz</i> and <i>ConFuzzius</i> . . . . .	67



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar  
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Bibliography

- [AFL17] American fuzzy lop (afl). [https://lcamtuf.coredump.cx/afl/technical\\_details.txt](https://lcamtuf.coredump.cx/afl/technical_details.txt), 2017. Accessed: 2022-10-21.
- [Ash21] Mohammadreza Ashouri. An extensive security analysis on ethereum smart contracts. In Joaquin Garcia-Alfaro, Shujun Li, Radha Poovendran, Hervé Debar, and Moti Yung, editors, *Security and Privacy in Communication Networks*, pages 144–163, Cham, 2021. Springer International Publishing.
- [BBGM12] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A taint based approach for smart fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 818–825, 2012.
- [BCD<sup>+</sup>18] Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), may 2018.
- [BCR21] Marcel Boehme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Softw.*, 38(3):79–86, 2021.
- [BMC<sup>+</sup>15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121, 2015.
- [Böh06] Rainer Böhme. A comparison of market approaches to software vulnerability disclosure. In Günter Müller, editor, *Emerging Trends in Information and Communication Security*, pages 298–311, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [BPNR17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS ’17*, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

- [But14] Vitalik Buterin. Ethereum whitepaper. <https://ethereum.org/en/whitepaper/>, 2014. Accessed: 2022-03-21.
- [BV15] Vitalik Buterin Buterin and Fabian Vogelsteller. Eip-20: Token standard. <https://eips.ethereum.org/EIPS/eip-20>, Nov 2015.
- [CCCP21] Filippo Contro, Marco Crosara, Mariano Ceccato, and Mila Dalla Preda. Ethersolve: Computing an accurate control-flow graph from ethereum bytecode. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 127–137, 2021.
- [Cha83] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology*, pages 199–203, Boston, MA, 1983. Springer US.
- [Cho15] Stephen Chong. Symbolic execution. <https://groups.seas.harvard.edu/courses/cs252/2015fa/lectures/Lec07-SymExec.pdf>, 2015. Accessed: 2022-10-21.
- [Claa] Smart Contract Weakness Classification. Swc-107: Reentrancy. <https://swcregistry.io/docs/SWC-107>. Accessed: 2022-10-14.
- [Clab] Smart Contract Weakness Classification. Swc-131: Presence of unused variables. <https://swcregistry.io/docs/SWC-131>. Accessed: 2022-10-14.
- [Clac] Smart Contract Weakness Classification. Swc-135: Code with no effects. <https://swcregistry.io/docs/SWC-135>. Accessed: 2022-10-14.
- [coi] Total cryptocurrency market cap. <https://coinmarketcap.com/charts/>. Accessed: 2022-01-04.
- [CPNX20] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Comput. Surv.*, 53(3), jun 2020.
- [dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [DN18] Ardit Dika and Mariusz Nowostawski. Security vulnerabilities in ethereum smart contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (Green-Com) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 955–962, 2018.

- [DP17] Massimo Di Pierro. What is the blockchain? *Computing in Science Engineering*, 19(5):92–95, 2017.
- [Enea] Common Weakness Enumeration. Cwe-1164: Irrelevant code. <https://cwe.mitre.org/data/definitions/1164.html>. Accessed: 2022-10-14.
- [Enub] Common Weakness Enumeration. Cwe-710: Improper adherence to coding standards. <https://cwe.mitre.org/data/definitions/710.html>. Accessed: 2022-10-14.
- [Enuc] Common Weakness Enumeration. Cwe glossary. <https://cwe.mitre.org/documents/glossary/index.html#Vulnerability>. Accessed: 2022-10-13.
- [Enud] Common Weakness Enumeration. Cwe glossary. <https://cwe.mitre.org/documents/glossary/index.html#Weakness>. Accessed: 2023-06-12.
- [eth] History of ethereum. <https://ethereum.org/en/history/>. Accessed: 2022-11-07.
- [FGG19] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15, 2019.
- [FLWvG07] G. Fischer, J. Lusiardi, and J. Wolff von Gudenberg. Abstract syntax trees - and their role in model driven software development. In *International Conference on Software Engineering Advances (ICSEA 2007)*, pages 38–38, 2007.
- [Fou16a] Ethereum Foundation. Application binary interface. <https://docs.soliditylang.org/en/latest/abi-spec.html>, 2016. Accessed: 2022-04-21, Revision 55917405.
- [Fou16b] Ethereum Foundation. Solidity. <https://docs.soliditylang.org/en/latest/index.html>, 2016. Accessed: 2022-04-21, Revision 55917405.
- [Fou21] Ethereum Foundation. Solidity by example. <https://docs.soliditylang.org/en/latest/solidity-by-example.html>, 2016-2021. Accessed: 2022-05-06.
- [FTB19] Yu Feng, Emina Torlak, and Rastislav Bodík. Precise attack synthesis for smart contracts. *CoRR*, abs/1902.06067, 2019.

