Imperial College London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Centaur: An EVM-Based Blockchain Vulnerability Analysis Framework

Author:
Marcos-Antonios Charalambous

Supervisor:
Arthur Gervais

Second Marker:
William Knottenbelt

Submitted in partial fulfillment of the requirements for the MSc degree in Computing (Software Engineering) of Imperial College London

September 2022

Acknowledgements

I want to express my deepest gratitude to my thesis supervisor, Assistant Professor Dr Arthur Gervais, for his pivotal guidance, consistent encouragement, support and valuable advice that helped me complete and accomplish this project. During the passage of my thesis, Dr Gervais' interest, enthusiasm and expertise in the field of Decentralised Systems and Security were undoubtedly a key source of inspiration. He sparked my interest in Blockchain, which led to the undertaking of this ambitious thesis.

Furthermore, it would go amiss if I did not convey gratitude to my fellow members of the Decentralized Systems And Security group, especially PhD student Stefanos Chaliasos. I acknowledge that his invaluable guidance and positive input made this endeavour an exciting but challenging experience.

Moreover, I would like to thank the authors of the SmartBugs framework for providing guidance on the extension of their framework with new analysis tools and its modification for working on Bytecode.

Needless to say, I must thank all my lecturers from whom I received a first-class learning experience gaining invaluable knowledge and enabling me to achieve a Master's degree in Computing after my year-long study at the Department of Computing at Imperial College London.

Finally, it must be stated that I owe an incredible amount of gratitude to my friends and family for their unswerving support during the many twists and turns on the road to fruition that serves as a milestone on my academic journey.

Abstract

Blockchain is gaining steady recognition mostly for its pivotal role in cryptocurrency systems, such as Bitcoin, by keeping a decentralised record of transactions. Its data structure is what makes it unique. It is an append-only distributed ledger that removes intermediates, conducts secure transactions, and tracks anything of value. Smart contracts, which are programs deployed on the blockchain, are an important development that led to the adoption of Ethereum, the first blockchain with smart contract capabilities.

Like other types of programs, smart contracts contain vulnerabilities that can be exploited. The fact that smart contracts handle large amounts of money makes them an attractive target to malicious attackers who are incentivised to exploit their vulnerabilities for financial gain.

Although smart contract vulnerability research has evolved significantly on Ethereum, other EVM-compatible chains invariably have received limited attention. Little to no work has been undertaken to compare and analyse the vulnerabilities of different chains until now. Therefore, we devised a process that attempts to collect and probe smart contracts for bugs so a universal approach can be adopted to limit vulnerabilities in smart contracts.

This paper presents an empirical study on two EVM-based blockchains, namely Ethereum and Binance Smart Chain. It explores the existence of vulnerabilities in deployed smart contracts in these two chains using smart contract automated analysis tools for EVM bytecode. Our codebase artefact is encapsulated into the Centaur framework. The framework also extends the SmartBugs framework for analysing the dataset of smart contracts bytecodes using multiple analysis tools, and it is purposely easily extendable to support other EVM chains.

After executing Centaur on 4,000 blocks from Ethereum and Binance Smart Chain, we crafted a dataset of 17,592 deployed smart contracts. Once these smart contracts were filtered by keeping only the ones with unique bytecodes and those utilised in practice, we used Centaur to run nine state-of-the-art analysis tools on them and extract any reported vulnerabilities. Vulnerabilities were classified according to an extended version of the DASP10 taxonomy and the Smart Contract Weakness Classification registry. The process took more than five days to complete.

Our findings reveal that smart contracts deployed on Ethereum contain more vulnerabilities when compared to smart contracts deployed on Binance Smart Chain. In addition, there are balances in danger of being stolen by attackers that leverage these vulnerabilities, although the number is limited. Finally, some smart contracts are deployed on both EVM blockchains, and each chain has many duplicate smart contracts.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Research Questions . . . 3
- 1.3 Methodology . . . 4
- 1.4 Contributions . . . 4
- 1.5 Project Outline . . . 5

2 Background **6**
- 2.1 Blockchain . . . 6
- 2.2 Smart Contracts and EVM . . . 7
- 2.3 Systematisation of Vulnerabilities . . . 9
- 2.4 Using Analysis Tools to Detect Bugs . . . 14

3 Related Work **17**
- 3.1 Vulnerabilities in Smart Contracts . . . 17
- 3.2 Analysing Smart Contracts . . . 18
- 3.3 Smart Contract Datasets . . . 18
- 3.4 Empirical Studies . . . 19

4 Methodology **21**
- 4.1 Finding Automated Analysis Tools . . . 21
  - 4.1.1 Tool Description . . . 23
- 4.2 Datasets Design . . . 26
- 4.3 Framework Execution . . . 28
- 4.4 Step-by-Step Analysis Procedure . . . 29
  - 4.4.1 Data Collection . . . 29
  - 4.4.2 Data Storage . . . 30
  - 4.4.3 Dataset Processing . . . 30
  - 4.4.4 Results Analysis . . . 31

5 Implementation **32**
- 5.1 Database Deployment . . . 32
- 5.2 Random Sampling . . . 33
- 5.3 Block Crawler . . . 34
  - 5.3.1 Collecting Contract Addresses . . . 35
- 5.4 Blockchain Explorer Crawler . . . 36

5.5 Analysis 38

6 Results **41**
6.1 Experimentation Details 41
6.2 Dataset Analysis 42
6.2.1 Self-Destructed Contracts 42
6.2.2 Gas Token Contracts 43
6.3 RQ1: Vulnerability State 44
6.3.1 Analysis Tool Discussion 50
6.4 RQ2: Potential Balance at Stake 52
6.5 RQ3: Smart Contract Duplication 53

7 Discussion **55**
7.1 Limitations 55
7.2 Future Work 56
7.3 Threats to Validity 57

8 Conclusion **59**
8.1 Ethical Conduct 60

A Database **72**

B Vulnerabilities Reported **73**

List of Figures

2.1 Structure of a block and a smart contract on blockchain. 7
2.2 Reentrancy bug. 10
2.3 Access Control bug. 10
2.4 Arithmetic Issues bug. 11
2.5 Unchecked Return Values For Low Level Calls bug. 11
2.6 Denial of Service bug. 11
2.7 Time Manipulation bug. 12
2.8 Assert Violation bug. 13
2.9 Delegate Call To Untrusted Contract bug. 13
2.10 Write to Arbitrary Storage Location bug. 14

4.1 Methodology pipeline for including a tool in our study. 22
4.2 Analysis Pipeline Overview. 31

5.1 ER diagram of our *analysis* database. 33
5.2 CREATE contract transaction in Ethereum. 36
5.3 Control Flow Chart of Centaur's execution. 40

6.1 Proportion of self-destructed contracts on each chain. 44
6.2 Gas Token Contract Numbers. 45
6.3 Proportions of Chi gastoken and CasToken.io for each chain. 45
6.4 Example of contract created by Chi gastoken. 45
6.5 Average Vulnerabilities Reported For Each Class. 51

A.1 docker-compose YAML file for deploying MariaDB. 72

B.1 Ethereum-deployed smart contract vulnerabilities reported. 73
B.2 BSC-deployed smart contract vulnerabilities reported. 73
B.3 Vulnerabilities per 100 contracts for both chains. 74

List of Tables

2.1 Vulnerability classes and their level of introduction. 9

4.1 List of smart contract security analysis tools. 24
4.2 Tools that pass the inclusion criteria. 25
4.3 Analysis tools and vulnerability classes mapping. 27

6.1 Dataset statistics. 43
6.2 Execution time for the data collection phase. 46
6.3 Execution time for the result analysis phase. 47
6.4 BSC vulnerability state. 48
6.5 Ethereum vulnerability state. 49
6.6 Balance of deployed smart contracts. 52
6.7 Smart contracts that exists on both chains. 53

Chapter 1

Introduction

Blockchain is a pioneering technology that has caught the imagination of researchers and developers due to its immutable and transparent nature. It is an append-only data structure run by mutually dubious nodes, governed by a consensus protocol, in a Peer-To-Peer (P2P) network. A blockchain facilitates the recording of transactions into blocks, linking these blocks securely using cryptography and tracking anything of value over a decentralised network [1]. It is a system of recording information that makes it almost impossible to amend. A blockchain acts as a digital ledger of transactions duplicated and distributed across an entire network of computer systems. It was introduced in 2008 by Satoshi Nakamoto as the fundamental technology of Bitcoin [2].

Its most sought-after application is in the field of Decentralised Finance (DeFi) especially decentralised digital currencies (commonly known as cryptocurrencies), with their global market cap and Total Value Locked (TVL) reaching $1.08 Trillion [3] and $62.08 Billion [4] respectively in late August 2022. DeFi gained wider acceptance and public support due to its mandate of disrupting the traditional finance system by removing the need to rely on trusted third-party intermediaries and trust mechanisms while remaining democratic in its accessibility and providing parallel services (e.g., borrowing, fund management, payment solutions). Cryptocurrencies utilise the blockchain as a public ledger, recording any currency transfers and thus avoiding money being double spent.

A major platform that utilises blockchain technology is Ethereum [5]. Ethereum provides a decentralised ecosystem for developers to create decentralised applications (dapps). Dapps are applications that run on more than one node and often consist of a frontend (user interface) for interactions and a backend for data storage. The backend logic is outsourced to so-called smart contracts, self-enforceable pieces of code representing contractual agreements. An example of a decentralised application that attracted so much attention at the height of its popularity, disrupting transactions on the Ethereum main network, is *CryptoKitties* (a digital game centred around collectable virtual cats) [6].

The growing adoption of smart contracts, and the fact they can store vast amounts

of money in virtual currencies, has led to the allure and financial incentivisation of malicious adversaries. Hackers attempt to profit by searching for vulnerabilities in smart contracts that can be exploited. Examples include TheDAO exploit [7, 8] and the Parity wallet bug [9] where large sums of money were lost due to bugs ($60M and $280M respectively).

Also, the permissionless nature of smart contract platforms, like Ethereum plays a key role in their security as arbitrary, mutually untrusted participants (miners) can join and manipulate the smart contracts' intended execution flow for their benefit. For instance, they can alter the sequence of transactions and block timestamps that lead to financial losses for the legal beneficiary.

Research proves that smart contract security has not received equal attention as smart contracts themselves [10, 7, 11, 12]. Durieux *et al*. [12] had discovered that 97% of unique smart contracts deployed on Ethereum are flagged as buggy when nine state-of-the-art automated security analysis tools analysed their source code. This, however, indicates these tools report a high number of false positives in their findings, as there are some inconsistencies between their outcomes.

Apart from Ethereum, other chains have gradually gained greater recognition. Not far behind Ethereum is the Binance Smart Chain [13, 14], also known as the BSC chain, based on the Ethereum Virtual Machine (EVM) like Ethereum. In fact, it is the second biggest EVM-based blockchain after Ethereum, with a TVL of $36.38 Billion[1] and $5.35 Billion[2] for Ethereum and BSC respectively.

The Binance Smart Chain should not be confused with the Binance Chain, which is not EVM compatible and does not support smart contracts. Binance Chain's primary objective is to provide rapid, decentralised trading, whereas BSC focuses on decentralised apps. A dual-chain architecture facilitates the interoperability of these two chains. Expectantly, the biggest dapp deployed on Binance Chain is the popular Binance DEX [15].

Since both Ethereum and Binance Smart Chain are EVM compatible, it means the same smart contracts can be deployed on both chains. It also means they are susceptible to the same type of smart contract vulnerabilities. However, to our surprise, BSC is given far less attention from the research community regarding smart contract vulnerabilities, especially compared to Ethereum.

1.1 Motivation

Blockchain is a lively research field, with new analysis tools frequently published to make smart contract development less troublesome. However, in our experience,

[1] https://defillama.com/chain/Ethereum
[2] https://defillama.com/chain/BSC

most studies in the field, if not all, the focus is exclusively on Ethereum. Although Ethereum is the most popular framework for smart contract deployment, other EVM-based blockchains like BSC and Avalanche (AVAX) [16] are witnessing a steady increase in TVL and adoption.

As they are EVM compatible, in theory, it involves the same Solidity source code and bytecode, dapp projects, and the same automated analysis tools can be deployed as-is on multiple chains without extra effort. In other words, the rich Ethereum sphere can be transported to the BSC, along with its vulnerabilities, however.

To the best of our knowledge, in this work, we performed the first empirical study on Ethereum and Binance Smart Chain combined. We explored their state of vulnerability using automated analysis tools. The work presented by Durieux *et al*. [12] prompted us to perform a similar study but on two chains instead of one (Ethereum) while we leveraged the SmartBugs open-source framework they developed.

Specifically, we investigated how widespread vulnerabilities are in deployed smart contracts on the two chains to compare their overall vulnerability states. By doing this, we acquire insights that would help (1) understand which developers of the two blockchains are more security-oriented, (2) practitioners to avoid common smart contract vulnerability patterns, (3) researchers and analysis tool developers to design and implement better approaches to detect and deter smart contract bugs, (4) how duplication happens in each chain and across chains, and (5) the total potential balance of each chain at stake due to vulnerable smart contracts. To this end, we will attempt to answer the research questions posed in the following section.

1.2 Research Questions

The goal of this study is to seek answers to following research questions:

RQ1: **How do the current states across two EVM-based chains compare in terms of vulnerabilities?**

Our first and primary research question is that we would like to investigate the vulnerabilities present on two EVM-compatible chains, namely Ethereum and Binance Smart Chain. We would consider detecting the most popular and common vulnerabilities in deployed smart contracts, which the selected automated analysis tools can detect and the consensus between them on these vulnerabilities. We used the *DASP10* [17] and *SWC Registry* [18] for classifying vulnerabilities (cf. Section 2.3).

RQ2: **What is the correlation between the number of vulnerable smart contracts and their total balance?**

With this research question, we aim to determine how much ETH and BNB are at stake from finding buggy smart contracts and summing their balance. If the smart

contract has one or more vulnerabilities, any currency it stores could be drained or become unusable by attackers.

RQ3: **How many smart contracts are identical on each chain, and how many are deployed on both chains?**

Here we sought to detect the proportions of unique and duplicate smart contracts deployed on each chain mentioned above. Also, since no extra effort is required to modify a smart contract to be deployed on other EVM-compatible chains, we expect a greater percentage of the same smart contracts to be deployed across both chains we investigated.

1.3 Methodology

To answer the above questions, we assembled a list of all the smart contract automated analysis tools we could find (cf. Table 4.1). From these tools, we shorten the list to a few that would fulfil the requirements recorded in Section 4.1. One of the criteria was for the tools to work on bytecode. This choice is essential as we have archive nodes on both chains that give us access to all historical data, including the bytecode of smart contracts.

After crawling blocks throughout the history of Ethereum and BSC chains by performing random sampling (cf. Section 5.2), we created the dataset thoroughly presented in Section 4.2. We also added several metadata for every smart contract in the dataset. After the dataset was finalised, we executed it on a modified version of SmartBugs that works on bytecode with the tools selected. The results were then parsed by various analysis tools parsing scripts to output any potential vulnerabilities. The parser's output maps vulnerabilities to an extended version of the DASP10 taxonomy and SWC registry. Afterwards, we manually studied the outcome to answer our research questions.

1.4 Contributions

This project makes the following contributions:

- We present, to the extent of our knowledge, the first empirical analysis of deployed smart contract bytecode vulnerabilities for BSC and Ethereum. We executed nine state-of-the-art automated analysis tools for the study on 473 unique bytecodes for more than five days.

- We provide the infrastructure used to conduct this study, namely the Centaur security analysis framework, to encourage replicating our work on more EVM-compatible chains.

- By manually examining the vulnerabilities reported by the nine analysis tools, we introduced an extended version of the DASP10 taxonomy that includes the 18 distinct vulnerability classes reported by these tools.
- The technical artefact of this study, namely the Centaur framework and corpus of 17,592 bytecodes collected after crawling 4,000 blocks on Ethereum and BSC are released as open source to foster further research in the field of blockchain and secure smart contract development.

Summary of Findings: The most important conclusions of our study are the following: (1) smart contracts deployed on Ethereum contain 3.95 times more vulnerabilities per 100 contracts than BSC-deployed smart contracts, (2) vulnerability classes Arithmetic Issues and Reentrany are in the top three most common vulnerabilities for both chains, (3) seven vulnerabilities groups are more prevalent on Ethereum, seven on BSC and four have zero vulnerabilities reported on both chains, (4) categories Write to Arbitrary Storage Location, and Hardcoded Gas exist only on BSC but in small numbers, (5) $1,612 on Ethereum and $2,113 on BSC are the total potential balances at stake due to vulnerabilities in smart contracts, (6) only 3% of smart contracts are unique, while 94% are duplicates on Ethereum and 98% on BSC, (7) only four contract bytecodes appear identically on both chains.

1.5 Project Outline

This project has eight chapters. In the first chapter, we presented our inspiration for undertaking this research, related work on the topic, and the contributions made through this thesis. In the second chapter, we state any relevant background information required to grasp the perspective of this work.

Continuing to the third chapter, we elaborate on the related work in smart contract security in recent years. The fourth chapter discusses the framework's methodology at a higher level without delving into too much implementation detail. The fifth chapter is dedicated to discussing the technical aspects of the Centaur framework. The sixth chapter evaluates the vulnerability state of Ethereum and Binance Smart Chain using Centaur while attempting to answer the research questions posed in the first chapter.

In chapter seven, we review the limitations faced during the research process while unfurling what plans we have for our tool. We also discuss threats to validity and the mitigation strategies we adopted. In the eighth and final chapter, we summarise and reflect on the research done, concluding with a discussion regarding legal, professional, and ethical considerations relevant to this project.

Availability: The technical artefact is available at github.com/mchara01/centaur.

Chapter 2

Background

This chapter briefly introduces several key concepts that are essential to understanding this innovative project. First, we define the jargon surrounding the blockchain field deemed important for understanding this in-depth study. Then, we discuss what smart contracts and the Ethereum Virtual Machine (EVM) are. Later in this chapter, we elaborate on the vulnerabilities in smart contracts and conclude with how to use analysis tools to discover potential weaknesses.

2.1 Blockchain

A blockchain maintains a decentralised ledger that unalterably and transparently records transactions into blocks. This distributed ledger is shared among the nodes of a P2P network, which are responsible for processing new transactions through a consensus protocol and appending new blocks to the end of the blockchain, thus maintaining a chronological structure.

Proof-of-Work (PoW) is a popular example of a consensus mechanism used for validating erroneous transactions [19]. Bitcoin and Ethereum utilise this consensus mechanism; however, critics claim that it is not eco-friendly and requires sizeable participation to maintain its security (at least 51% of the network's computing power is not malicious). As a result, many newer chains use other consensus protocols, such as Delegated Proof-of-Stake (DPoS) [20]. For example, BSC uses Delegated Proof-of-Staked-Authority (PoSA), a combination of Delegated Proof-of-Stake and Proof-of-Authority (PoA), where participants stake BNB to become validators [15]. Notably, Ethereum plans to change its consensus mechanism from PoW to PoS.

The typical structure of a block that constitutes a blockchain can be seen in Figure 2.1. The time needed to generate such a block on Ethereum is approximately 14 seconds and on the Binance Smart Chain it is only three seconds.

There are diverse types of nodes on a blockchain's network, such as full, archive, and light nodes. The specific type that we used in our study is an archive node. An archive node retains all history of a blockchain, meaning it has a snapshot of

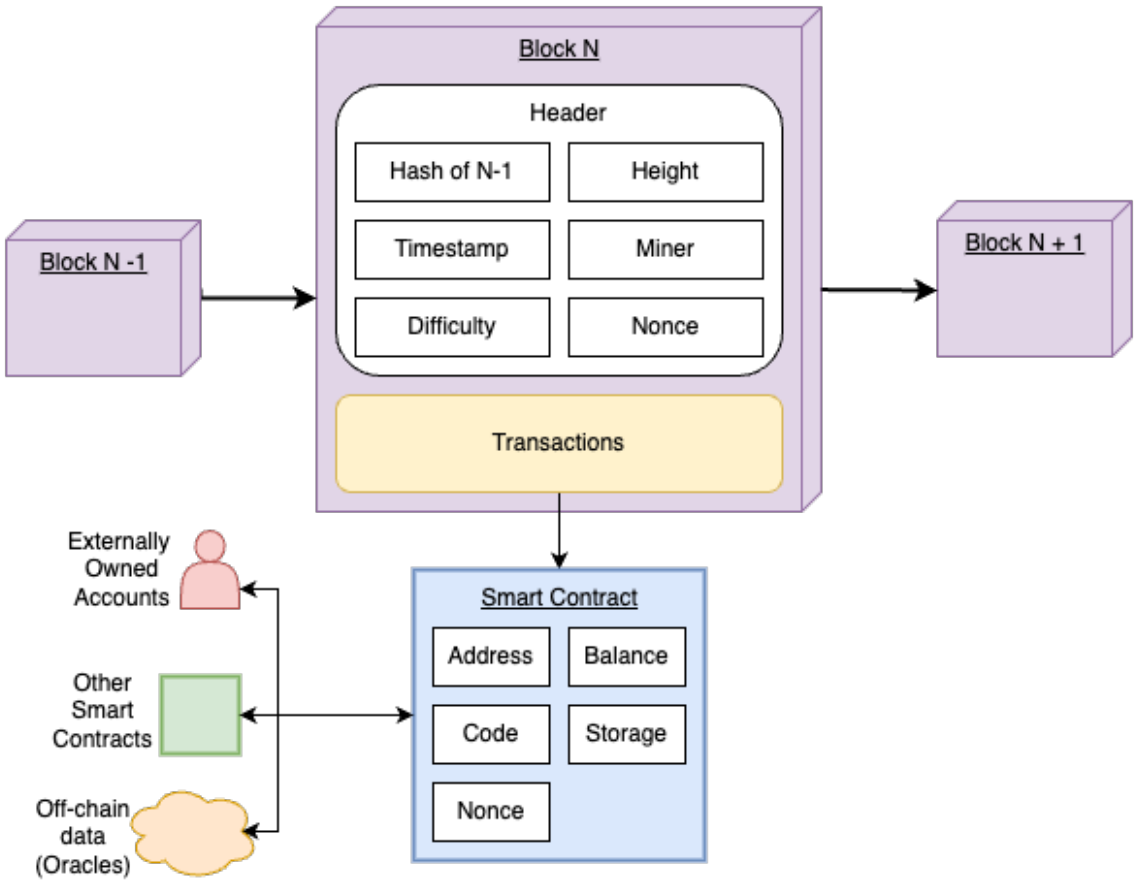


Figure 2.1: Structure of a block and a smart contract on blockchain.

the blockchain at each block since its genesis block [21]. This enabled us to examine the past states of the Ethereum and Binance Smart Chain's main blockchains and extract useful information about their past. These two popular blockchains also have Testnets besides their Mainnet, which are used to assess smart contracts in a production-like environment before deploying them on the Mainnet. These Testnets are incredibly useful since the blockchain is immutable, and mistakes on the Mainnet can prove costly [22].

You can use a client available in most major programming languages to interact with the network nodes. The most prominent client for Ethereum is go-ethereum (geth) written in Golang [23, 24] and for BSC it is a fork of geth, modified to work for BSC [25]. We used these two clients in this project to gain important information from their respective archive nodes.

2.2 Smart Contracts and EVM

The most distinguished blockchain with smart contract capabilities is Ethereum. There are two types of accounts on Ethereum, the Externally Owned Accounts (EOA)

that are addresses with an accompanied balance and specialised accounts (smart contracts) that contain balances, executable code, volatile and non-volatile storage and a nonce indicating the number of contract creations made by this account. Balances in Ethereum are measured in *Wei*, the smallest denomination of ether (1 Wei = 10^{-18} ether) and in BSC in *Jager*, the smallest denomination of BNB (1 Jager = 10^{-8} BNB).

Smart contracts are full-fledged program objects embedded in the blockchain application layer and execute when predetermined contractual terms are met, without needing the intervention of trusted third parties [26]. Figure 2.1 illustrates what constitutes a smart contract and how it interacts with other entities of the blockchain ecosystem. Smart contracts are initialised and executed via transactions. Real-life use cases of smart contracts include [27, 28];

- **Finance**: Trading, investing, lending and other traditional banking and financial services can be coded into smart contracts to provide speedy automated financial transactions. The Decentralised Finance (DeFi) ecosystem powers 24/7 and reduces the costs of financial transactions without needing a central authority (i.e., a bank).
- **Real Estate**: By converting the property into a unique token (tokenisation), much of the record management can be done by a smart contract and save the implicating parties unnecessary costs, such as title transfers and legal counsel.
- **Wallet**: Smart contracts manage funds, handle private keys of one or multiple owners, and send/receive transactions to simplify communications with the blockchain.

Another crucial application of smart contracts is they are used to develop protocols deployed on top of a blockchain, for example, Polkadot [29], and the Lightning Network [30]. The former provides interoperability by enabling transfers between blockchains (cross-blockchain), and the latter enables instant payments across participants and can scale to billions of transactions per second.

An additional, equally important use of smart contracts is creating tokens, such as Non-Fungible Tokens (NFT). Token standards form a guide for the creation, issuance, and deployment of new tokens [31]. Some essential token standards for Ethereum include ERC-20 and ERC-721, and for BSC are BEP-20 and BEP-721. ERC-20 and BEP-20 describe a usage rule blueprint for the fungible tokens on their respective chain, whereas ERC-721 and BEP-721 define the rules for the creation of NFTs.

Smart contract developers prefer to use high-level languages when coding. The most popular coding language for developing smart contracts is Solidity [1], an object-oriented, JavaScipt-like, contract-oriented language with a complete instruction set. Features of Solidity include static typing, inheritance, and complex user-defined types. There are other high-level languages available for writing a smart contract.

[1] https://docs.soliditylang.org/en/v0.8.16/

Harz *et al.* [32] presented a thorough literature survey on contract languages and their security features.

The high-level code is compiled into the targeted low-level VM bytecode and then executed by the Ethereum Virtual Machine (EVM). EVM is a quasi Turing-complete, stack-based, low-level intermediate representation (IR). EVM operations have a predefined gas cost to execute to ensure that smart contracts eventually terminate. This prevents the launch of DoS attacks on EVM-based chains [33].

2.3 Systematisation of Vulnerabilities

The goal of Centaur is to perform a vulnerability analysis on Ethereum and Binance Smart Chain, using automated security analysis tools to explore their respective Mainnet vulnerability state. However, before moving on to the specifics of Centaur, we need to understand what kind of bugs exist in smart contracts and how these appear. To do this, we systematise the security vulnerabilities using an extended version of the *DASP10* [17] classification and *SWC Registry* [18], as seen below. Table 2.1 further categorises these vulnerabilities into three groups, namely Solidity, EVM, and Blockchain, according to the level at which they appear. As the table illustrates, most vulnerabilities stem from Solidity code errors.

Level	**Vulnerability**
Solidity	1.Reentrancy 2.Access Control 3.Arithmetic Issues 4.Unchecked Return Values For Low Level Calls 5.Denial of Service 10. Honeypots 11.Non-Isolated External Calls (Wallet Griefing) 12.Greedy 13.Assert Violations 14.Delegate Call To Untrusted Contract 16.Hardcoded Gas 17.Jump to an Arbitrary Instruction
EVM	9.Short Address Attack 15.Write to Arbitrary Storage Location 18.Callstack Depth Attack Vulnerability
Blockchain	6.Bad Randomness 7.Front-Running 8.Time Manipulation

Table 2.1: Mapping between vulnerability categories and the level they are introduced.

1. Reentrancy (SWC-107): One of the most popular vulnerabilities in smart con-

tracts. A type of recursive call attack where the called external contract is allowed to make new calls back to the calling smart contract before the initial execution finishes, performing undesirable operations. A remarkable real-world Reentrancy attack is TheDAO attack[7, 8] where $60M were lost.
For example, a reentrancy vulnerability can be found in Line 4 in Figure 2.2, where function *call()* sends money to the address of *msg.sender*. If this address is a smart contract, its fallback function will be executed, calling back the *reentrancy()* function of the victim's contract. To protect against Reentrancy attacks, you should modify balances/code before calling an external contract.

```
1     function reentrancy(uint256 _amount) public {
2         uint bal = balances[msg.sender];
3         require(bal >= _amount);
4         require(msg.sender.call.value(_amount));
5         balances[msg.sender] -= _amount;
6     }
```

Figure 2.2: Reentrancy bug.

2. Access Control (SWC-105/106/115): An issue common in all types of programs, not just smart contracts. Improper or missing Access Control, such as visibility, can allow adversaries to withdraw money from a contract's balance or *self-destruct* the contract.
In Figure 2.3, we can view in Line 2 the *selfdestruct()* function that can be called by anyone to delete the bytecode stored to the smart contract address and send the remaining balance to the parameter-specified address. To prevent Access Control related vulnerabilities, a multiple signature scheme can be used for approving a *self-destruct* action and implementing controls so only authorised parties can trigger a withdrawal.

```
1           function access_control() {
2             selfdestruct(msg.sender);
3           }
```

Figure 2.3: Access Control bug.

3. Arithmetic Issues (SWC-101): This type of issue is triggered when an arithmetic operation exceeds the upper or lower bound of the variable's type involved, causing overflow or underflow (e.g., of an integer). This type of bug is common in other types of programs as well. A real-world example can be found at the *batchTransfer()* function Beauty Ecosystem Coin [34].
For example, Figure 2.4 illustrates a code snippet where an attacker can call the function *arithmetic_issue()* with an input big enough to overflow the *counter* variable and reset it back to zero. As a remedy, it is advisable to use safe mathematics libraries when using arithmetic operations in smart contracts.

```
1         uint public counter = 3;
2
3         function arithmetic_issue(uint256 amount) public {
4             counter += amount;
5         }
```

Figure 2.4: Arithmetic Issues bug.

4. Unchecked Return Values For Low Level Calls (SWC-104): As the name suggests, the return boolean value of low-level functions *call(), delegatecall(),* and *send()* is not checked, even if it is an exception. It could cause unanticipated flows in the subsequent program logic.
An example code snippet can be seen in Figure 2.5. To prevent this type of issue, simply check the possibility of the functions mentioned above failing and handle errors accordingly (e.g., by using a *require()* statement).

```
1         function unchecked_call(address callee) public {
2             callee.call();
3             // return of call() not checked
4             // unanticipated flow
5         }
```

Figure 2.5: Unchecked Return Values For Low Level Calls bug.

5. Denial of Service (SWC-113): The Denial of Service (DoS) category is broad, and consists of attacks that have the power to take permanently offline smart contracts. A shocking example of such an attack is the Parity Multi-sig wallet [9], which has resulted in a $280M damage.
A DoS vulnerability can be found in Line 2 of Figure 2.6, where the attacker may pass an enormous number as a parameter knowing that the for loop would require huge amounts of gas to finish, thus rendering the function useless when the block gas limit is reached. As a preventative measure, contracts should not loop over external user-defined structures.

```
1         function dos(uint256 _winnerNumber) {
2                 for(uint256 i = 0; i < _winnerNumber, i++) {
3                         // some heavy code
4                 }
5                 winner = _winnerNumber;
6         }
```

Figure 2.6: Denial of Service bug.

6. Bad Randomness (SWC-120): This class of vulnerabilities refers to the generation of seemingly random numbers, which renders it an insecure operation as

various fields can be used as a source of randomness and manipulated by the miner. Insecure fields considered insufficiently random include *blockhash, block.difficulty,* and *block.timestamp*. To overcome this security issue, we can use an Oracle as an external source of randomness.

7. Front-Running (SWC-114): Also known as the transaction ordering dependence (TOD). Transactions propagate through the network nodes for processing, which is made public for everyone to see. Miners include in the blocks that they mine the transactions that pay a high enough gas price. Since pending transactions are transparent, anyone could steal, for example, the solution to a problem, create a transaction with higher fees and bypass the original solution. This issue occurs when the smart contract code depends on the order of the transactions, creating a race condition vulnerability (e.g., only one winner). A commit reveal hash scheme is one solution to this vulnerability class.

8. Time Manipulation (SWC-116): Some smart contracts often depend on time values to trigger time-related events. However, malicious miners can manipulate the *block.timestamp* or *block.number*, as seen in Figure 2.7, which are often used for this purpose, to their advantage. Similar to Bad Randomness, one can use an Oracle to block the time entry window to prevent this type of bug. Smart contracts cannot receive and process external data, but oracles enable them to send and receive data from external servers.

```
1        function time_manipulation() public {
2                require(block.timestamp >= 1554422);
3                // do something if condition passes
4        }
```

Figure 2.7: Time Manipulation bug.

9. Short Address Attack (N/A SWC ID): Concerns third-party applications that interact with smart contracts. It is possible to send parameters that do not follow the ABI specification. If the addresses are shorter, the EVM will pad the remaining bytes with 0's. The issue arises when third parties do not validate input, allowing attackers to withdraw money that does not belong to them.

The tenth category of DASP10 represents any unknown vulnerability that does not fall into the first nine categories. During this project, we decided to extend this taxonomy and replace the tenth category with other vulnerabilities discovered by the analysis tools. Subsequently, we added eight additional vulnerability groups, reaching a total of 18 vulnerability classes. Below we can see the remaining eight.

10. Honeypots (N/A SWC ID): A new type of vulnerability class where attackers take a proactive approach and do not search for vulnerable contracts. Instead, they deploy seemingly vulnerable contracts to lure hackers into trying to exploit them,

but they end up losing their money as these contracts contain traps.

11. Non-Isolated External Calls a.k.a Wallet Griefing (SWC-126): This vulnerability group contain contracts that funnel data into sub-calls to other contracts. An attacker can leverage this by providing just enough gas to execute a transaction but not enough for the sub-call to succeed, thus censoring transactions.

12. Greedy (N/A SWC ID): This grouping refers to contracts that lock funds indefinitely, not allowing them to be released. Some times this is a result of developers errors, as instructions that release funds such as *send(), call(),* and *transfer* are either not reachable (among dead code) or completely missing from a contract. An attack that exploited this vulnerability resulted in locking $200M worth of Ether indefinitely [35].

13. Assert Violations (SWC-110): The Solidity *assert()* function should only be used for asserting invariants. Satisfactorily working code should not be able to reach a failing assert statement, as it could mean a vulnerability exists, or the *assert()* statements is used the wrong way, like in Line 2 in Figure 2.8.

```
1         function assert_violations() public {
2             // failing assert statement reached
3             assert(false);
4         }
```

Figure 2.8: Assert Violation bug.

14. Delegate Call To Untrusted Contract (SWC-112): A variant of the *call()*, but with *delegatecall()* the bytecode at the target is executed in the context of the calling contract instead. Calling into untrusted contracts is risky, as the target code has the power to alter storage values and control the caller's balance. As remediation, avoid calling untrusted/unknown contracts and always check user-defined target addresses to ensure they are safelisted. At Line 2 in Figure 2.9 we can see an example where the user input is used as-is in a *delegatecall()*.

```
1     function delegatecall(address callee, bytes _input) public {
2         require(callee.delegatecall(_input));
3     }
```

Figure 2.9: Delegate Call To Untrusted Contract bug.

15. Write to Arbitrary Storage Location (SWC-124): Data related to smart contracts are stored on the EVM. If an attacker manages to write to arbitrary storage locations of a contract, any checks in place can be bypassed and corrupt the storage. For instance, they can overwrite the contract owner field.

In Figure 2.10, we can view a write to arbitrary storage location using dynamic arrays in Line 8.

```
1          uint256[] map;
2
3          function writeArbitrary(uint256 key, uint256 value) public {
4              if (map.length <= key) {
5                  map.length = key + 1;
6              }
7
8              map[key] = value;
9          }
```

Figure 2.10: Write to Arbitrary Storage Location bug.

16. Hardcoded Gas (SWC-134): Functions such as *transfer()* and *send()* forward a stipend of 2300 gas. However, events such as Hard Forks can cause a significant variation in gas costs of EVM instructions. This can lead to breaking deployed contracts that assume a fixed gas cost. Do not specify a fixed amount of gas when performing calls to prevent this from happening.

17. Jump to an arbitrary instruction (SWC-127): When this type of vulnerability exists, attackers can arbitrarily change function type variables and consequently execute random code instructions, bypassing, for example, any validations. This problem is further amplified when inline assembly exists in a contract.

18. Callstack Depth Attack Vulnerability (N/A SWC ID): The EVM design limits the call-stack's depth to 1024 frames. Each time a *send()* or *call()* instruction are used during the communication of two smart contracts, the call-stack's depth increases by one. An attacker can take advantage of this and prepare a contract to call itself 1023 times before sending a transaction that will fail. A real-world example of this vulnerability could be found in the popular *King Of The Ether Dapp*, where the king would not receive any payment due to this issue [36]. A hard fork of the Ethereum blockchain redefined how gas consumption was calculated, which meant the maximum reachable depth of the call stack always falls under 1024.

2.4 Using Analysis Tools to Detect Bugs

There have been significant efforts from the blockchain community in recent years to help developers build secure smart contracts with the creation of automated analysis tools (cf. Table 2.1). These tools seek to detect bugs in smart contracts before deployment, as bugs in that stage, can prove costly because they are irreversible and irremediable.

The proliferation of automated analysis tools is justifiable as they became necessary for scanning smart contracts for security flaws. Currently, these tools focus on smart contracts intended for Ethereum due to its steady adoption and being the most prominent platform for deploying smart contracts. Nevertheless, to our disbelief, there is no published study that aims to uncover the state of smart contract vulnerabilities on other EVM-based blockchains beyond Ethereum.

These tools can be categorised into two groups based on their characteristics. Some tools, for example, work on EVM bytecode and others on its high-level representation (e.g., Solidity source code). Many tools work on both [37]. One other tool, FSolidM [38] generates solidity code for analysis by taking as input a formal specification. It is worthy of note at this point that there is a subtle distinction regarding the EVM bytecode. More precisely, there is the *init bytecode* that includes the contract constructor and initialisation code for deploying the code the first time. There is also the *runtime bytecode* which is the code the EVM evaluates every time a contract is called. The latter is what blockchain explorers show and what most tools analyse; however, some tools work on the former only, such as Manticore [39]. Throughout this study, when we refer to EVM bytecode, we mean the *runtime bytecode* [40].

Moreover, there are different methods with which analysis is conducted. One is *Static analysis*, where the source code or bytecode is analysed without executing it. Control flow analysis is an instance of static analysis. The opposite of *Static analysis* is *Dynamic analysis*, with which the smart contract is examined while it is executed. Symbolic execution and fuzzing are examples of dynamic analysis techniques. Since the first analysis tool Oyente, back in 2016, several tools have been developed in both categories (cf. Table 4.1). The majority of tools are in the *Static analysis* category [37].

The state of security issues in smart contracts is diverse. Issues could be classified according to the layer they appear. For instance, they could be categorised as Solidity, EVM, or Blockchain-related issues, as illustrated in Table 2.1 [41, 42]. Tools usually can detect only a subset of all known security bug types, as so many of them exist. There are some bugs that are not discovered by any tools (e.g., *Short Address Attack*), whereas common bugs, such as *Reentrancy*, are covered by many tools. Therefore, producing a completely secure smart contract is a thankless task. Developed contracts require a thorough audit before being released on the chain. It is argued that the most common reason for the vulnerabilities in smart contracts is the programme architecture of blockchain platforms that remain unfamiliar to developers.

Summary: A blockchain is a distributed storage system shared among P2P network nodes. Blockchains are best known for their prime role in cryptocurrency systems, such as Ethereum, for maintaining a secure and decentralised record of transactions. Smart contracts are specialised programs stored on a blockchain to execute transactions where all counterparties can trust the outcome. However, smart contracts have weaknesses and security issues that can undermine the platform's core security prin-

ciples. This chapter outlines how attackers can undermine a smart contract and destabilise the blockchain for their benefit. In this work, we present Centaur, which helps detect vulnerabilities by leveraging existing program analysis tools and making smart contracts more secure on multiple chains.

Chapter 3

Related Work

The nature of work relevant to this project appertains to EVM-based blockchain smart contracts, their vulnerabilities and automated analysis tools. In this section, we present relevant material (e.g., approaches, tools, surveys) of the work published in the field thus far. More specifically, we present recent research relating to vulnerabilities and their detection, datasets consisting of smart contracts, and other empirical studies similar to ours, exploring the complex ecosystem of blockchain.

3.1 Vulnerabilities in Smart Contracts

Since smart contracts handle assets of substantial value, their development must be steal-and-tamper-proof. A step toward detecting vulnerabilities via automated analysis tools and defending against them was made by Atzei *et al*. [42], with a survey of analysis tools which presented 12 common types of vulnerabilities, nine attacks, and three defences. In a more comprehensive survey by Chen *et al*. [43], 40 vulnerabilities, 29 attacks, and 51 defences locations and root causes were examined. In the same survey, there is a discussion on best practices to avoid such issues. *Consensys* also provides a documentation on smart contract security best practices [44], which help in avoiding common pitfalls.

There have also been efforts to create taxonomies while grouping vulnerabilities in classes depending on the level where they are introduced. Atzei *et al*. [42] and Angelo *et al*. [41] categorised vulnerabilities into three groups; Solidity (*Call To The Unknown, Gasless Send, Exception Disorders, Type Casts, Reentrancy, Keeping Secrets*), EVM (*Immutable Bugs, Ether Lost In Transfer, Stack Size Limit*) and Blockchain (*Unpredictable State, Generating Randomness, Time Constraints*). Chen *et al*. [43] classified vulnerabilities into four Ethereum layers; *Application, Data, Consensus, Network*. Another taxonomy used by Durieux *et al*. in [12] is *DASP10* [17], allowing easy evaluation of automated analysis tools. In our work, we also used the DASP10 taxonomy to group the vulnerabilities reported by the analysis tools; however, as mentioned in Section 2.3, we have extended it to accommodate more vulnerabilities. To enrich the vulnerability categorisation, we have also used the *Smart Contract Weakness Classification* (SWC) Registry [18] to map found vulnerabilities to an *SWC ID*, which

can help users of the framework to learn more about a specific vulnerability (e.g. description, remediation, code examples). In addition, *Sigma Prime* [45] comprised a comprehensive source of information for us regarding smart contract vulnerabilities, their patterns, prevention techniques and real-world examples.