- [glo] Global cryptocurrency market charts. <https://coinmarketcap.com/charts/>.
- [God20] Patrice Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, 63(2):70–76, 2020.
- [HR92] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, page 392–411, New York, NY, USA, 1992. Association for Computing Machinery.
- [JLC18] Bo Jiang, Ye Liu, and W.K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 259–269, 2018.
- [Kam95] Mariam Kamkar. An overview and comparative classification of program slicing techniques. *Journal of Systems and Software*, 31(3):197–214, 1995.
- [KCR21] Nikita Karandikar, Antorweep Chakravorty, and Chunming Rong. Blockchain based transaction system with fungible and non-fungible tokens for a community-based energy infrastructure. *Sensors*, 21(11), 2021.
- [KR18] Johannes Krupp and Christian Rossow. teEther: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1317–1333, Baltimore, MD, August 2018. USENIX Association.
- [Lib17] Libfuzzer – a library for coverage-guided fuzz testing. <https://l1vm.org/docs/LibFuzzer.html>, 2017. Accessed: 2022-10-21.
- [M21] Laura M. Token vs coin: What's the difference? <https://www.bitdegree.org/crypto/tutorials/token-vs-coin>, Nov 2021.
- [MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990.
- [Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 03 2009.
- [NBF<sup>+</sup>16] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, USA, 2016.
- [NGHS17] Michael Nofer, Peter Gomber, Oliver Hinz, and Dirk Schiereck. Blockchain. *Business & Information Systems Engineering*, 59(3):183–187, Jun 2017.

- [NKS<sup>+</sup>18] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*, pages 653–663, 2018.
- [NPS<sup>+</sup>20] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. Sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20*, page 778–788, New York, NY, USA, 2020. Association for Computing Machinery.
- [PC90] A. Podgurski and L.A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.
- [PL21] Daniel Perez and Benjamin Livshits. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *USENIX Security Symposium*, 2021.
- [Pro] Remix Project. Remix ide - example contract. <https://remix.ethereum.org/>. Accessed: 2022-05-06.
- [Ram21] Heidelinde Rameder. Systematic review of ethereum smart contract security vulnerabilities, analysis methods and tools, 2021. Master thesis, TU Wien.
- [Sny19] Hannah Snyder. Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, 104:333–339, 2019.
- [SRH11] N Sasirekha, A Edwin Robert, and Dr M Hemalatha. Program slicing techniques and its applications. *arXiv preprint arXiv:1108.1352*, 2011.
- [SSD<sup>+</sup>21] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing, and Baoxu Liu. Evil under the sun: Understanding and discovering attacks on ethereum decentralized applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1307–1324. USENIX Association, August 2021.
- [Sza96] Nick Szabo. Smart contracts: building blocks for digital markets. *EX-TROPY: The Journal of Transhumanist Thought*, (16), 18(2):28, 1996.
- [TIGS20] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. Towards smart hybrid fuzzing for smart contracts. *CoRR*, abs/2005.12156, 2020.
- [TIGS21] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 103–119, 2021.

- [TKA<sup>+</sup>22] Arianna Trozze, Josh Kamps, Eray Arda Akartuna, Florian J. Hetzel, Bennett Kleinberg, Toby Davies, and Shane D. Johnson. Cryptocurrencies and future financial crime. *Crime Science*, 11(1):1, Jan 2022.
- [W<sup>+</sup>14] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.
- [Wil97] David S Wile. Abstract syntax from concrete syntax. In *Proceedings of the 19th international conference on Software engineering*, pages 472–480, 1997.
- [WLL<sup>+</sup>20] Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. Oracle-supported dynamic exploit generation for smart contracts. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2020.
- [WSX<sup>+</sup>21] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 8(2):1133–1144, 2021.
- [YGAR14] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, 2014.
- [YHKC<sup>+</sup>16] Jesse Yli-Huumo, Deokyoon Ko, Sujin Choi, Sooyong Park, and Kari Smolander. Where is current research on blockchain technology?—a systematic review. *PLOS ONE*, 11(10):1–27, 10 2016.
- [ZWLM20] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. Ethploit: From fuzzing to efficient exploit generation against smart contracts. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 116–126, 2020.
- [ZXD<sup>+</sup>20] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems*, 105:475–491, 2020.