3.2 Analysing Smart Contracts

As we present in Section 4.1, various freely available automated analysis tools exist to test a smart contract for bugs. There is also the SmartBugs execution framework [12], which allows exploiting parallelism and running multiple analysis tools simultaneously on a given dataset of smart contracts. We modified this framework to conduct our measurements on bytecode instead of Solidity source code using our selection tools (cf. Table 4.2). Solhydra [46] was another execution framework containing six static analysis tools that produced an HTML report with their results; however, its open-source repository does not exist anymore. During our research, we discovered there are very few execution frameworks which can be used for large-scale comparison between existing and new tools. Furthermore, despite the ubiquity of analysis tools, bugs continue to proliferate in smart contracts [47].

Recent academic work has focused on a different path from the above in order to secure smart contracts. More specifically, on producing test cases (exploits) for a smart contract under deployment to ensure their quality. Wu *et al.* [48] proposed a framework for mutation testing of smart contracts, which outperforms the alternative, coverage-based method for detecting bugs. In the domain of fuzz testing (fuzzing), the tool ContractFuzzer by Jiang *et al.* [49] generates exploits for input based on the Application Binary Interface (ABI) specification of a smart contract for detecting two vulnerabilities. On the contrary, Echidna [50] employs grammar-based fuzzing for testing smart contracts.

Our work focuses on the first technique (multiple analysis tools) from above for testing smart contracts, where analysis tools are utilised to scan smart contracts for vulnerabilities.

3.3 Smart Contract Datasets

Standardised benchmarks are useful for evaluating automated analysis tools for smart contract vulnerability detection. Benchmarks consist of many smart contracts for practitioners and researchers to utilise during their experimentation phase.

For analysing smart contract Solidity source code, there are a few datasets of vulnerable contracts in the wild. Popular examples include *Ethernaut* [51], an online educational game with vulnerable smart contracts that need to be hacked, and *(Not*

So) Smart Contracts [52], an open-source collection of vulnerable smart contracts. Other examples include *VeriSmartBench* [53], *EVM Analyzer Benchmark Suite* [54] and *Smart Contract Benchmark Suite* [55]. However, all these repositories consist of not many (tens) smart contracts, making it difficult to produce large-scale comparisons and evaluations.

Furthermore, Durieux *et al*. [12] have open-sourced their manually curated collection of 143 (at the time of writing) vulnerable smart contracts with 208 tagged vulnerabilities [56]. In addition, the same team published a dataset of 47,587 unique Solidity smart contract source codes [57] involved in at least one transaction, taken from the Ethereum network to measure the state and performance of existing automated analysis tools. Their SmartBugs framework influences our work, but on the contrary, we studied, apart from Ethereum, another EVM-based chain, the Binance Smart Chain, to see how the two blockchain states compare in terms of vulnerabilities without emphasising the automated tools' quality. Moreover, for creating a collection of real-life verified smart contracts, one can use a blockchain explorer, such as Etherscan [58], which provides a free API for developers.

For bytecode analysis, smart contract bytecodes are often gathered from the Mainnet of a blockchain. We collected our material for Centaur's experiments this way using archive nodes from the main chains of Ethereum and BSC. The issue with source code and bytecode techniques is that it is difficult to know a priori from the vast number of contracts which are the vulnerable ones. And establishing ground truths about those that are is inevitably tricky. Hence, a selected vulnerable smart contract's benchmark is important for evaluating analysis tools. *SolidiFI* by Ghaleb *et al*. [47, 59] is a framework that can be deployed to inject various types of bugs into several locations of a smart contract's invulnerable code to make it vulnerable. It enables them to perform large-scale experimentation consisting of 9,369 distinct bugs to evaluate six analysis tools. Using any tool as an oracle purely based on factors such as popularity is at the very least problematic because research has shown that all analysis tools have a large number of false positives [12].

3.4 Empirical Studies

Durieux *et al*. [12] performed a large-scale study on 47,587 smart contracts deployed on the Ethereum network to compare the performance of nine automated analysis tools and gauge their effectiveness. They have also crafted a dataset of 69 manually annotated smart contracts with 112 bugs collected from online repositories. However, a large proportion of the 69 contracts were not agnostic to the tools used, indicating that the result could be biased [47].

In another approach, Ghaleb *et al*. [47] created *SolidiFI* with which 9,369 distinct bugs were injected into 50 smart contracts and evaluated six commonly used static analysis tools on them. The focus was on creating a comprehensive coverage evalua-

tion; thus, non-vulnerable code was transformed into vulnerable by the bug injection process so that vulnerabilities would be present in all valid locations. Both studies mentioned above have shown the existence of bugs not initially detected by any tool, while the number of false positives reported was high.

Parizi *et al*. [60] presented an empirical evaluation of four smart contract analysis tools; namely *Mythril, Oyente, Securify,* and *Smartcheck* on ten real-world smart contracts. *Smartcheck* was found to be the most effective, whereas *Mythril* was the most accurate. Work done in [12] corroborates this. Pinna *et al*. [61] carried out a comprehensive empirical study containing 10,174 smart contract source codes from the Ethereum blockchain and provided empirical results regarding smart contract features, like transactions, the role of the development community, and the source code characteristics.

Harz *et al.* [32] studied 10 smart contract verification tools, presenting various aspects of their security features. The survey by Angelo *et al.* [41] forms a great starting point for research regarding existing, state-of-the-art automated analysis tools, their underlying methodology, the type of vulnerabilities they can detect and availability. They presented data for 27 analysis tools. Hu *et al.* [62] examined 39 analysis tools regarding their methodology and input type. Finally, Kushwaha *et al.* [63] presented a systematic survey of 86 analysis tools, the highest number among all research papers and articles, and considered their analysis techniques and tool type.

To the best of our knowledge, the work conducted in this paper is the first of its kind since empirical studies focus on vulnerabilities present in Ethereum and not other EVM-based chains. We have crawled bytecodes from deployed smart contracts on Binance Smart Chain to provide a thorough comparison of vulnerabilities between these two blockchains.

Summary: Like any software, there is a tendency for smart contracts to contain bugs. Increased attacks on smart contracts cause financial pain and erosion of trust. Tools are necessary to detect vulnerabilities before smart contracts are activated [47]. Our research shows there is very little evidence available for comparing smart contract analysis tools on different platforms to measure their vulnerability state. Therefore, this study has taken the next step in using a specific toolset to compare vulnerabilities on Ethereum and BSC chains. In addition, there is a body of work on smart contract bug analysis for Ethereum and the different types of intrusion that can occur, but the state of other EVM-compatible chains is unexplored.

Chapter 4

Methodology

In this chapter and the next, we present the work that has been accomplished throughout the duration of this project. More accurately, we delve deeper into the details of the automated security analysis tools for smart contracts used in our study (Section 4.1). Then, we discuss the execution framework for these tools on the dataset we defined (Sections 4.2 and 4.3). Finally, we present the data collection, storage, and processing (Section 4.4).

4.1 Finding Automated Analysis Tools

We gathered the analysis tools to evaluate the smart contracts as an initial step in the process. In Table 4.1, we can see the 45 automated smart contract analysis tools we first gathered for the purpose of this study. To collect these tools, we initially searched major conferences in the fields of Software Engineering, Computer Security, and Programming Languages (i.e., ACM CCS, NDSS, USENIX, ICSE, ACM ASE, PLDI, IEEE S&P) to find publications referring to state-of-the-art analysis tools. To do this, we used the advanced keyword search option [1] of Google Scholar, and then followed their references for finding additional work. We manually omitted papers irrelevant to the EVM bytecode or Solidity source code analysis tools or did not conduct their work on Ethereum or Binance Smart Chain. To further enrich our list, we crawled GitHub for any tools using similar related keywords and used Google to search for any other tools in the wild.

Even though in Table 4.1 analysis tools appear abundant, only a subset of these was included in our study as we wanted these tools to fulfil criteria that will enable us to discover vulnerabilities in smart contracts without needing to modify their codebase. The tool inclusion criteria were the following;

1. Analysis tools are open source. We avoided using proprietary or closed-source

[1] with at least one of the following keywords: [“Smart contract”, “Smart contract security”, “Ethereum”, “ETH”, “Binance Smart Chain”, “BSC”, “Ethereum Virtual Machine”, “EVM”, “EVM bytecode”, “Solidity”, “Ethereum automated analysis tools”, “Blockchain”, “Blockchain security”, “Ethereum security”, “Ethereum vulnerabilities”]

tools, not **publicly available**, so we could inspect their code, gain insight and ensure the tools used were actively maintained and used by the community.

2. A tool must be able to be used for **vulnerability detection**. However, for many of these tools, we discovered that even though they are categorised as vulnerability-analysis tools, they only produce items, such as Control Flow Graphs, and do not warn the user explicitly on whether a vulnerability has been discovered. For this reason, we decided to exclude fuzzing tools as some require input besides the source code/bytecode and report only when the target crashes, or assertions are triggered.

3. We were only interested in tools that worked on the **EVM bytecode** level and not on Solidity source code. We decided this as we have access to an archive node on both Ethereum and BSC chains, which enabled us to collect any smart contract bytecode throughout their history. Many research papers that do not have this capability had to crawl various blockchain explorers programmatically via their API to get this metadata.

4. Analysis tools that require human interaction via a GUI while executing were unsuitable for this study. Therefore, we focused only on tools that work with a **CLI** in an automated fashion so they can be used in large-scale analysis.

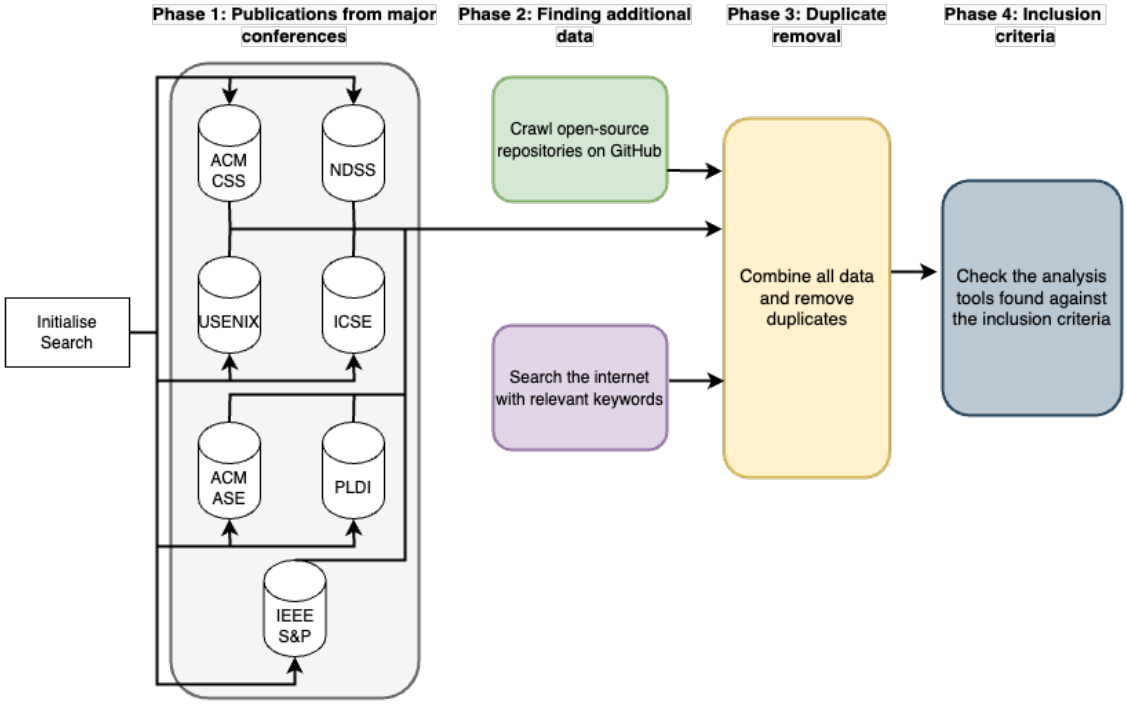


Figure 4.1: Methodology pipeline for including a tool in our study.

Figure 4.1 illustrates the above-described pipeline steps used for collecting the final set of analysis tools. The outcome of the above process is depicted in Table 4.1.

Starting from the left, the first and second columns are the tool number and name. The third column points to the research paper in the bibliography section, if available or code repository. The fourth column shows whether the analysis tool is open source. And the fifth displays whether the tool can be used for vulnerability detection. The next two columns display the level at which the analysis tool operates on, bytecode and/or source code, respectively. The final column depicts whether a tool can be used for bulk analysis. To find out whether these tools fulfilled these fields, we read their respective paper or inspected their open-source repository to answer the questions mentioned above.

The fourth, fifth, sixth and eighth columns of Table 4.1 illustrate which of the four constraints are fulfilled for every tool. We proceed only with the tools that passed these constraints. The nine successful candidate tools can be found in Table 4.2. For each of the nine analysis tools we included a reference to their publication (paper or repository), year of implementation, type of analysis performed, venue of publication if one exists and their publicly available code repository.

4.1.1 Tool Description

Although the core objective of this study was not to see how these analysis tools perform, we believe it is important to understand the underlying technology that enables them to detect vulnerabilities. So, below we briefly introduce the tools used in this study. The information is gathered from each tool's publication, and the work of Angelo *et al.* [41] and Kushwaha *et al.* [37].

Conkas: Static analysis tool utilises symbolic execution and an Intermediate Representation (IR) to find vulnerabilities. It can detect up to five vulnerability types from the DASP10 taxonomy and accepts input EVM bytecode and Solidity source code. Written in Python and implemented in 2019.

HoneyBadger: Contrary to other analysis tools, HoneyBadger is a static analysis tool that focuses on detecting honeypot contracts that are seemingly vulnerable but contain hidden traps. It performs systematic analysis combined with heuristics to expose honeypots. Its systematic analysis consists of symbolic, cash flow, and honeypot analysis. It is based on Oyente, developed in Python in 2019.

MadMax: Static analysis tool invented in 2018 uses the Gigahorse IR lifter. MadMax focuses on detecting gas-focused vulnerabilities. Employs a combination of control-flow-analysis-based decompiler and declarative program structure queries.

Maian: A dynamic analysis tool that performs symbolic execution and concrete validation on both source code and EVM bytecode using SMT solver Z3 to solve path constraints. Detects Lost Ether, Unchecked Calls, Usage of self-destruct bugs, and omits blockchain-related bugs completely. It was implemented in Python in 2018.

Smart Contract Automated Analysis Tools							
No.	**Tool Name**	**Ref.**	**Availability**	**Vuln. Det.**	**Bytecode**	**Source**	**Bulk Analysis**
1	Concas	[64]	✓	✓	✓	✓	✓
2	contractFuzzer	[49]	✓	✓		✓	
3	contractLarva	[65]	✓			✓	
4	E-EVM	[66]	✓		✓		
5	Echidna	[67]	✓	✓		✓	
6	Erays	[68]	✓		✓		
7	Ethainter	[69]		✓	✓	✓	✓
8	EthBMC	[70]	✓	✓	✓		✓
9	Ether	[71]		✓		✓	✓
10	EtherSolve	[72]	✓		✓		
11	Ethersplay	[73]	✓		✓	✓	
12	EtherTrust	[74]	✓		✓		✓
13	EthIR	[75]	✓		✓		
14	eThor	[76]					
15	FSolidM	[38]	✓				
16	Gasper	[77]		✓	✓		✓
17	Gigahorse	[78]	✓				
18	HoneyBadger	[79]	✓	✓	✓	✓	✓
19	KEVM	[80]	✓		✓		
20	pakala	[81]	✓	✓	✓		✓
21	MadMax	[82]	✓	✓	✓	✓	✓
22	Maian	[35]	✓	✓	✓	✓	✓
23	Manticore	[39]	✓	✓	✓*	✓	✓
24	Mythril	[83]	✓	✓	✓	✓	✓
25	Octopus	[84]	✓		✓		
26	Osiris	[85]	✓	✓	✓	✓	✓
27	Oyente	[10]	✓	✓	✓	✓	✓
28	Porosity	[86]	✓+	✓	✓	✓	
29	rattle	[87]	✓		✓		
30	ReGuard	[88]		✓	✓	✓	✓
31	Remix	[89]	✓	✓		✓	
32	SASC	[90]		✓		✓	✓
33	sCompile	[91]		✓		✓	✓
34	Securify	[92]	✓+	✓	✓	✓	✓
35	Securify (v2.0)	[92]	✓	✓	✓	✓	✓
36	Slither	[93]	✓	✓		✓	✓
37	SmartCheck	[94]	✓+	✓		✓	✓
38	Solgraph	[95]	✓	✓		✓	
39	Solhint	[96]	✓			✓	
40	SolMet	[97]	✓			✓	
41	teEther	[98]	✓	✓	✓	✓	✓
42	Vandal	[99]	✓	✓	✓	✓	✓
43	Vertigo	[100]	✓				
44	VeriSol	[101]	✓			✓	
45	Zeus	[102]		✓			✓

Table 4.1: Comprehensive list of smart contract security analysis tools. ✓ - feature is applicable for the tool, ✓* - runs on deployment bytecode and not runtime bytecode, ✓+ - not maintained anymore (deprecated)

No.	Tool Name	Ref.	Year	Type	Venue	GitHub Repository
1	Concas	[64]	2019	SE & IR	N/A	/nveloso/conkas
18	HoneyBadger	[79]	2019	SE & H	USENIX	/christoftorres/ HoneyBadger
21	MadMax	[82]	2018	CFAD & DPSQ	OOPSLA	/nevillegrech/MadMax
22	Maian	[35]	2018	SE & CV	ACSAC	/MAIAN-tool/MAIAN
24	Mythril	[83]	2017	SE & SMT & TA	N/A	/ConsenSys/ mythril-classic
26	Osiris	[85]	2018	SE & TA	ACSAC	/christoftorres/Osiris
27	Oyente	[10]	2016	SE	CCS	/melonproject/oyente
34	Securify2	[92]	2018	SE & CVP	CCS	/eth-sri/securify2
42	Vandal	[99]	2018	SLR	CoRR	/usyd-blockchain/ vandal

Table 4.2: Smart contract analysis tools that pass the inclusion criteria and did not timeout consistently or failed during testing. SE - Symbolic Execution, H - Heuristics, IR - Intermediate Representation, CFAD - Control-Flow-Analysis-Based Decompiler, DPSQ - Declarative Program-Structure Queries, CV - Concrete Validation, SMT - Satisfiability Modulo Theories solving, TA - Taint Analysis, CVP - Compliance and Violation Pattern checking, SLR - Semantic Logic Relations

Maian, as opposed to most other tools, tries the exploits it produces to reduce false positives.

Mythril: Created in 2017 and implemented in Python by *ConsenSys*, a popular blockchain software technology company. Accepts Solidity source code and EVM bytecode and uses three approaches for scanning smart contracts for vulnerabilities: namely symbolic execution, Z3 to prune the search space, and taint analysis. Mythril is considered one of the most complete and accurate open source tools and detects several types of bugs.

Osiris: Employs symbolic execution and taint analysis on EVM bytecode and Solidity source code. It mainly serves for detecting variations of unchecked integer bugs. It was developed in Python in 2018. Osiris is based on Oyente.

Oyente: One of the oldest smart contract analysis tools in the field, implemented in Python in 2016. Performs symbolic execution on both source code and EVM bytecode to statically analyse all feasible paths of a smart contract. Detects the following bugs: Timestamp, Transaction Order Dependency, Callstack Depth, and Reentrancy. Its main analysis components include a CFG builder and a Z3 solver for excluding unreachable paths.

Securify v2.0: A fully automated security analyser that exercises symbolic execution on the smart contract's dependency graph to extract specific code semantic data and checks against violation patterns to detect whether a security property holds or not. It can detect Transaction Order Dependency, Lost Ether, Reentrancy and other bugs. Works on both source code and EVM bytecode. Written in Java in 2018.

Vandal: A static analysis tool implemented in Python 2018. It transforms EVM bytecode into semantic logic relations. More specifically, it decompiles EVM bytecode into an IR, then translates the IR into Horn clauses and passes these into a reasoner for vulnerability detection. It Detects Reentrancy and other Solidity-level coding bugs. Written in Python.

As observed in Table 4.1, more tools could be included in the final set as they pass the inclusion requirements. For example, EthBMC, teEther, pakala, and Manticore pass all the inclusion criteria. However, we eventually had to exclude them as the first three tools required more time to run than the 30-minute limit we set during testing. Manticore works only on deployment bytecode and not runtime bytecode (cf. Section 2.4 for a distinction between the two). Also, there are two version of Securify that pass the inclusion criteria, the second being an update of the first one. We only included v2.0 as the first version is deprecated.

Table 4.3 depicts the vulnerability categories detected by the analysis tools, according to the extended DASP10 taxonomy we created and the SWC registry.

4.2 Datasets Design

One of the primary objective of this study is the creation of the dataset of smart contract bytecodes for the analysis tools to execute on with the aim of finding vulnerabilities, if any exist. Therefore, it was part of our mandate for both the Ethereum and Binance Smart Chain blockchains to create a dataset that will contain the smart contract features seen below. In addition, we include their name for each of these features, why they are needed, and ways to collect them.

- **Address**: The address is the most significant metadata of a contract, as it uniquely identifies it. Using the address of the smart contract, we can extract other information about it, for example, the source code and history of transactions. In other research [12], all types of data was collected from blockchain explorers such as EtherScan for Ethereum, BscScan for BSC, and Snowtrace for the Avalanche network. Our team has access to archive nodes on the Ethereum and BSC blockchain, which enabled us to get the smart contract addresses given that a valid block number is provided.

- **EVM Bytecode**: Is necessary for all analysis tools used in this study to operate on the EVM bytecode level. Thus, our dataset must include this information. On the contrary, other work [12] included the Solidity source code in their

Tool Name	DASP10+ Vulnerabilities Detected	SWC ID
Concas	1.Reentrancy	107
	3.Arithmetic Issues	101
	4.Unchecked Return Values For Low Level Calls	104
	7.Front-Running	114
	8.Time manipulation	116
HoneyBadger	10. Honeypot	N.A
MadMax	3.Arithmetic Issues	101
	5.Denial of Service	113
	11.Non-Isolated External Calls (Wallet Griefing)	126
Maian	2.Access Control	105, 106
	12.Greedy	N.A
Mythril	1.Reentrancy	107
	2.Access Control	105, 106, 115
	3.Arithmetic Issues	101
	4.Unchecked Return Values For Low Level Calls	104
	5.Denial of Service	113
	6.Bad Randomness	120
	8.Time manipulation	116
	13.Assert Violations	110
	14.Delegate Call To Untrusted Contract	112
	15.Write to Arbitrary Storage Location	124
	17.Jump to an arbitrary instruction	127
Osiris	1.Reentrancy	107
	3.Arithmetic Issues	101
	7.Front-Running	1114
	8.Time manipulation	116
Oyente	1.Reentrancy	107
	7.Front-Running	114
	8.Time manipulation	116
	18.Callstack Depth Attack Vulnerability	N.A
Securify v2.0	1.Reentrancy	107
	2.Access Control	105
	5.Denial of Service	113
	7.Front-Running	114
	16.Hardcoded Gas	134
Vandal	1.Reentrancy	107
	2.Access Control	105, 106, 115
	4.Unchecked Return Values For Low Level Calls	104

Table 4.3: Mapping between analysis tools and the vulnerability classes detected, based on the extended DASP10 taxonomy and SWC registry.

dataset. This entails that different analysis tools where used to analyse source code instead of bytecode. Again, the blockchain explorers contains this piece of data but we access it via the archive nodes as it much faster as no requests limits are imposed.

- **Number of transactions, token transfers**: These feature are needed to check if a smart contract has at least one transaction or one token transfer before adding it to our dataset. This is to eliminate redundant computation on contracts that have not been utilised in practice yet. For the collection of these two features, we followed the approach of other studies and developed scripts to crawl the two blockchain explorers for Ethereum and BSC via their free API and collect the two features using the addresses we acquired from the archive nodes.

- **Balance**: In addition to the above, we collect also the balance of a smart contract. In case there are no transactions or token transfers recorded for the respective smart contract, we check if at least it has a balance bigger than zero to add the smart contract bytecode to corpus of contracts to be analysed.

The above features are deemed essential and where the minimum for completing this study. However, there is more data available on the blockchain explorers that we opted not to collect as it will serve no purpose in the specs of this empirical study and it will required a considerable amount of time to collect. However, as a future goal, we aim to collect this secondary data to enrich our dataset and provide a more comprehensive contribution to the community. The non-essential features are the following: Source Code, Last transaction date, Compiler version, Balance of contract, Name of contract, Contract creation date, Lines of code. All these features can be gathered from the blockchain explorers, as we discussed above.

4.3 Framework Execution

After deciding on the analysis tools to be included in this study, we worked with them. Initially, we installed these tools locally. To do this, we used their official Docker container image when available or cloned their open-source GitHub repository otherwise. Afterwards, we tested them on a small set of 30 manually selected smart contract bytecodes, of which half were definitely vulnerable and the other half random. The testing dataset was crafted from deployed smart contracts found on Etherscan and BscScan blockchain explorers.

As a next step, we thought of developing an execution framework that would allow us to run multiple tools in parallel on a dataset of smart contracts that would be collected at a later stage. During development, we came across the *SmartBugs* framework [12] that did exactly this. However, *SmartBugs* was developed to work on Solidity source code. Thus, we started exploring its codebase and modifying it to work on bytecode instead. During this stage, we got in touch with one of the main

contributors of the tool, who provided us guidance on how to modify their framework. This phase also consisted of adding configurations for the analysis tools we wanted to include in our study.

Once this was accomplished, we proceeded to test the modified SmartBugs framework on the sample dataset. Our intention for developing this small dataset was to enable us to understand better how SmartBugs works. It also forms a much-smaller scale indicative sample of the final dataset, which contains a plethora of smart contract bytecodes for the framework to work on.

4.4 Step-by-Step Analysis Procedure

This section is purposefully broken down into subsections which constitute the entire process Centaur follows every time it is executed to analyse the EVM bytecode of smart contracts. Our data collection, storage, processing and analysis approach is summarised in Figure 4.2.

4.4.1 Data Collection

An initial starting point of the process was to find smart contracts to collect their bytecodes. Access to archive nodes on Ethereum and Binance Smart Chain proved beneficial for this endeavour. More specifically, we developed scripts that connect to these archive nodes, crawl over the transactions of the blocks provided, and extract the contract addresses and their respective bytecodes. This data is stored in a local RDBMS for easy access, as demonstrated in the next subsection.

The block numbers passed through the above crawling script are generated via another script using a random sampling technique on all the blocks of Ethereum and BSC blockchains. Those block numbers generated are stored in a file for the crawler to read from. The user determines the sample size.

Besides the smart contract addresses and bytecodes, we needed to capture additional metadata from these contracts that are required for filtering. In practice, it meant we wrote programs that crawl Etherscan [2] or BscScan [3] blockchain explorers to gather the metadata for given smart contract addresses and API key. The APIs are freely accessible for a specified number of requests. To circumvent any request obstacles, we implemented a request rate limit checker.

Moreover, for every smart contract address we have, we gather the following metadata: “number of transactions”, “number of ERC20 token transfers”, “number of

[2]https://etherscan.io/
[3]https://www.bscscan.com/

ERC721 token transfers", "number of BEP20 token transfers", "number of BEP721 token transfers", "balance", and "block number".

4.4.2 Data Storage

At this point, all the required data for this study was collected, and we needed a place to store it. For this, we opted to continue with a popular open-source, ACID-compliant SQL database, MariaDB[4]. We used its free official Docker image from the Docker hub, which allowed us to conveniently deploy it and have a database to store our data running promptly.

However, during another project stage, we decided to migrate the data into an sqlite3 database as well; it was more convenient in terms of portability. The data of sqlite3 resides in a single file which someone can easily inspect and reproduce/replicate our work. Our open-source repository currently contains the two databases and instructions for working with both.

The outcome of this approach is a corpus of deployed Ethereum and BSC smart contract addresses and relevant metadata, such as block number, balance, number of transactions, and number of token transfers. In addition, we can use this database to perform aggregate queries, such as finding the number of vulnerable smart contracts deployed on BSC.

4.4.3 Dataset Processing

With all the smart contract data stored in a database, it was necessary to extract the EVM bytecodes we gathered in the previous step and write them on the local filesystem in the form of a dataset for the SmartBugs framework to execute on. However, as the results in Section 6 shows, there are thousands of smart contracts, and a significant proportion of them are duplicates. This means that running SmartBugs on all of them will take a huge amount of time, and much of this time will be wasted as it will be spent on the duplicate bytecodes. Hence, we applied the following filtering conditions to restrict SmartBugs to only smart contracts that have a balance or have been used before.

- balance > 0
- number of transactions > 0
- number of token transfers > 0

Suppose at least one of these conditions stands and the bytecode is original (not a duplicate). In that case, the respective smart contract bytecode is extracted from the database and written in a file to become part of the dataset.

[4]https://mariadb.com/

4.4.4 Results Analysis

After finishing successfully with the above steps, we had everything we needed ready to run the SmartBugs framework and execute the analysis tools on the EVM bytecodes written into files on the local file system. SmartBugs executed the nine analysis tools outlined in Table 4.2 on every contract from the resulting corpus of smart contracts. Consequently, this particular step took days to complete.

Once SmartBugs had finished, a result file in a JSON format was created for every smart contract analysed, one for each of the nine analysis tools. The JSON format choice was made so we could easily parse the results during our analysis. In addition, we created a result parser for every analysis tool to interpret its result file and map any vulnerabilities to the extended DASP10+ and SWC taxonomies. Finally, the findings of the experimentation are stored in the database.

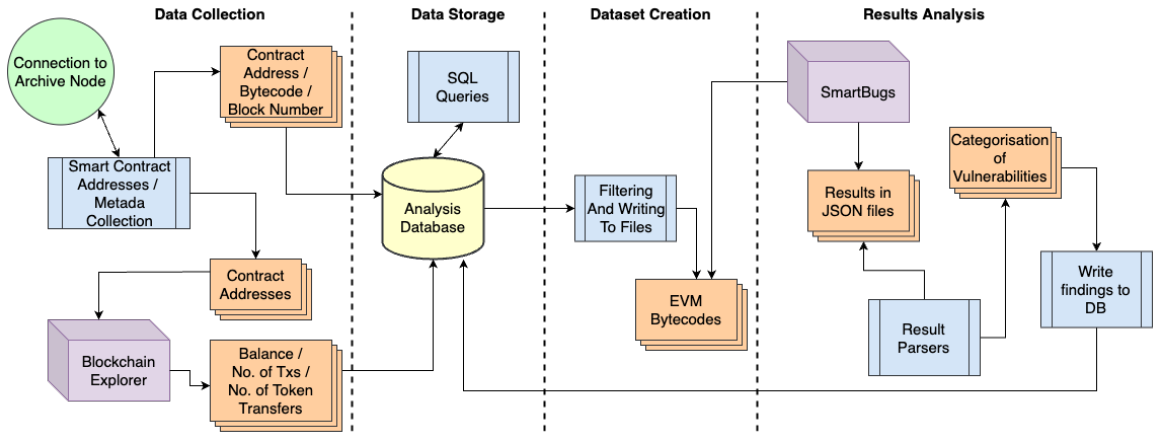


Figure 4.2: Overview of the analysis pipeline for smart contract EVM bytecodes.

Summary: This chapter outlines the procedure for selecting the tools that test smart contracts for possible bugs and the thought process behind the selection. Then, we explain why the nine analysis tools were selected and their designated function. In addition, the features of each tool and why some were unsuitable for this project are given. To ensure thoroughness, we also connected with an author from the SmartBugs team to help analyse our bytecode dataset. Finally, we discuss the step-by-step analysis procedure Centaur follows to analyse smart contracts.

Chapter 5

Implementation

This chapter on implementation details the blueprint that powers the Centaur framework. We commence with the smart contract analysis database and move on to the random sampling of block numbers. After that, we present the details of the block crawler and blockchain explorer crawler we developed. Finally, we discuss the last step of the process, the analysis stage of Centaur. This chapter, as opposed to the rest of the report, includes substantial figures and abstract pseudocode snippets of key parts of the implementations to understand the codebase better.

5.1 Database Deployment

For storing the smart contract data discussed in Sections 4.2 and 4.4 we chose to deploy a Docker container running MariaDB. The database is one of the key components of our contribution as it contains the results of block and metadata crawling and the analysis tools execution statistics and findings. It allows anyone to reproduce our work or use the data for conducting other similar studies on the two chains. MariaDB is deployed via the docker-compose YAML file seen in the figure in Appendix A.

Furthermore, we created a Bash script that can perform a database backup by exporting an SQL file that contains all *CREATE* and *INSERT* statements of our database and tables. In addition, we developed a restoration script that conveniently imports the above backup to any other MariaDB database on other machines. Moreover, we employed a mapping between important queries and their descriptions for users to conveniently use on our data.

In Figure 5.1, the Entity Relation (ER) diagram of the database is presented. Our database, named *analysis.db*, contains four tables: Address, Bytecode, Result, and Finding. These tables are connected via foreign key constraints. The Address table holds all the crawled data discussed in Section 4.2, along with a hashed version of the bytecode (SHA-256), the chain on which the contract is deployed (e.g., “eth” or “bsc”), and the execution timestamp. We preferred only to store the hashed versions as bytecodes tend to be long, and a hash allows us to check faster for duplicates

(checking is done by comparing strings). Therefore, we store the smart contracts' bytecodes separately in the Bytecode table, and we can retrieve them from there when we need them. The Result table stores the statistics for all executions, and Finding holds any vulnerabilities reported by the tools. The diagram is generated by using the open-source module *eralchemy* [1].

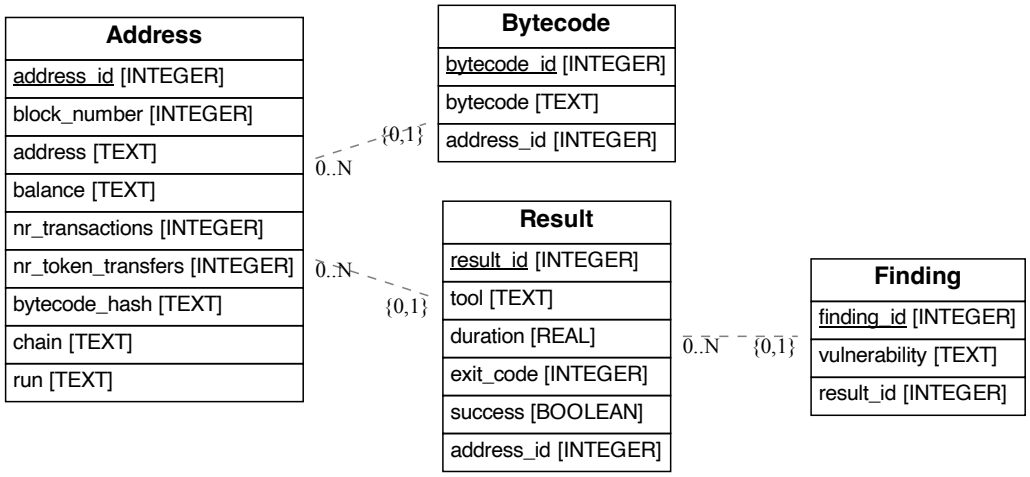


Figure 5.1: ER diagram of our *analysis* database.

5.2 Random Sampling

Although we would have preferred to capture the data seen in Section 4.2 for all the history of the two chains to make our study as comprehensive and beneficial to the community as possible, this was not feasible due to the time constraints imposed on this project. Empirical and measurement studies like this one take a long time to complete if they are thorough and provide as much data as possible. In our case, processing the execution traces of the blocks from the start of the blockchains would be an expensive and time-consuming process.

Even though we consider this to be one of the limitations of our study, we tried to mitigate it by using a different technique. As a workaround, we performed a random sampling technique where we collected random blocks throughout the history of these blockchains and performed our study on them, similar to the work of Pîrlea *et al.* [103]. We believe that with a large enough random sampling pool, the results are representative of the state of these chains in terms of vulnerabilities.

[1] https://github.com/Alexis-benoist/eralchemy

Algorithm 1, presents the technique with which we generate a random sample of block numbers from both Ethereum and BSC blockchains. As the pseudocode illustrates, it first connects to two blockchain explorers' API, namely Blockcyper [2] for Ethereum and BscScan for BSC to find the latest blocks and produce a random sample of the blockchain. The random sample could be any block from the first block after the genesis block to the last block of the blockchain. It then proceeds to write it to a file in a sorted fashion for the crawler to find. Blockcypher was chosen instead of Etherscan as it provides a JSON that is easier to process to locate the height of Ethereum. Unfortunately, the same did not apply to BscScan.

Algorithm 1 Random sampling algorithm

Input: API key ▷ api_key
Input: Sample size ▷ sample_size
Output: Eth block sample ▷ eth_block_sample
Output: Bsc block sample ▷ bsc_block_sample

$eth\_block\_sample \leftarrow set()$
$eth\_height \leftarrow request\_to\_blockcypher(api\_key)$
$bsc\_block\_sample \leftarrow set()$
$bsc\_height \leftarrow request\_to\_bscscan(api\_key)$
$blocks \leftarrow 0$
while $blocks \neq sample\_size$ **do**
    $new\_block\_eth \leftarrow random\_number\_generator(1, eth\_height)$
    $eth\_block\_sample \leftarrow add(new\_block\_eth)$
    $new\_block\_bsc \leftarrow random\_number\_generator(1, bsc\_height)$
    $bsc\_block\_sample \leftarrow add(new\_block\_bsc)$
    $blocks \leftarrow blocks + 1$
end while
$write\_block\_samples\_to\_files(eth\_block\_sample, bsc\_block\_sample)$

5.3 Block Crawler

After the database is up and running and we have the block numbers we want to traverse from the chains, it is the turn of the block crawler to iterate through the transactions of these blocks. The crawling scripts we implemented accept as arguments the file with the block numbers generated with Algorithm 1, the archive node IP address and port, the number of threads to execute the code, and a boolean value representing whether the tracing functionality will be used. We opted to write the crawling-related scripts in Golang [3], as it is the official language of the Ethereum protocol.

[2] https://api.blockcypher.com/v1/eth/main
[3] https://geth.ethereum.org/

It is noted that the code accepts as input only the archive node IP and port, which hints that the crawler can be deployed to perform the same work on other EVM-based chains as well, given that connection details to an archive node are provided. It is part of our future work to expand this study to more EVM-based chains, such as Avalanche [4].

One of the highlights of our crawler is the usage of lightweight threads to parallelise work and speed up the overall process. Threads in Golang are executed with the *go* built-in keyword and are known as *goroutines*. Of course, making the codebase thread-safe deepened the complexity during development. Nevertheless, because we noticed early on, while conducting preliminary tests, that the crawling of blockchain blocks is a time-consuming method, we decided it was worth the extra effort to incorporate threads in our code.

The last argument, tracing, essentially runs the archive node in tracing mode, requesting the node to re-execute the desired transaction. We did this because a single transaction can interact with hundreds of contracts, and tracing allows collecting data on whatever was executed by the EVM. The data we are looking for, as we will see in the next subsection, are the contract addresses involved in a transaction, if there are any. With tracing, we can find the contract addresses by tracing to where the smart contract was first deployed. Tracing an arbitrary transaction at any point in the chain's history is only doable because of our access to the archive node. Tracing is not run by default and is given as a boolean argument because it significantly slows down the process of crawling blocks.

5.3.1 Collecting Contract Addresses

As we described in Background (Section 2), a block is made out of the block header and the transactions. The latter is of particular interest to us, as this is where the smart contract address can be found. The EVM supports three types of transactions, namely, a transaction from one account to another, contract deployment, and deployed contract executions [104]. One of the challenges we faced was differentiating Externally Owned Accounts (EOA) and smart contract accounts while we processed the block transactions, as our only interest was in the latter.

One way to differentiate them is to iterate through the transactions of a block and check for transactions that have an empty (*null*) value as the destination ("to") address. This type of transaction is called *CREATE*; it indicates the creation of a contract, and from there, we can obtain the address of the newly created contract. The data field is used for the newly deployed contract EVM bytecode. A *CREATE* contract transaction can be seen in Figure 5.2. As the figure illustrates, when a contract is created in the same transaction, the EVM bytecode is contained (after it was compiled from the source code), so we can extract the bytecode simultaneously. Besides

[4] https://www.avax.network/

CREATE, we also checked for *CREATE2*[5] transactions similar to the former, but with the ability to know where the contract will be deployed.

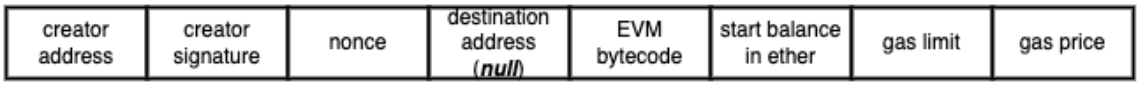


Figure 5.2: CREATE contract transaction in Ethereum.

Algorithm 2 presents the pseudocode for collecting smart contract addresses from a block. As the code illustrates, on an abstract level, *GetContractAddresses()* is the function where threads are deployed. The *blockChannel* is a structure shared between the threads. It acts as a job queue from where threads are assigned blocks. The *workerWaitGroup* structure waits for all threads to finish their jobs before continuing with the main thread only. At Line 6, the helper function *WorkerCrawlTransactions()* is called by every thread. In this function, the thread gets an unprocessed block number, acquires its data from the archive node, and goes over its transactions one by one to extract any addresses found and their individual bytecodes. Once this is done, we mass insert the collected data into the database using prepared SQL statements. These two functions do not return any data as the data is directly inserted into the database. Many details are missing in the pseudocode, including important error checking.

One of the biggest obstacles faced during this project's development was the mass insertion of data into the database. The reason was that the threads were deadlocking for an unknown reason when trying to add the block data they collected. We solved the error by passing the same database connection object to all threads instead of initialising a separate connection for each one. Also, we conducted the mass insertion right before the termination of the thread, when all blocks had been crawled instead of after each block. It took us roughly a week of trial and error to overcome this issue, as documentation for this specific error proved inadequate.

5.4 Blockchain Explorer Crawler

At this point, we have already gathered the contract addresses we want to analyse, their individual bytecodes, and the block numbers they were deployed in. What remains is to collect the balance, the number of transactions, and the number of token transfers (ERC20, ERC721, BEP20, BEP721). To collect this metadata, we implemented blockchain explorer scripts connecting Etherscan for Ethereum-deployed smart contracts and BscScan for BSC-deployed smart contracts.

The scripts are written in Python and utilise wrappers[6] that streamline the process of

[5]https://eips.ethereum.org/EIPS/eip-1014

[6]https://github.com/pcko1/etherscan-python, https://github.com/pcko1/bscscan-python

Algorithm 2 Collecting Contract Addresses Algorithm

Input: Thread Number ▷ thread_num
Input: Archive Node URL ▷ client_url
Input: Block Numbers for Crawling ▷ block_numbers
1: **function** GETCONTRACTADDRESSES($thread\_num, client\_url, block\_numbers$)
2: $blockChannel \leftarrow set(block\_numbers)$
3: $workerWaitGroup \leftarrow initialise(thread\_num)$
4: $count \leftarrow 0$
5: **while** $count \neq thread\_num$ **do**
6: $go\ WorkerCrawlTransactions(client\_url, blockChannel)$
7: $count \leftarrow count + 1$
8: **end while**
9: $workerWaitGroup.Wait()$
10: **end function**

Input: Archive Node URL ▷ client_url
Input: Block Channel ▷ blockChannel
11: **function** WORKERCRAWLTRANSACTIONS($client\_url, blockChannel$)
12: $addresses \leftarrow set()$
13: $bytecodes \leftarrow set()$
14: **while** $blockChannel$ not empty **do**
15: $block\_number \leftarrow blockChannel.pop()$
16: $block \leftarrow client\_url(block\_number)$
17: **for** every transaction in block **do**
18: **if** transaction destination is Null **then**
19: $addresses \leftarrow transaction.getaddress()$
20: $bytecodes \leftarrow address.getbytecode()$
21: **else**
22: **if** tracing mode is enabled **then**
23: $addresses, bytecodes \leftarrow TraceTransaction()$
24: **end if**
25: **end if**
26: **end for**
27: **end while**
28: $addDataToDatabase(addresses, bytecodes)$
29: **end function**

connecting to these blockchain explorers' free APIs. Similar to the block crawling of the previous section, this process also takes time to complete. Hence, we developed our codebase in an asynchronous/concurrent fashion, utilising the asyncio[7] Python module. The benefit of using the asyncio framework is that it enables API requests to pause. At the same time, they wait for their results and let other tasks run in the meantime instead of leaving the CPU idle, such as in the case of serial program execution. Concurrency aims to speed up the overall performance of input/output (I/O) bound programs, whose performance can be dramatically slowed when they are obliged to wait for I/O operation from an external resource frequently. In our case, I/O are considered the web requests and database transactions, which bottleneck the whole process. As with block crawling, it involves doing extra work upfront, but the benefit, in our case, vindicated the additional effort.

The pseudocode for the smart contract metadata collection can be seen at Algorithm 3. The algorithm takes as input a database object, the blockchain name out of ETH and BSC, and the API key for the respective blockchain explorer. The first step required a list of all the addresses crawled at an earlier stage from the database. Then, we performed an asynchronous request to the blockchain explorer for each address to obtain the abovementioned metadata. Before sending the request for each address, we checked whether the free API limits were violated and waited a second before sending a new request. We incorporated the limit checking methodology as the API will drop any request that breaches its limit, and thus we would have to rerun the scripts. Furthermore, any data returned from the blockchain explorers is in a JSON format, requiring further processing before storing it. Finally, the algorithm returns nothing but directly stores the collected data in the database.

5.5 Analysis

At this point, the database is populated with all the necessary data we need to perform our tests. Before moving on to the SmartBugs execution, we needed to write in files the bytecodes of the smart contracts we wished to analyse. The constraints for extracting a bytecode from the database and writing it into files were thoroughly discussed in Subsection 4.4.3. All bytecodes that are written into files are unique.

Our corpus of smart contracts was ready for SmartBugs to analyse at this juncture. However, we needed to modify SmartBugs first. The modifications we made were adding configurations for the analysis tools we wanted to include in our study and adding support for bytecode since it only supported source code analysis. For the latter, the SmartBugs team explained this feature is in progress if we needed to work on it and offered guidance on how to work with it for this study. We also modified the framework to accept a dataset we defined.

After SmartBugs finishes execution, it creates a directory containing all the results

[7]https://docs.python.org/3/library/asyncio.html

Algorithm 3 Collecting Contract Metadata Algorithm

Input: Database Object ▷ db_obj
Input: API Key ▷ api_key
Input: Chain ▷ chain

```
addresses ← connect_to_db(db_obj, chain)
for every address in addresses do
    limit_checker()
    balance ← get_balance(api_key, address)
    nr_transactions ← get_transaction_count(api_key)
    if chain is ETH then
        erc20 ← get_erc20(api_key)
        erc721 ← get_erc721(api_key)
        nr_token_transfers ← erc20 + erc721
    else
        bep20 ← get_bep20(api_key)
        bep721 ← get_bep721(api_key)
        nr_token_transfers ← bep20 + bep721
    end if
    store_to_db(address, balance, nr_transactions, nr_token_transfers)
end for
```

of its execution for each analysis tool. For every tool, we implemented a parser that interprets these results and prints the reported vulnerabilities according to the extended DASP10+ and SWC taxonomies, total execution times, average contract execution time, and a description of the analysis tool and a description of any vulnerability reported. The main parsing script is configured to run the parsers of every tool at once for convenience.

Algorithm 4 illustrates the aforementioned process simplistically and abstractly.

Algorithm 4 Analysis Algorithm

Input: Database Object ▷ db_obj
Input: SmartBugs Framework ▷ smartbugs

```
rows ← connect_to_db(db_obj)
for every row in rows do
    if row.balance or row.nr_transactions or row.nr_token_transfers > 0 then
        files ← write_to_filesystem(row.bytecode)
    end if
end for
results ← run_smartbugs(files)
parse_results(results)
```

Although our repository contains an elaborate *README* file with all the necessary instructions to execute the above steps, we added everything into a single shell file

with necessary checks and helpful output to make the replication process more convenient. This file is at the heart of our tool Centaur, as with one command, you can execute the whole infrastructure we created for this study and obtain insight into the vulnerability state of EVM-based blockchains. Furthermore, all of the arguments the script takes are passed in the form of a file; the user only has to update the variables in the config file. Finally, after connecting the various pieces of Centaur, we achieve the control flow chart seen in Figure 5.3.

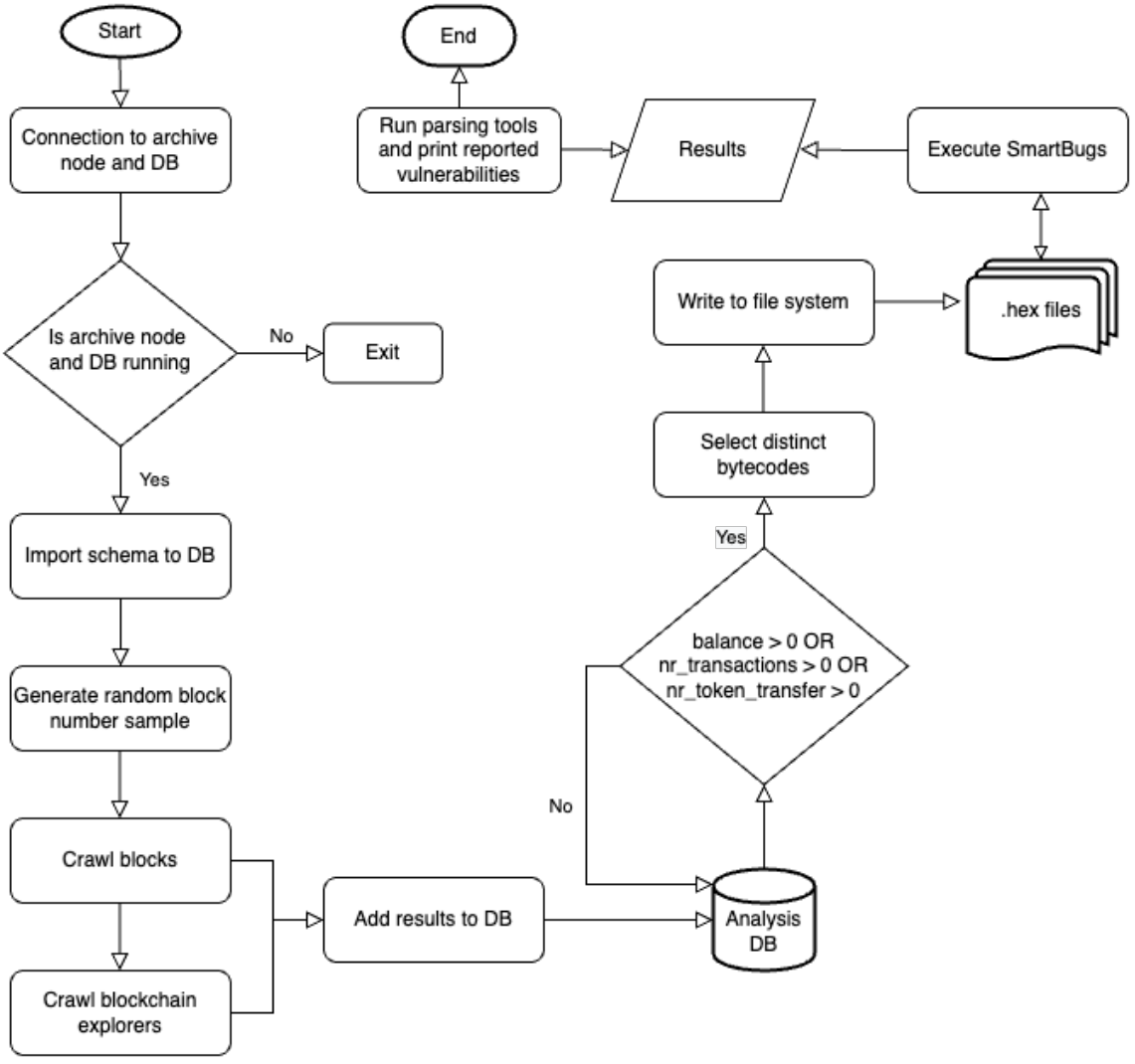


Figure 5.3: Control Flow Chart of Centaur's execution.

Summary: This chapter on implementing the Centaur framework outlines the fundamental technology behind this tool to collect and analyse smart contract bytecodes. Furthermore, it gives a detailed picture of how smart contract bytecodes are gathered and inspected. This architecture can provide insight into the potential vulnerabilities of smart contracts deployed on EVM-based blockchains.

Chapter 6

Results

This chapter examines the core findings of the Centaur framework when executed on a corpus of deployed smart contract bytecodes of the Ethereum and Binance Smart Chain Mainnets. Then, we provide an extensive evaluation of these results that explore the vulnerability state of the two blockchains. We first asses the details of our experimentation before providing essential statistics on our dataset. Finally, we comprehensively answer the three research questions we posed in the introductory Chapter 1.

6.1 Experimentation Details

For evaluating our framework, we executed the process explained in Section 4.4 twice, collecting a random sample of 2,000 blocks crawled from the Ethereum and BSC blockchains each time. We separated the experiments into two batches of 1,000 blocks to validate the results between the two sets and thus reinforce confidence in our findings. It reduces the doubt of bias caused by the random sampling procedure we used to collect the block numbers from the two blockchains.

The first batch of 1,000 blocks (for each chain) contained 4,048 smart contract addresses deployed on Ethereum and 3,896 on BSC. As these contracts were too many to analyse, we only kept those that were useful. Of these, only 123 unique smart contracts passed the constraints set in Section 4.4.3 from Ethereum and 110 from BSC. To reach this number of smart contracts, we filtered those with a balance, number of transactions or token transfers bigger than zero. The nine state-of-the-art analysis tools from Section 4.1 reported 343 and 272 vulnerabilities for Ethereum and BSC contracts, respectively.

In the second batch of 1,000 blocks (for each chain) crawling the ETH and BSC archive nodes, we gathered a total of 5,123 and 6,460 smart contract bytecodes, respectively. Of these, only 191 on ETH and 103 on BSC past the required parameters. The same analysis tools reported 421 and 321 vulnerabilities, respectively.

The difference in smart contract numbers from the first and second batch of byte-

codes is down to the randomness in sampling block numbers, and each block has a varying number of smart contract deployment (*CREATE*) transactions. The vulnerability numbers reported do not suggest a significant variance between them for us to ascertain any conclusions or insights are biased. Ethereum vulnerabilities differ 23% in total and BSC vulnerabilities only 18% between the two runs. In the second run, the unique smart contract number that passes the criteria for Ethereum increased by 55%, and BSC decreased by 6%. In both runs, Ethereum has more vulnerabilities in smart contracts than BSC.

The technological artefact has been tested on a 64-bits 20.04 Ubuntu machine and an Apple M1 Mac mini 12.3.1, both with 8 cores and 16GB of RAM. However, our Docker image[1] can be used on any machine that has Docker installed. The rest of the Results chapter focuses on the two runs as a whole, so we explore the type of vulnerabilities that proliferate in smart contracts in detail.

6.2 Dataset Analysis

After combining and storing all the data from the two runs into our database, we executed various aggregated SQL queries to get descriptive statistics on our dataset as a whole, as seen in Figure 6.1. As the figure illustrates, we applied our approach to 2,000 blocks on Ethereum and 2,000 blocks on Binance Smart Chain, equaling 4,000 crawled blocks. Within these blocks, 7,236 smart contracts were deployed on Ethereum and 10,356 on BSC. Furthermore, 899 smart contracts have a balance above 0 or at least one transaction or token transfer on Ethereum and 327 on BSC, at only 12% and 3% of their total respective bytecode corpus. Finally, only 268 (4%) and 205 (3%) of the contracts are unique, meaning that 17,119 (97%) are duplicates. Any percentages in the figure are rounded to the closest integer number.

6.2.1 Self-Destructed Contracts

Another noteworthy fact about this dataset is the sizeable number of self-destructed contracts. As the bar chart in Figure 6.1 illustrates, there were a total of 772 (4.4%) self-destructed contracts, from which 706 belonged to Ethereum and 66 to BSC at 91% and 9% of the self-destructed corpus, respectively. We detected self-destructed contacts in our dataset by searching for empty bytecode strings. We also noticed that 541 self-destructed contracts (70%) are spread into only 10 blocks out of the 4,000 we crawled. During the rest of the experimentation, we excluded the self-destructed contracts when calculating percentages out of the total amounts of contracts on each chain. As we described in our limitations in Section 7, we do not apply special handling during block crawling in cases where we recovered a self-destructed contract.

[1] mchara01/centaur:1.0

Ethereum Crawled Blocks	2,000
BSC Crawled Blocks	2,000
Ethereum Bytecodes	7,236 (41%)
BSC Bytecodes	10,356 (59%)
Total Bytecodes	17,592
Ethereum	
Duplicate bytecodes	6,783 (94%)
Unique bytecodes	453 (6%)
Bytecodes with a balance above 0	60 (1%)
Bytecodes with at least one transaction	882 (12%)
Bytecodes with at least one token transfer	46 (1%)
Bytecodes without a transaction, token transfer or balance	6,337 (88%)
Bytecodes with a balance above 0 or at least one transaction or token transfer	899 (12%)
Binance Smart Chain	
Duplicate bytecodes	10,124 (98%)
Unique bytecodes	232 (2%)
Bytecodes with a balance above 0	13 (0%)
Bytecodes with at least one transaction	298 (3%)
Bytecodes with at least one token transfer	175 (2%)
Bytecodes without a transaction, token transfer or balance	10,029 (97%)
Bytecodes with a balance above 0 or at least one transaction or token transfer	327 (3%)
Ethereum unique bytecodes that pass constraints	268 (4%)
BSC unique bytecodes that pass constraints	205 (2%)
Total unique bytecodes that pass constraints	473 (3%)
Total duplicate bytecodes	17,119 (97%)

Table 6.1: Statistics on the collection of deployed smart contract bytecodes from the Ethereum and BSC blockchains.

6.2.2 Gas Token Contracts

After inspecting the dataset more, we also found another enormous number of contract types that would be an oversight for us not to discuss – the gas token contracts[2]. This type of contract allows users to store gas by tokenising it at a relatively low price and using it when its price rises, thus saving money on gas. This is done as gas prices on EVM-based chains can fluctuate and become expensive, especially at peak times. Those who use gas tokens can use higher gas prices without paying correspondingly higher fees and gain priority in their transactions. Judging by the number of gas token contracts, 11,993 deployed on both chains; they are customary practice within the blockchain community. They comprise 68% of the total bytecode corpus.

We detected this type of contract while manually inspecting bytecodes, with many of

[2]Fundamentals of Gas Tokens - https://blog.openzeppelin.com/fundamentals-of-gas-tokens/

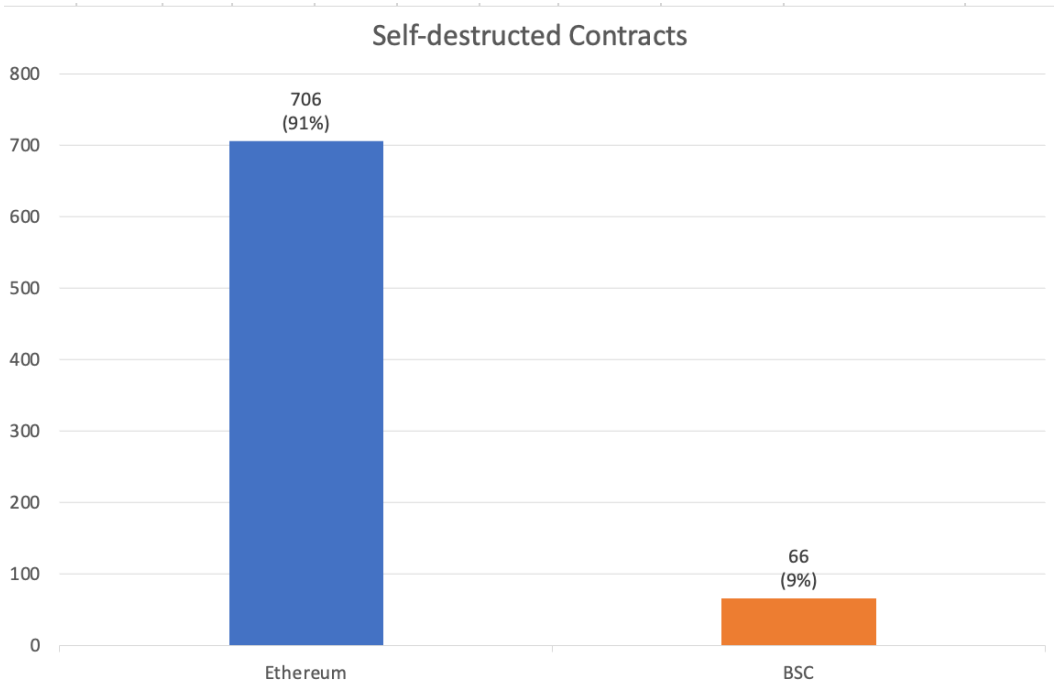


Figure 6.1: Proportion of self-destructed contracts on each chain.

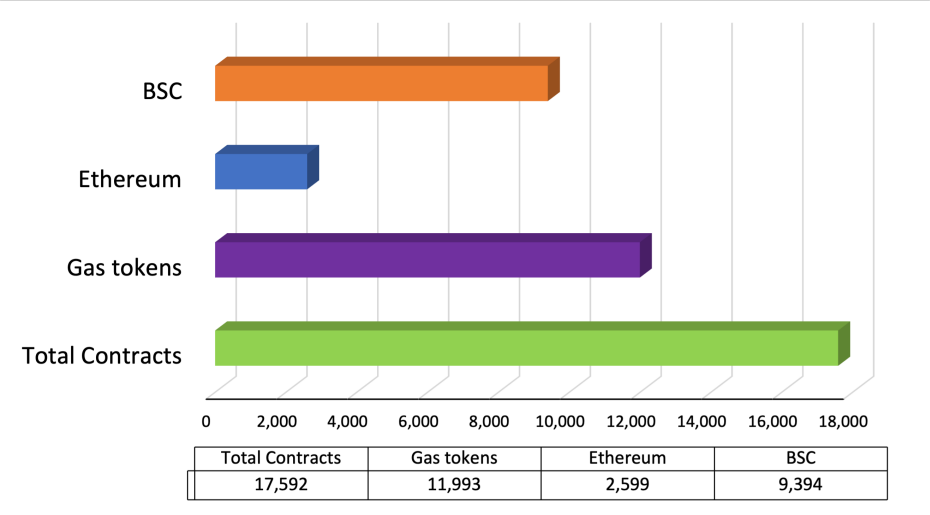
them short and ending with the self-destruct functionality (opcode *FF* in bytecode). After doing some research on these contracts, we learnt about gas tokens and, specifically the GasToken.io[3] [105] and Chi gastoken[4] contracts. Since the contracts must contain the address of either GasToken.io or Chi gastoken contract, we searched which bytecodes ended with the FF opcode and contained any of the addresses of these two contracts. An example of a contract we recovered from our dataset that contained the address of Chi gastoken can be seen in Figure 6.4. The results are shown in Figures 6.2 and 6.3. Looking at the first figure, we can see that far more gas tokens are deployed on BSC than on Ethereum, and more than two-thirds of the total are gas token contracts. The second figure shows the ratio of the GasToken and Chi gastoken contracts. The Chi gastoken contract is deployed on both chains, whereas GasToken is deployed exclusively on Ethereum.

6.3 RQ1: Vulnerability State

Moving on to our first and most vital research question, we wanted to know how the states of Ethereum and Binance Smart Chain compare in terms of vulnerabilities. We crawled 4,000 blocks from Ethereum and BSC archive nodes starting with the data collection phase. We then collected the metadata for 17,592 smart contracts from Etherscan and BscScan, with the methodology described in Subsection 4.4.1. To gather the metadata, we made 31,586 requests to BscScan and 15,321 to Ether-

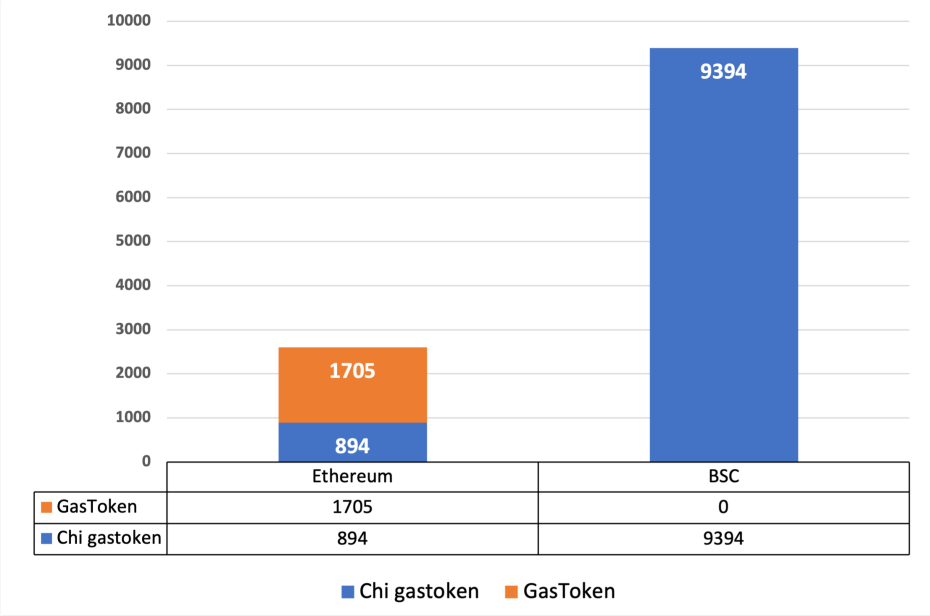
[3]https://etherscan.io/address/0x0000000000b3F879cb30FE243b4Dfee438691c04

[4]https://etherscan.io/token/0x0000000000004946c0e9F43F4Dee607b0eF1fA1c (same address on Bscscan)



Total Contracts	Gas tokens	Ethereum	BSC
17,592	11,993	2,599	9,394

Figure 6.2: Number of gas token contracts on each chain, and in total, compared to the total number of contracts.



	Ethereum	BSC
GasToken	1705	0
Chi gastoken	894	9394

Figure 6.3: Proportions of Chi gastoken and CasToken.io for each chain.

```
function _fallback() payable {
  require(caller ^ 0x4946c0e9f43f4dee607b0ef1fa1c == 0)
  selfdestruct(caller)
 }
```

Figure 6.4: Example of contract created by Chi gastoken.

scan. The entire collection phase took nearly 10 hours to complete. A breakdown of the execution times during this phase can be seen in Table 6.2. After the table is examined, it is reasonable to assume that BSC blocks contain more transactions than Ethereum blocks since the time for processing the transactions of Ethereum blocks was 0.08% of that of BSC blocks. Also, the number of smart contracts extracted from BSC blocks is higher.

	Execution Times
Crawling 2,000 blocks on BSC	01:40:05
Crawling 2,000 blocks on Ethereum	00:07:38
Collecting metadata for 10,356 contacts from BscScan	03:41:45
Collecting metadata for 7,236 contacts from Etherscan	04:24:12
Total	**09:53:40**

Table 6.2: Execution time for the data collection phase.

After finishing the data collection phase, we moved to dataset processing, which comprised running SQL queries to extract the files used in practice and write them to the local file system. This part of the procedure took a few seconds to complete, so we did not calculate it in our overall execution time.

The results analysis phase followed. The flow we followed for this phase can be seen in more detail in Subsection 4.4.4. The total execution times and average time to analyse a contract for the nine automated analysis tools on the corpus of 527 unique bytecodes during the analysis phase can be seen in Table 6.3. The fastest executing tool was MadMax, taking only three seconds on average to analyse a contract and almost 33 minutes to analyse the whole dataset. On the other hand, Mythril takes significantly longer than the rest of the tools to analyse a single contract and the whole corpus, at eight minutes for the former and more than three days for the latter. It was expected as Mythril can detect more vulnerability classes (11) than any other tool and also employs Symbolic Execution, Taint Analysis and during analysis which are Satisfiability Modulo Theories time-consuming.

The same table also shows the number of 30-minute timeouts each tool had on our corpus. Only three tools experienced timeouts, namely Conkas, Osiris, and Securify, totalling 95 timeouts. It meant that from the five-day experiment, 39% of this time was spent on timeouts.

The number of unique bytecodes analysed was not balanced between the two chains, with Ethereum having a larger amount. However, the time taken in total for analysing BSC-deployed smart contracts was more than a day longer than Ethereum-deployed smart contracts. This led us to conclude that BSC-deployed smart contracts are generally more complex because they need this much time to be analysed. It is supported by the fact that the average time analysing a contract is 12 seconds, whereas

Ethereum is eight seconds.

No.	Tool Name	Execution Times (days hh:mm:ss) Avg / contract	Total	Timeouts
1	Conkas	0:01:12	10:37:04	9
2	HoneyBadger	0:00:21	3:05:17	0
3	MadMax	0:00:03	0:32:46	0
4	Maian	0:00:15	2:19:25	0
5	Mythril	0:08:17	3 days, 0:46:29	0
6	Osiris	0:00:11	1:44:46	1
7	Oyente	0:00:06	0:58:42	0
8	Securify	0:01:53	16:40:56	85
9	Vandal	0:00:14	2:11:10	0
	Ethereum	0:07:53	1 day, 17:19:16	
	BSC	0:11:38	2 days, 21:37:23	
	Total		**4 days, 14:56:39**	**95**

Table 6.3: Execution time for the result analysis phase along with the number of 30-minute timeouts each tool experienced.

The final stage of mapping the result to the DASP10+ and SWC classifications was completed in under a minute, as our parsing scripts are designed to work quickly. The total time it took for us to complete all phases (i.e., data collection, data storage, dataset creation, result analysis) and conduct our experiments was **5 days 0 hours 50 minutes 19 seconds**.

The results of the analysis are illustrated in Table 6.4 for BSC and Table 6.5 for Ethereum. The tables show the number of vulnerabilities for each DASP10+ vulnerability class and its respective SWC ID reported by each analysis tool. The cells map the vulnerability groups with analysis tools containing two numbers. The first number shows the potential vulnerabilities in unique bytecodes, and the second shows the same vulnerabilities when considering duplicate bytecodes. Hence, the second number is always the same or more than the first. Even though the analysis was done on unique bytecodes to avoid wasting time analysing the same bytecodes, the duplicate bytecode numbers are what we care about more as the blockchains contain duplicate smart contracts that are still prone to attacks. Any cells with "-" inside indicate that the tool does not detect the vulnerability class.

However, since the number of smart contracts analysed on each chain was not the same (268 for Ethereum and 205 for BSC), we opted for a fairer comparison between the two by calculating the number of vulnerabilities per 100 smart contracts for each and then making comparisons. This result is shown in Figure 6.5. For calculation of the number of vulnerabilities per 100 smart contracts, we used the formula:

DASP10 (SWC-ID)	Conkas	MadMax	Maian	Mythril	Osiris	Oyente	Vandal	Total	Total w/ duplicates
1.Reentrancy (SWC-107)	30/31	-	-	12/12	0/0	0/0	99/147	**141**	**190**
2.Access Control (SWC-105, 106, 115)	-	-	2/255	4/258	-	-	6/260	**12**	**773**
3.Arithmetic Issues (SWC-101)	55/89	0/0	-	42/43	42/96	-	-	**139**	**228**
4.Unchecked Return Values For Low Level Calls (SWC-104)	2/2	-	-	1/1	-	-	116/171	**119**	**174**
5.Denial of Service (SWC-115)	-	1/1	-	8/8	-	0/0	-	**9**	**9**
6.Bad Randomness (SWC-120)	-	-	-	28/82	-	-	-	**28**	**82**
7.Front-Running (SWC-114)	2/2	-	-	0/0	1/2	0/0	-	**3**	**4**
8.Time manipulation (SWC-116)	23/77	-	-	0/0	0/0	0/0	-	**23**	**77**
9.Short Address Attack (N/A)	-	-	-	-	-	-	-	**0**	**0**
10.Honeypots (N/A)	-	-	-	-	-	-	-	**0**	**0**
11.Non-Isolated External Calls (SWC-126)	-	0/0	-	-	-	-	-	**0**	**0**
12.Greedy (N/A)	-	-	65/70	-	-	-	-	**65**	**70**
13.Assert Violations (SWC-110)	-	-	-	18/18	-	-	-	**18**	**18**
14.Delegate Call To Untrusted Contract (SWC-112)	-	-	-	1/1	-	-	-	**1**	**1**
15.Write to Arbitrary Storage Location (SWC-124)	-	-	-	2/2	-	-	-	**2**	**2**
16.Hardcoded Gas (SWC-134)	-	-	-	0/0	-	-	-	**0**	**0**
17.Jump to an arbitrary instruction (SWC-127)	-	-	-	1/1	-	-	-	**1**	**1**
18.Callstack Depth Attack Vulnerability (N/A)	-	-	-	-	-	32/84	-	**32**	**84**
Total	**112**	**1**	**67**	**117**	**43**	**32**	**221**	**593**	
Total w/ duplicates	**201**	**1**	**325**	**426**	**98**	**84**	**578**		**1713**

Table 6.4: BSC vulnerability state. Each cell shows the number of potential vulnerabilities in unique bytecodes and in duplicate bytecode, separated by a /. When a cell contains a - means that the vulnerability class is not detected by the respective tool. Analysis tools HoneyBadger and Securify have been removed as they did not report any vulnerabilities. Figure B.2 in Appendix B shows the same table with these two tools included also.

DASP10 (SWC-ID)	Conkas	MadMax	Maian	Mythril	Osiris	Oyente	Vandal	Total	Total w/ duplicates
1.Reentrancy (SWC-107)	56/1264	-	-	34/433	7/406	8/302	146/1220	**251**	**3625**
2.Access Control (SWC-105, 106, 115)	-	-	3/62	8/75	-	-	11/13	**22**	**150**
3.Arithmetic Issues (SWC-101)	68/1370	1/1	-	20/23	95/123	-	-	**184**	**1517**
4.Unchecked Return Values For Low Level Calls (SWC-104)	5/66	-	-	2/9	-	-	142/1320	**149**	**1395**
5.Denial of Service (SWC-115)	-	2/2	-	12/404	-	-	-	**14**	**406**
6.Bad Randomness (SWC-120)	-	-	-	16/21	-	-	-	**16**	**21**
7.Front-Running (SWC-114)	1/1	-	-	0/0	10/537	21/446	-	**32**	**984**
8.Time manipulation (SWC-116)	14/27	-	-	0/0	0/0	2/2	-	**16**	**29**
9.Short Address Attack (N/A)	-	-	-	-	-	-	-	**0**	**0**
10.Honeypots (N/A)	-	-	-	-	-	-	-	**0**	**0**
11.Non-Isolated External Calls (SWC-126)	-	0/0	-	-	-	-	-	**0**	**0**
12.Greedy (N/A)	-	-	12/37	-	-	-	-	**12**	**37**
13.Assert Violations (SWC-110)	-	-	-	49/664	-	-	-	**49**	**664**
14.Delegate Call To Untrusted Contract (SWC-112)	-	-	-	0/0	-	-	-	**0**	**0**
15.Write to Arbitrary Storage Location (SWC-124)	-	-	-	0/0	-	-	-	**0**	**0**
16.Hardcoded Gas (SWC-134)	-	-	-	0/0	-	-	-	**0**	**0**
17.Jump to an arbitrary instruction (SWC-127)	-	-	-	1/4	-	-	-	**1**	**4**
18.Callstack Depth Attack Vulnerability (N/A)	-	-	-	-	-	18/25	-	**18**	**25**
Total	**144**	**3**	**15**	**142**	**112**	**49**	**299**	**764**	
Total w/ duplicates	**2728**	**3**	**99**	**1633**	**1066**	**775**	**2553**		**8857**

Table 6.5: Ethereum vulnerability state. Each cell shows the number of potential vulnerabilities in unique bytecodes and in duplicate bytecode, separated by a /. When a cell contains a - means that the vulnerability class is not detected by the respective tool. Analysis tools HoneyBadger and Securify have been removed as they did not report any vulnerabilities. Figure B.1 in Appendix B shows the same table with these two tools included also.

$$Vulns\ /\ 100\ SC\ = \frac{total\ vulns\ in\ class\ w/\ duplicates\ \times\ Unique\ SC\ that\ pass\ reqs}{100}$$

After executing the nine security analysis tools via the Centaur framework and examining its process results, it was apparent, based on our random sample of blocks on both chains, that smart contracts deployed on the Ethereum blockchain contains significantly more potential vulnerabilities than their BSC counterpart. More specifically, for every 100 smart contracts in our corpus, including duplicates, which are deployed on Ethereum contain, on average, 3,305 vulnerabilities, whereas smart contracts deployed on BSC contain 836 vulnerabilities.

On Ethereum, the most prevalent vulnerability class is by far the Reentrancy (category 1), responsible for 41% of all vulnerabilities at 1353, followed by Arithmetic Issues (category 3) at 17% with 566. The third most frequent vulnerability class is Unchecked Return Values For Low-Level Calls (category 4), accountable for 16% of all reported vulnerabilities at 521.

On the other hand, the most common vulnerability group on BSC is Access Control (category 2), responsible for 45% of all vulnerabilities at 377. In second place is Arithmetic Issues (category 3) at 13% with 111, followed by Reentrancy (category 1), accountable for 11% of all reported vulnerabilities at 93.

The Arithmetic Issues and Reentrancy vulnerability classes are the top two most frequent vulnerabilities on Ethereum and Binance Smart Chain. Together they are accountable for 24% of the vulnerabilities on BSC and 58% of vulnerabilities on Ethereum. Interestingly, the Access Control class, which is the most frequent on BSC, is ranked 7th on Ethereum. This vulnerability class exists in large numbers on other types of programs as well, ranked as the top vulnerability among web applications[5].

More elaborately, there is a balance in the number of vulnerability classes of the extended DASP10 between the two chains. More accurately, seven categories (1, 3, 4, 5, 7, 13, 18) prevail on Ethereum rather than BSC, whereas seven other categories (2, 6, 8, 12, 14, 15, 17) are more widespread on BSC. Furthermore, four categories (9, 10, 11, 17) reported zero vulnerabilities on both blockchains. Intriguingly, vulnerability classes Write to Arbitrary Storage Location (15) and Hardcoded Gas (16) are reported only on BSC, although in small numbers, with one and two potential vulnerabilities, respectively.

6.3.1 Analysis Tool Discussion

Although the aim of this thesis is not to explore the efficiency and performance of automated analysis tools, we believe it would go amiss not to discuss some key facts

[5]https://owasp.org/www-project-top-ten/

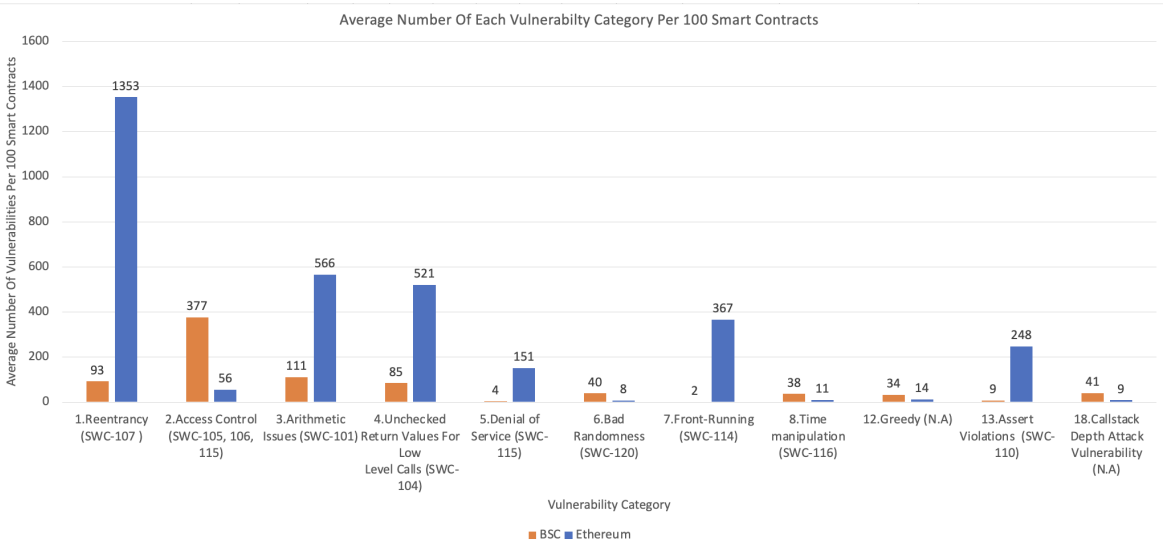


Figure 6.5: Average number of each vulnerability class per 100 smart contracts deployed on Ethereum and BSC. Categories 9, 10, 11, 14 and 16 are removed from the bar chart as they have zero vulnerabilities reported for both chains. Categories 15 and 17 were removed as well as they have only one potential vulnerability per 100 smart contracts with both chains combined. The same figure in a larger size can be seen in Figure B.3 of Appendix B.

regarding their analysis on our dataset.

Intriguingly, none of the bytecode automated analysis tools can detect vulnerability category nine of the DASP10, the *Short Address Attack*. This could be because the vulnerability is detected on the EVM level, and none of the tools is designed to work on that level, apart from Mythril and Oyente that can detect the other two vulnerability classes (15 and 18). Most analysis tools operate on the Solidity and Blockchain (block) level. With this in mind, future research should focus on filling this gap and developing more tools that work on the EVM level to secure smart contracts from all possible attacks.

Also, HoneyBadger is the only tool designed to detect only one vulnerability in smart contracts, honeypots. However, from our results, we can safely say this type of vulnerability is uncommon among deployed smart contracts, as HoneyBadger did not flag any contracts. More interestingly, honeypots are not the only category that only a single tool can detect. Categories 10 to 18, excluding 16, can only be detected by one analysis tool. Such information enables us to conclude that assessing a smart contract with multiple analysis tools is paramount before deploying it. Doing so would cover as many vulnerability classes as possible and minimise the room for costly errors.

Moreover, some analysis tools perform poorly by not reporting many or any vulnerabilities. Securify has not flagged any contract from our corpus as potentially vulnerable, and Madmax reports one on BSC and three on Ethereum. We could not

pinpoint the reason for Securify not reporting anything. For MadMax, we inspected the code and found that the tool looks for specific patterns of the three vulnerabilities it searches for (categories 3, 5, 12) but misses other vulnerabilities that still fall into these groups but don't follow those patterns.

In general, there are wide discrepancies in the number of vulnerabilities reported by the analysis tools. For example, Conkas, Mythril and Osiris detect four vulnerabilities in common between them (categories 1, 3, 7, and 8), but as the results illustrate, their numbers do not comply. It is noted the same does happen with other tools as well. The lack of consensus between the tools makes us suspicious about the number of false positives and false negatives these tools generate. However, establishing a ground truth for such a sizeable corpus is a difficult and time-consuming action. Therefore, we suggest the efficiency of these tools be re-evaluated and improved by future research in the field as smart contract security pivots on analysis tools.

6.4 RQ2: Potential Balance at Stake

In this section, we study the impact severity of vulnerabilities by calculating the balance of smart contracts with reported vulnerabilities in both chains. Consequently, we searched the database to find the contracts in our dataset flagged as vulnerable by the analysis tools and extracted their balances. Then, we developed a Python script that summed the balances of all these contracts and calculated the total amount in Wei and Ether for Ethereum and Jager and BNB for BSC. The conversion was achieved using an external Python module[6]. Finally, we got the conversion rate from Ether and BNB to USD to find the balance in dollars at stake to answer the research question. To obtain the conversion rate, we use the API of CryptoCompare[7].

Ethereum balance in Wei	972350019151223217
Ethereum balance in Ether	0.97235
Ethereum balance in USD	**$1,612.91**
BSC balance in Jager	7108496674752441248
BSC balance in BNB	7.10849
BSC balance in USD	**$2,113.49**

Table 6.6: Balance of deployed smart contracts with vulnerabilities reported on Ethereum and BSC in their respective currency and US Dollars.

The results from the above process are shown in Table 6.6. The numbers show that a relatively small amount of money is exposed. More precisely, just under one Ether is prone to be taken by malicious adversaries and just seven BNB. This amounts to

[6]https://github.com/ethereum/eth-utils
[7]https://min-api.cryptocompare.com/

$1,612 and $2,113 on Ethereum and BSC, respectively. Although this amount does not seem significant, we must remember these numbers represent a small random corpus of 2,064 smart contracts that have been flagged as potentially vulnerable (i.e., have at least one vulnerability reported). When accounting for the vulnerabilities of every deployed smart contract on the two chains, these amounts could become astronomical. Furthermore, the numbers shown in the figure are at the time of writing (late August 2022).

6.5 RQ3: Smart Contract Duplication

In this section, we deviate from the smart contracts vulnerability perspective and study a different metric that has also not been studied before. On the third and final research question, we want to find the number of smart contract duplicates on each chain and the cross-chain duplicates. Regarding the number of duplicates, as Figure 6.1 depicts that a total of 97% (17,119) of all bytecodes in our dataset are duplicates, leaving only a fraction of 3% (473) as unique. Categorically, 6,783 out of 7,236 (94%) of contracts collected from Ethereum and 10,124 out of 10,356 (98%) of contracts collected from Binance Smart Chain are duplicates. This high frequency of duplicate contracts corroborates with Durieux *et al.* [12] findings were out of the 972,975 Solidity smart contracts they gathered; only 47,518 were unique, meaning that 95% were duplicates.

After witnessing such a high percentage of duplicates on both chains, we wanted to discover how many smart contracts are copied between chains since both chains are EVM-based and smart contracts could be copied effortlessly. For measuring the smart contract cross-chain interoperability, we selected from our dataset all contracts with a bytecode that appears on both chains and excluded the self-destructed bytecodes, which we have shown in Section 6.2.1 appear in large numbers on both chains. Surprisingly, only the four smart contracts seen in Table 6.7 were deployed on both chains.

BSC Address	Occ.	Ethereum Address	Occ.	Bytecode (SHA-256)
0x4DA72E275aCFC5df8014e51c9369A72e59a878E2	5	0xDb5439D027CEc9570f864D9F138fa69e11C2f37B	2	0e047651c4ee6df4dbdc1630117c6604 06a8823f0b71f547b5935a7cb6ff4cc4
0x353205ccC78d551D5559816f63fC6366c07e7a13	7	0x83476Fee436EbdD68F6E5f28701c99DD962135f2	1	5df205883b26d7831d6276f020338883 cdbe955f719f7de39a8934eaf3794105
0xBD5bb1C28e70b6e56227Ee5Cd88BEaE28c1a5AE2	9,393	0xb296376556aAa5e6744fF630d89088408451754A	894	c9110d06a1991a9c6b9a0668cacad7439 181433f56847dec6f35eb6ce3cb4d10
0x8A1d0a7DfE2EC135486B481Bd03418c2661Fb9dD	1	0x3A44D6B7F2D79ac803B8B17DEa7e08edE15D3b93	2	de2d2aae4aa6c9b83c09570603512a7a ae26a5ecce751fe89f82e25f104ee91a

Table 6.7: Smart contracts that exists on both Ethereum and BSC along with their occurrences for each chain. The third bytecode with the high number of occurrences on both chains is a Chi Gastoken contract.

After exploring these smart contracts using Etherscan and BscScan, the first and fourth contracts are proxies, the third is a Chi gastoken contract like the one from Subsection 6.2.2, and the second one is a normal contract. What is interesting is the proxy contracts which pass on calls to another contract. For interacting with some

contracts, first, you must go through the proxy that will redirect you to the target. This pattern provides upgradeability as the target contract can change, and we only need to change its address in the proxy contract.

Although the number of identical contracts on both chains is small, we still assert the actual amount is much larger, even though the random samples of blocks we collected do not reflect this. We stand by this belief as contracts can be copied between EVM-compatible chains with minimal changes made to them. Also, some contracts may not be identical as they may be changed slightly before being copied to other chains but still fill the same purpose. If instead of using bytecode equality to answer this research question, we performed a bytecode similarity analysis where two bytecode could be considered identical without being 100% the same, the outcome could be much higher.

Summary: Exactly 4,000 blocks were crawled on Ethereum and BSC archive nodes. BSC blocks contained more smart contract deployments and took longer for their transactions to be crawled, which indicates they have a higher number of transactions within a block than Ethereum. Our findings show that potential losses in unsafe smart contracts are nearly $4,000 for our random smart contract corpus. The analysis shows that some available tools perform poorly by not reporting vulnerabilities, whereas others report a vast amount. In general, there are wide discrepancies between the analysis tools and the number of vulnerabilities they discover. The number of duplicate contracts within a chain is high, with 94% duplicates on Ethereum and 98% on BSC. In total, only 3% of smart contracts were unique, while 4.4% were self-destructive. Moreover, the number of identical contracts on both chains is small, only four smart contracts, but this could be due to the random sample we got; the actual number could be much higher. In addition, the number of vulnerabilities in smart contracts deployed on Ethereum is almost four times higher than those deployed on BSC, at 3,305 and 836 on average per 100 smart contracts, respectively. Finally, the top three vulnerability classes that prevail on each chain are different. More research is needed to improve analysis tools' efficiency to make smart contracts less vulnerable.

Chapter 7

Discussion

In the discussion chapter, we outline in detail the limitations we faced during the development and execution of Centaur. There are deliberations on future plans and what is under consideration for upgrading and improving our framework and empirical study. Moreover, we discuss the threats to the validity of our results and mitigation strategies.

7.1 Limitations

During the development of this project, we faced various obstacles that must be addressed to produce a more comprehensive study and framework. The following are the most prevalent limitations of this project.

Our first major obstacle was the lack of time. It is common knowledge that empirical studies, such as this, require much time to obtain the results, process them and evaluate the findings. For instance, executing the SmartBugs framework on 47,518 unique Solidity smart contracts deployed on Ethereum takes over 564 days. Although we would much prefer to follow a similar approach on bytecodes deployed on Ethereum and BSC, unfortunately, the allocated time for this project was inadequate for completing such a large-scale and comprehensive study. However, the infrastructure (Centaur) for redoing our work over a longer period and on other EVM-based chains is complete and ready for access.

Our approach in Section 5.3 for crawling blocks has one significant limitation. When we recover bytecodes throughout the history of the blockchain, we do not extract the original bytecodes of *self-destructed* contracts. The self-destruct function (`opcode FF`) removes the bytecode of smart contracts from the contract address and sends all the balance stored to a specified address. An empty bytecode is placed at that address, indicating the contract has self-destructed. Chen *et al*. [106] has shown that on Ethereum, 2,786 out of the 54,739 (5.1%) verified smart contracts from Etherscan contains the self-destruct function. In our dataset, we detected 706 contracts that self-destructed on Ethereum and 66 on BSC, nearly 5% of all contracts. This in-

dicates that self-destructed contracts are a sizeable portion of the dataset and should be treated as having no value.

Moreover, previous work [12, 60] has shown that most of the analysis tools included in this study are inaccurate, containing many false positives - falsely reporting vulnerabilities when these do not exist. Our findings corroborate with their experiments as to a lack of consensus between the tools on the number of vulnerabilities reported. However, in our work, we did not validate their findings and provide specific numbers on their accuracy in our dataset. The smart contracts are part of a random sample, meaning there is no ground truth to check against. In addition, the number of smart contracts is too large, meaning it would be time-consuming to find the tool's precision, even if this were part of the project's goal. Other works measure efficiency but only on a small number of smart contracts, which makes it a lot easier to find the ground truth – even manually.

7.2 Future Work

This project is incomplete. Despite our initial accomplishments, there is more we must do to elevate this promising study and framework to a higher degree of scrutiny. Undoubtedly, improvements must be made to ensure it is an effective and trustworthy framework. Below are some ideas on future progress.

A blueprint for the future is to expand our work on more EVM-compatible chains. Such popular chains include Avalanche and Polygon, with a TVL of $2.04B and $1.85B, respectively. As these chains are EVM based, the same smart contracts deployed on Ethereum and BSC could be deployed on them too. We aim to conduct a similar study to explore their vulnerability status, balances at stake and identical contracts deployed between multiple chains. Since we do not have access to other archive nodes on blockchains besides Ethereum and BSC, data and metadata will need to be collected from the chain's respective blockchain explorer.

An equally important aim for the future is to execute Centaur for a longer period. Instead of taking a random sample, we could instruct Centaur to crawl entire blockchains and produce a full-picture, comprehensive analysis. If this proves too ambitious and the time constraints do not permit its completion, we could set a specific period (e.g., one year), find the block numbers from each blockchain within that time frame and crawl them. If, in the future, we have more time to conduct our experiments. In that case, we seek to include more analysis tools in our study, especially those excluded because they time out within the 30-minute execution constraint applied. Moreover, we could compare the efficiency of different analysis tools that employ various techniques during their analysis.

As mentioned in the limitations section above, self-destructed contracts are handled like all other contracts, even though they don't have any value to us. At a later stage,

we would integrate an approach into Centaur that would detect self-destructed smart contracts during transaction crawling and recover their original bytecode before being removed from the blockchain. Additionally, we could incorporate a methodology for detecting proxy contracts as they are usually too short and simple (just delegates calls to other contracts) to contain any vulnerabilities. Removing any proxy contracts from our dataset will significantly save us time. Both the self-destructed and proxy approaches will assist in improving the quality of our final dataset.

Another idea for upgrading Centaur is to include a functionality for source code analysis. The chains' respective blockchain explorer can collect source code that developers submit to the platforms for verification and publication. This would make our framework more extensive and enable comparing analysis tools results on bytecodes and source codes to see if analysis tools perform differently on one or the other. We can also check how many bytecodes in the dataset have their source codes available and verify a correlation between available source codes and vulnerabilities. The reasoning behind this idea is that if the source code is available, people can verify it to reduce the chances of it containing vulnerabilities.

Finally, another plan that we believe will further enhance our project is to expand Table 4.1 to include proprietary analysis tools that are not publicly available for free use. More specifically, we aim to contact companies that specialise in the field of smart contract security to gain temporary access to any proprietary analysis tools or audit services to compare their findings with the open-source equivalents. Examples of such companies we have shortlisted include; Trail of bits[1], ConsenSys[2], Certora[3], Chainsecurity[4] and Beosin[5].

7.3 Threats to Validity

This section analyses any identified validity threats and discusses mitigation strategies. Threats to External, Internal, and Construct Validity is examined. In this work, we noticed that we share a proportion of threats with the SmartBugs paper [12], unsurprisingly, as our framework is inspired and uses a modified version of SmartBugs. For discussing threats to validity, we use the methodology introduced by Feldt *et al*. [107].

One potential threat to External Validity is the corpus of smart contracts we collected and analysed may not represent the vulnerability state of the respective chain we recovered them from. This is further exacerbated as the dataset size is minor compared to the entire Ethereum and BSC blockchains. To address this threat, we performed random sampling on all blocks of the blockchains at the time of writing to eliminate

[1]https://www.trailofbits.com/services/software-assurance
[2]https://consensys.net/diligence/
[3]https://www.certora.com/
[4]https://chainsecurity.com/audits/
[5]https://beosin.com/service/audit

any bias regarding the data quality. And to get a crystal-clear picture of the state of these blockchains, even though the sample size may not be as big as we wanted it to be. All smart contracts recovered are real-life and deployed on the blockchains; we only performed analysis on them if they had been utilised before. It is part of our upcoming work to increase the sample size. Another potential threat to External Validity is not including a tool that passes the requirements we set in Section 4.1 and may outperform the other nine tools we included. As a mitigation strategy for this threat, we searched all major conferences for papers that present tools, followed their references and crawled online open-source repositories for tools. If any other tool should be included on the list and it is not, then it was probably published after the time of writing.

One potential threat to Internal Validity is related to the implementation of Centaur and the execution of our experiments. Since our framework constitutes approximately 21K LOC[6], there may still be a bug in our codebase which could affect the final outcome. We thoroughly tested each component of the framework to mitigate this threat. Another issue associated with Internal Validity is the smart contracts we selected in our study. We had constraints in place for smart contracts to be included (i.e., no self-destructed contracts, have a balance bigger than zero, or at least on the transaction or token transfer) to ensure they are used in practice.

Furthermore, to ensure that any threats to Internal and External Validity are mitigated, we release our work as open source, including the Centaur's framework source code, the modified version of SmartBugs and all raw data collected. Public access ensures checking the validity of our conclusions.

A potential threat to Credibility Validity is linked to the labelling of reported vulnerabilities to the DASP10 and SWC taxonomies done manually. To lessen this issue, we first labelled the contracts to the vulnerability class that best matched the description and real-world examples. Then, to ensure we map every vulnerability to the correct category, we expanded the DASP10 taxonomy to include 18 classes instead of 10 to accommodate all potential vulnerabilities to the right group.

Summary: Due to time constraints, this project had limitations for a more comprehensive study. And therefore, there is fertile ground for further progress to advance the work already undertaken. The report indicates that analysis tools are inaccurate. More needs to be done to expand to other EVM-compatible chains. Centaur can crawl entire blockchains for a more accurate picture given the time. It can also be upgraded for source code analysis and access to non-open source tools in the future.

[6]excluding SmartBugs code, the database and any .sol, .hex, .json files

Chapter 8

Conclusion

Smart contract security has received greater attention from academia and the industry as the financial risks have grown more apparent. While new analysis tools are published frequently with different functions to uncover destructive vulnerabilities in the most prominent smart contract platform, Ethereum, other EVM-compatible platforms have fallen under the radar.

In this detailed work, we present Centaur, a publicly available framework for analysing vulnerabilities on EVM-based chains. Centaur collects smart contracts and stores their metadata. It uses an extended version of the SmartBugs framework to process and analyse the results. We deployed Centaur to conduct an empirical study on the vulnerability state of two EVM compatible chains, Ethereum and Binance Smart Chain.

We crawled 4,000 blocks from archive nodes on the Ethereum and Binance Smart Chain networks during our experimentation. We recovered over 17,592 smart contract bytecodes and executed nine state-of-the-art automated analysis tools on 473 unique bytecodes. The vulnerabilities discovered from this analysis were taxinomised based on an extended version of DASP10 and the SWC registry. Extended DASP10 was created, containing 18 vulnerability classes instead of 10. The experimentation phase took more than five days to complete.

Our findings discovered that, per 100 smart contracts, Ethereum contains 3.95 times the vulnerabilities BSC has, while the vulnerability groups that flourish the most on these two chains have specific differences. Moreover, many Ether and BNB are prone to malicious attackers due to vulnerabilities in smart contracts. Even though the total amount is only $3,725, we estimate the figure could be much larger considering the entire blockchain. Furthermore, a small fraction of contracts is believed to be deployed on both chains. In our case, only four contracts were deployed cross-chain taking advantage of the interoperability of the two chains.

We desire this project's findings to help guide future research in the complex field of smart contract security. Despite laser focus on Ethereum and Binance Smart Chain, we believe our framework and taxonomy remain generic and, therefore, applicable

to all other EVM-based blockchains.

8.1 Ethical Conduct

The highest ethical standards were observed in this study. Every step was taken to ensure the integrity and honesty of this project, with no inherent tendency to deceive. Our mission was to improve the new world of blockchain. Our work is dedicated to helping the community understand how the vulnerabilities in smart contracts work and avoid any costly mistakes in the future.

Professional transparency was our guiding light which is why the framework codebase is released as open source so others can replicate and advance our work on other EVM compatible chains and compare different vulnerabilities. We expect our exploration to assist fellow researchers in tool development to introduce new approaches to uncover vulnerabilities and stay protected from attackers. This scientific endeavour does not negatively affect the lives of others.

The essence of blockchain is its transparency, so we are providing a pioneering approach. Analysis tools are publicly available; although blockchain is transparent, it is anonymous. Anonymity is preserved, and there is nothing done to de-anonymise an account on either blockchain. Privacy is preserved and protected.

We were extremely careful not to take credit for something we did not own; there is a clear reference/recognition of other people's work (i.e., papers, open-source repositories) and technologies used. Our sole purpose was to mitigate vulnerabilities, improve the EVM-based blockchain ecosystem, and promote awareness about them in smart contracts and their consequences.

There was full adherence to the law, data protection, and rigorous observance of academic ideals and principles. There was no intention to mislead; on the contrary, it enhances our knowledge of blockchain technology and smart contract vulnerabilities. It was part of the mission statement to promote awareness and understanding without bias or prejudice. We are confident there is no legal ambiguity in the results we achieved. In our opinion, there is no wiggle room for abuse or criminal behaviour; on the contrary, this study benefits the public interest and improves our knowledge of fintech innovation. It highlights vulnerabilities in smart contracts by adding a cybersecurity firewall against malicious attacks or obtaining financial enrichment through illegal means.

Moreover, there is no conflict of interest with data protection or software copyright. Through this work, we hope to make people's money more secure as they are better protected from attacks in our pursuit to make blockchain a safer space. We believe we have improved the knowledge and security technology in the IT community, which as always, is to provide full transparency and the highest ethical professional standards.

The open-source repository of this project is licensed under the terms of the MIT license. This license applies to the whole codebase except for the SmartBugs framework and .hex and .sol files found in the data directory, which are publicly available and retain their original licenses.

Bibliography

[1] What is blockchain technology? - ibm blockchain. URL https://www.ibm.com/topics/what-is-blockchain. pages 1

[2] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL https://bitcoin.org/bitcoin.pdf. pages 1

[3] Cryptocurrency prices, charts, and crypto market cap. URL https://www.coingecko.com/. pages 1

[4] Defi pulse - the decentralized finance leaderboard: Stats, charts and guides. URL https://www.defipulse.com/. pages 1

[5] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, Jul 2022. URL https://ethereum.github.io/yellowpaper/paper.pdf. pages 1

[6] Cryptokitties craze slows down transactions on ethereum, Dec 2017. URL https://www.bbc.com/news/technology-42237162. pages 1

[7] Thedao smart contract code, . URL https://etherscan.io/address/0x71c7656ec7ab88b098defb751b7401b5f6d8976f. pages 2, 10

[8] Phil Daian. Analysis of the dao exploit. URL https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/. pages 2, 10

[9] Ryan Browne. 'accidental' bug may have frozen $280 million worth of digital coin ether in a cryptocurrency wallet, Nov 2017. URL https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html. pages 2, 11

[10] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 254–269, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978309. URL https://doi.org/10.1145/2976749.2978309. pages 2, 24, 25

[11] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, page 91–96, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450345743. doi: 10.1145/2993600.2993611. URL https://doi.org/10.1145/2993600.2993611. pages 2

[12] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, jun 2020. doi: 10.1145/3377811.3380364. URL https://doi.org/10.1145%2F3377811.3380364. pages 2, 3, 17, 18, 19, 20, 26, 28, 53, 56, 57

[13] Bnb-Chain. Bnb-chain/whitepaper, Jun 2020. URL https://github.com/bnb-chain/whitepaper/blob/master/WHITEPAPER.md. pages 2

[14] Explore the bnb smart chain ecosystem and binance dex. fast, decentralized, affordable and secure. URL https://www.binance.org/en. pages 2

[15] Binance Academy. An introduction to binance smart chain (bsc), Aug 2022. URL https://academy.binance.com/en/articles/an-introduction-to-binance-smart-chain-bsc. pages 2, 6

[16] Avalanche: Blazingly fast, low cost, and eco-friendly. URL https://www.avax.network/. pages 3

[17] Decentralized application security project. URL https://dasp.co/. pages 3, 9, 17

[18] Smart contract weakness classification and test cases. URL https://swcregistry.io/. pages 3, 9, 17

[19] Adam Hayes. Blockchain explained, Jun 2022. URL https://www.investopedia.com/terms/b/blockchain.asp. pages 6

[20] Consensus mechanisms. URL https://ethereum.org/en/developers/docs/consensus-mechanisms/. pages 6

[21] Sahil Sen. Ethereum full node vs archive node, Apr 2022. URL https://www.quicknode.com/guides/infrastructure/ethereum-full-node-vs-archive-node. pages 7

[22] Networks, Jul 2022. URL https://ethereum.org/en/developers/docs/networks/. pages 7

[23] Ethereum. Ethereum/go-ethereum: Official go implementation of the ethereum protocol, . URL https://github.com/ethereum/go-ethereum. pages 7

[24] Go ethereum. URL https://geth.ethereum.org/. pages 7

[25] Bnb-Chain. Bnb-chain/bsc: A bnb smart chain client based on the go-ethereum fork. URL https://github.com/bnb-chain/bsc. pages 7

[26] Nick Szabo. The idea of smart contracts. URL https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_idea.html. pages 8

[27] Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson, editors. *Financial Cryptography and Data Security*. Springer International Publishing, 2017. doi: 10.1007/978-3-319-70278-0. URL https://doi.org/10.1007%2F978-3-319-70278-0. pages 8

[28] Real world examples of smart contracts. URL https://www.gemini.com/cryptopedia/smart-contract-examples-smart-contract-use-cases. pages 8

[29] GAVIN WOOD. Polkadot: Vision for a heterogeneous multi-chain framework. URL https://polkadot.network/PolkaDotPaper.pdf. pages 8

[30] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. URL https://lightning.network/lightning-network-paper.pdf. pages 8

[31] What are token standards? an overview. URL https://crypto.com/university/what-are-token-standards. pages 8

[32] Dominik Harz and William Knottenbelt. Towards safer smart contracts: A survey of languages and verification methods, Nov 2018. URL https://arxiv.org/abs/1809.09805. pages 9, 20

[33] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. 2017. doi: 10.48550/ARXIV.1712.06438. URL https://arxiv.org/abs/1712.06438. pages 9

[34] Cve-2018-10299 detail. URL https://nvd.nist.gov/vuln/detail/CVE-2018-10299. pages 10

[35] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale, 2018. URL https://arxiv.org/abs/1802.06038. pages 13, 24, 25

[36] King of the ether. URL https://www.kingoftheether.com/thrones/kingoftheether/index.html. pages 14

[37] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Ethereum smart contract analysis tools: A systematic review. *IEEE Access*, 10:57037–57062, 2022. doi: 10.1109/ACCESS.2022.3169902. pages 15, 23

[38] Anastasia Mavridou and Aron Laszka. Tool demonstration: Fsolidm for designing secure ethereum smart contracts, 2018. URL https://arxiv.org/abs/1802.09949. pages 15, 24

[39] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189, 2019. doi: 10.1109/ASE.2019.00133. pages 15, 24

[40] Ethereum contract creation - explained from bytecode. URL https://monokh.com/posts/ethereum-contract-creation-bytecode. pages 15

[41] Monika di Angelo and Gernot Salzer. A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, pages 69–78, 2019. doi: 10.1109/DAPPCON.2019.00018. pages 15, 17, 20, 23

[42] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg. ISBN 978-3-662-54455-6. pages 15, 17

[43] 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. ISSN 0360-0300. doi: 10.1145/3391195. URL https://doi.org/10.1145/3391195. pages 17

[44] ConsenSys. Consensys/smart-contract-best-practices: A guide to smart contract security best practices, . URL https://github.com/ConsenSys/smart-contract-best-practices. pages 17

[45] Dr Adrian Manning. Solidity security: Comprehensive list of known attack vectors and common anti-patterns, Oct 2018. URL https://blog.sigmaprime.io/solidity-security.html. pages 18

[46] Solhydra. URL https://www.npmjs.com/package/solhydra. pages 18

[47] Asem Ghaleb and Karthik Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium*

on Software Testing and Analysis. ACM, jul 2020. doi: 10.1145/3395363.3397385. URL https://doi.org/10.1145%2F3395363.3397385. pages 18, 19, 20

[48] Haoran Wu, Xingya Wang, Jiehui Xu, Weiqin Zou, Lingming Zhang, and Zhenyu Chen. Mutation testing for ethereum smart contract, 2019. URL https://arxiv.org/abs/1908.03707. pages 18

[49] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, page 259–269, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. URL https://doi.org/10.1145/3238147.3238177. pages 18, 24

[50] Gustavo Grieco, Will Song, and Artur Cygan. Echidna: Effective, usable, and fast fuzzing for smart contracts. URL https://agroce.github.io/issta20.pdf. pages 18

[51] OpenZeppelin. Openzeppelin/ethernaut: Web3/solidity based wargame. URL https://github.com/OpenZeppelin/ethernaut. pages 18

[52] Crytic. Crytic/not-so-smart-contracts: Examples of solidity security issues, . URL https://github.com/crytic/not-so-smart-contracts. pages 19

[53] Soohoio. Soohoio/verismartbench: Benchmarks for solidity smart contracts. URL https://github.com/soohoio/VeriSmartBench. pages 19

[54] ConsenSys. Consensys/evm-analyzer-benchmark-suite: A benchmark suite for evaluating the precision of evm code analysis tools., . URL https://github.com/ConsenSys/evm-analyzer-benchmark-suite. pages 19

[55] Hrishioa. Hrishioa/smart-contract-benchmark. URL https://github.com/hrishioa/smart-contract-benchmark. pages 19

[56] Smartbugs. Smartbugs/smartbugs: Smartbugs: A framework to analyze solidity smart contracts, . URL https://github.com/smartbugs/smartbugs/tree/master. pages 19

[57] Smartbugs. Smartbugs/smartbugs-wild: This repository contains 47,398 smart contracts extracted from the ethereum network, . URL https://github.com/smartbugs/smartbugs-wild. pages 19

[58] Ethereum (eth) blockchain explorer, . URL https://etherscan.io/. pages 19

[59] DependableSystemsLab. Dependablesystemslab/solidifi: Solidifi is an automated and systematic framework for evaluating smart contracts' static analysis tools via fault injection. URL https://github.com/DependableSystemsLab/SolidiFI. pages 19

[60] Reza M. Parizi, Ali Dehghantanha, Kim-Kwang Raymond Choo, and Amritraj Singh. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, page 103–113, USA, 2018. IBM Corp. pages 20, 56

[61] Andrea Pinna, Simona Ibba, Gavina Baralla, Roberto Tonelli, and Michele Marchesi. A massive analysis of ethereum smart contracts empirical study and code metrics. *IEEE Access*, 7:78194–78213, 2019. doi: 10.1109/ACCESS.2019.2921936. pages 20

[62] Bin Hu, Zongyang Zhang, Jianwei Liu, Yizhong Liu, Jiayuan Yin, Rongxing Lu, and Xiaodong Lin. A comprehensive survey on smart contract construction and execution: Paradigms, tools, and systems, Feb 2021. pages 20

[63] Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. Ethereum smart contract analysis tools: A systematic review. *IEEE Access*, 10:57037–57062, 2022. doi: 10.1109/ACCESS.2022.3169902. pages 20

[64] Nuno Veloso. Conkas: A modular and static analysis tool for ethereum bytecode. URL `https://fenix.tecnico.ulisboa.pt/downloadFile/1689244997262417/94080-Nuno-Veloso_resumo.pdf`. pages 24, 25

[65] Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. Monitoring smart contracts: Contractlarva and open challenges beyond. In Christian Colombo and Martin Leucker, editors, *Runtime Verification*, pages 113–137, Cham, 2018. Springer International Publishing. ISBN 978-3-030-03769-7. pages 24

[66] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, and Andrea Cullen. Visual emulation for ethereum's virtual machine. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–4, 2018. doi: 10.1109/NOMS.2018.8406332. pages 24

[67] Crytic. Crytic/echidna: Ethereum smart contract fuzzer, . URL `https://github.com/crytic/echidna`. pages 24

[68] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. Erays: Reverse engineering ethereum's opaque smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1371–1385, Baltimore, MD, August 2018. USENIX Association. ISBN 978-1-939133-04-5. URL `https://www.usenix.org/conference/usenixsecurity18/presentation/zhou`. pages 24

[69] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 454–469, New York, NY, USA, 2020. Association for Computing Machinery. ISBN

9781450376136. doi: 10.1145/3385412.3385990. URL https://doi.org/10.1145/3385412.3385990. pages 24

[70] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2757–2774. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/frank. pages 24

[71] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun. S-gram: Towards semantic-aware security auditing for ethereum smart contracts. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 814–819, Los Alamitos, CA, USA, sep 2018. IEEE Computer Society. doi: 10.1145/3238147.3240728. URL https://doi.ieeecomputersociety.org/10.1145/3238147.3240728. pages 24

[72] SeUniVr. Seunivr/ethersolve: Source code of ethersolve: Static analysis of ethereum bytecode. URL https://github.com/SeUniVr/EtherSolve. pages 24

[73] Crytic. Crytic/ethersplay: Evm dissassembler, . URL https://github.com/crytic/ethersplay. pages 24

[74] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Ethertrust: Sound static analysis of ethereum bytecode. 2018. pages 24

[75] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. *EthIR: A Framework for High-Level Analysis of Ethereum Bytecode: 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, pages 513–520. 09 2018. ISBN 978-3-030-01089-8. doi: 10.1007/978-3-030-01090-4_30. pages 24

[76] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts, 2020. URL https://arxiv.org/abs/2005.06227. pages 24

[77] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446, 2017. doi: 10.1109/SANER.2017.7884650. pages 24

[78] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: Thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186, 2019. doi: 10.1109/ICSE.2019.00120. pages 24

[79] Christof Ferreira Torres, Mathis Steichen, and Radu State. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1591–1607,

Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira. pages 24, 25

[80] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018. doi: 10.1109/CSF.2018.00022. pages 24

[81] Palkeo. Palkeo/pakala: Offensive vulnerability scanner for ethereum, and symbolic execution tool for the ethereum virtual machine. URL https://github.com/palkeo/pakala. pages 24

[82] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018. doi: 10.1145/3276486. URL https://doi.org/10.1145/3276486. pages 24, 25

[83] ConsenSys. Consensys/mythril: Security analysis tool for evm bytecode. supports smart contracts built for ethereum, hedera, quorum, vechain, roostock, tron and other evm-compatible blockchains., . URL https://github.com/ConsenSys/mythril. pages 24, 25

[84] FuzzingLabs. Fuzzinglabs/octopus: Security analysis tool for webassembly module (wasm) and blockchain smart contracts (btc/eth/neo/eos). URL https://github.com/FuzzingLabs/octopus. pages 24

[85] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 664–676, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365697. doi: 10.1145/3274694.3274737. URL https://doi.org/10.1145/3274694.3274737. pages 24, 25

[86] Msuiche. Msuiche/porosity: Decompiler and security analysis tool for blockchain-based ethereum smart-contracts. URL https://github.com/msuiche/porosity. pages 24

[87] Crytic. Crytic/rattle: Evm binary static analysis, . URL https://github.com/crytic/rattle. pages 24

[88] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: Finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68, 2018. pages 24

[89] Ethereum. Ethereum/remix-project: Remix is a browser-based compiler and ide that enables users to build ethereum contracts with solidity language and to debug transactions., . URL `https://github.com/ethereum/remix-project`. pages 24

[90] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. Security assurance for smart contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2018. doi: 10.1109/NTMS.2018.8328743. pages 24

[91] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. *sCompile: Critical Path Identification and Analysis for Smart Contracts*, pages 286–304. 10 2019. ISBN 978-3-030-32408-7. doi: 10.1007/978-3-030-32409-4_18. pages 24

[92] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 67–82, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243780. URL `https://doi.org/10.1145/3243734.3243780`. pages 24, 25

[93] 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. doi: 10.1109/WETSEB.2019.00008. pages 24

[94] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16, 2018. pages 24

[95] Raineorshine. Raineorshine/solgraph: Visualize solidity control flow for smart contract security analysis. URL `https://github.com/raineorshine/solgraph`. pages 24

[96] Protofire. Protofire/solhint: Solhint is an open source project created by https://protofire.io. its goal is to provide a linting utility for solidity code. URL `https://github.com/protofire/solhint`. pages 24

[97] Péter Hegedus. Towards analyzing the complexity landscape of solidity based ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 35–39, 2018. pages 24

[98] 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. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/krupp. pages 24

[99] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts, 2018. URL https://arxiv.org/abs/1809.03981. pages 24, 25

[100] Joran J. Honig, Maarten H. Everts, and Marieke Huisman. Practical mutation testing for smart contracts. In Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquin Garcia-Alfaro, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 289–303, Cham, 2019. Springer International Publishing. ISBN 978-3-030-31500-9. pages 24

[101] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. Formal specification and verification of smart contracts for azure blockchain, 2018. URL https://arxiv.org/abs/1812.08829. pages 24

[102] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *NDSS*, 2018. pages 24

[103] George Pîrlea, Amrit Kumar, and Ilya Sergey. *Practical Smart Contract Sharding with Ownership and Commutativity Analysis*, page 1327–1341. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450383912. URL https://doi.org/10.1145/3453483.3454112. pages 33

[104] Evm transactions. URL https://ethereum.org/en/developers/docs/transactions/. pages 35

[105] Lorenz Breidenbach and Phil Daian. Gastoken.iohttps://gastoken.io/. URL https://gastoken.io/. pages 44

[106] Jiachi Chen, Xin Xia, David Lo, and John Grundy. Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum, 2020. URL https://arxiv.org/abs/2005.07908. pages 55

[107] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research - an initial survey. In *SEKE*, 2010. pages 57

Appendix A

Database

```yaml
version: '3.3'
services:
  mariadb:
    container_name: db_blockchain
    image: mariadb:10.6.4
    volumes:
      - ./db_blockchain:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD_FILE: /run/secrets/db_root_password
      MYSQL_DATABASE: db_blockchain
      MYSQL_USER: main
      MYSQL_PASSWORD_FILE: /run/secrets/db_password
    ports:
      - "3333:3306"
    networks:
      blockchain_net:
        ipv4_address: 192.168.100.20
    restart: always
    secrets:
      - db_root_password
      - db_password
secrets:
   db_password:
     file: db_password.txt
   db_root_password:
     file: db_root_password.txt
networks:
  blockchain_net:
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 192.168.100.0/24
```

Figure A.1: docker-compose YAML file for deploying MariaDB.

Appendix B

Vulnerabilities Reported

DASP10 (SWC-ID)	Conkas	HoneyBadger	MadMax	Maian	Mythril	Osiris	Oyente	Securify	Vandal	Total	Total w/ duplicates
1.Reentrancy (SWC-107)	56 | 1264	-	-	-	34 | 433	7 | 406	8 | 302	0 | 0	146 | 1220	**251**	**3625**
2.Access Control (SWC-105, 106, 115)	-	-	-	3 | 62	8 | 75	-	-	0 | 0	11 | 13	**22**	**150**
3.Arithmetic Issues (SWC-101)	68| 1370	-	1 | 1	-	20 | 23	95 | 123	-	-	-	**184**	**1517**
4.Unchecked Return Values For Low Level Calls (SWC-104)	5 | 66	-	-	-	2 | 9	-	-	-	142 | 1320	**149**	**1395**
5.Denial of Service (SWC-115)	-	-	2 | 2	-	12 | 404	-	-	0 | 0	-	**14**	**406**
6.Bad Randomness (SWC-120)	-	-	-	-	16 | 21	-	-	-	-	**16**	**21**
7.Front-Running (SWC-114)	1 | 1	-	-	-	0 | 0	10 | 537	21 | 446	0 | 0	-	**32**	**984**
8.Time manipulation (SWC-116)	14 | 27	-	-	-	0 | 0	0 | 0	2 | 2	-	-	**16**	**29**
9.Short Address Attack (N.A)	-	-	-	-	-	-	-	-	-	**0**	**0**
10.Honeypots (N.A)	-	0 | 0	-	-	-	-	-	-	-	**0**	**0**
11.Non-Isolated External Calls (Wallet Griefing) (SWC-126)	-	-	0 | 0	-	-	-	-	-	-	**0**	**0**
12.Greedy (N.A)	-	-	-	12 | 37	-	-	-	-	-	**12**	**37**
13.Assert Violations (SWC-110)	-	-	-	-	49 | 664	-	-	-	-	**49**	**664**
14.Delegate Call To Untrusted Contract (SWC-112)	-	-	-	-	0 | 0	-	-	-	-	**0**	**0**
15.Write to Arbitrary Storage Location (SWC-124)	-	-	-	-	0 | 0	-	-	-	-	**0**	**0**
16.Hardcoded Gas (SWC-134)	-	-	-	-	0 | 0	-	-	0 | 0	-	**0**	**0**
17.Jump to an arbitrary instruction (SWC-127)	-	-	-	-	1 | 4	-	-	-	-	**1**	**4**
18.Callstack Depth Attack Vulnerability (N.A)	-	-	-	-	-	-	18 | 25	-	-	**18**	**25**
Total	**144**	**0**	**3**	**15**	**142**	**112**	**49**	**0**	**299**	**764**	
Total w/ duplicates	**2728**	**0**	**3**	**99**	**1633**	**1066**	**775**	**0**	**2553**		**8857**

Figure B.1: Ethereum-deployed smart contract vulnerabilities reported.

DASP10 (SWC-ID)	Conkas	HoneyBadger	MadMax	Maian	Mythril	Osiris	Oyente	Securify	Vandal	Total	Total w/ duplicates
1.Reentrancy (SWC-107)	30 | 31	-	-	-	12 | 12	0 | 0	0 | 0	0 | 0	99 | 147	**141**	**190**
2.Access Control (SWC-105, 106, 115)	-	-	-	2 | 255	4 | 258	-	-	0 | 0	6 | 260	**12**	**773**
3.Arithmetic Issues (SWC-101)	55 | 89	-	0 | 0	-	42 | 43	42 | 96	-	-	-	**139**	**228**
4.Unchecked Return Values For Low Level Calls (SWC-104)	2 | 2	-	-	-	1 | 1	-	-	-	116 | 171	**119**	**174**
5.Denial of Service (SWC-115)	-	-	1 | 1	-	8 | 8	-	0 | 0	0 | 0	-	**9**	**9**
6.Bad Randomness (SWC-120)	-	-	-	-	28 | 82	-	-	-	-	**28**	**82**
7.Front-Running (SWC-114)	2 | 2	-	-	-	0 | 0	1 | 2	0 | 0	0 | 0	-	**3**	**4**
8.Time manipulation (SWC-116)	23 | 77	-	-	-	0 | 0	0 | 0	0 | 0	-	-	**23**	**77**
9.Short Address Attack (N.A)	-	-	-	-	-	-	-	-	-	**0**	**0**
10.Honeypots (N.A)	-	0 | 0	-	-	-	-	-	-	-	**0**	**0**
11.Non-Isolated External Calls (SWC-126)	-	-	0 | 0	-	-	-	-	-	-	**0**	**0**
12.Greedy (N.A)	-	-	-	65 | 70	-	-	-	-	-	**65**	**70**
13.Assert Violations (SWC-110)	-	-	-	-	18 | 18	-	-	-	-	**18**	**18**
14.Delegate Call To Untrusted Contract (SWC-112)	-	-	-	-	1 | 1	-	-	-	-	**1**	**1**
15.Write to Arbitrary Storage Location (SWC-124)	-	-	-	-	2 | 2	-	-	-	-	**2**	**2**
16.Hardcoded Gas (SWC-134)	-	-	-	-	0 | 0	-	-	0 | 0	-	**0**	**0**
17.Jump to an arbitrary instruction (SWC-127)	-	-	-	-	1 | 1	-	-	-	-	**1**	**1**
18.Callstack Depth Attack Vulnerability (N.A)	-	-	-	-	-	-	32 | 84	-	-	**32**	**84**
Total	**112**	**0**	**1**	**67**	**117**	**43**	**32**	**0**	**221**	**593**	
Total w/ duplicates	**201**	**0**	**1**	**325**	**426**	**98**	**84**	**0**	**578**		**1713**

Figure B.2: BSC-deployed smart contract vulnerabilities reported.

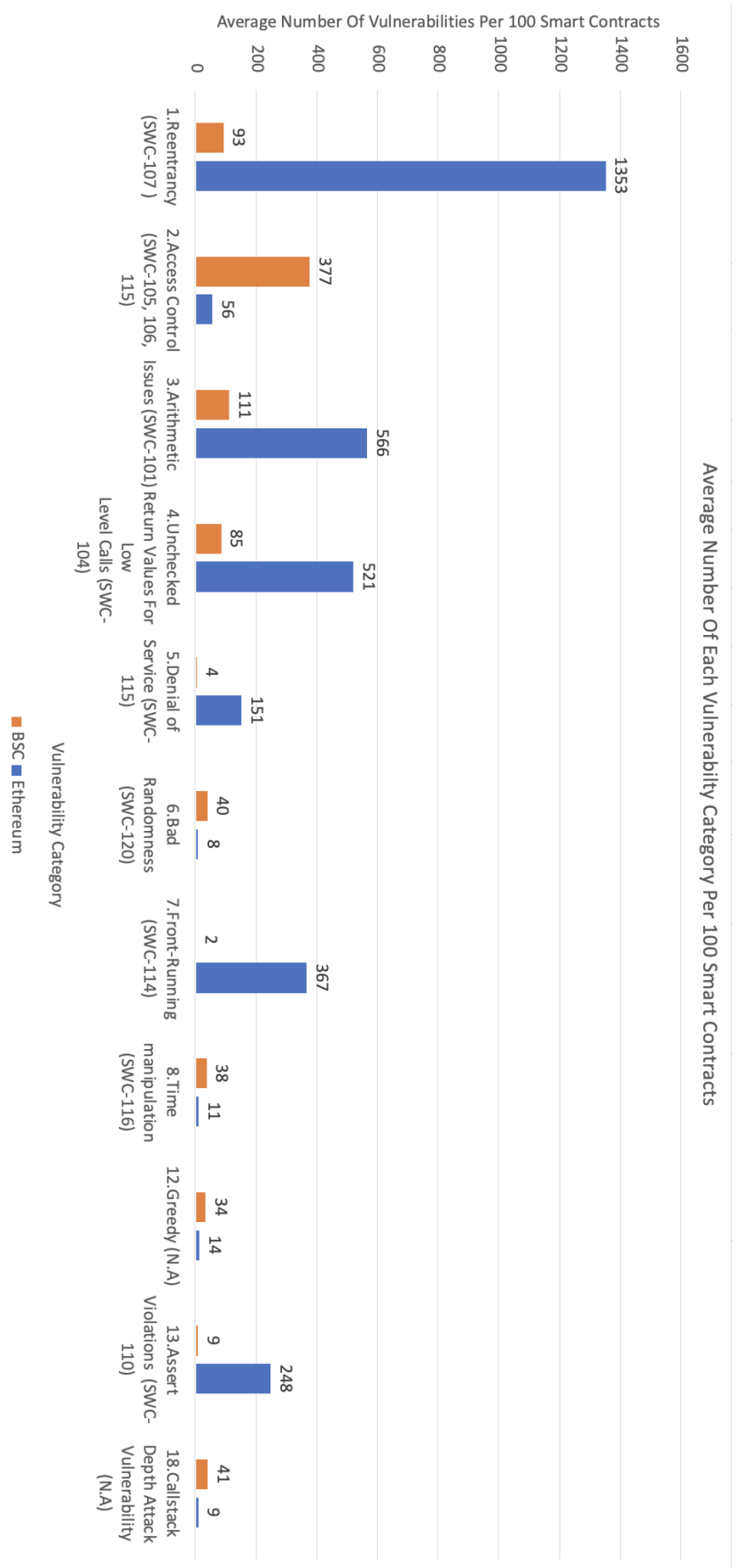


Figure B.3: Vulnerabilities per 100 contracts for both chains.