University of Piraeus
School of Information and Communication Technologies
Department of Digital Systems

Postgraduate Program of Studies
MSc Digital Systems Security

An automated assessment of smart contract vulnerabilities in the Ethereum blockchain using open-source tools

Supervisor Professor: Christos Xenakis

Name-Surname	E-mail	Student ID.
Vasilis Magkoutis	bmagoutis@gmail.com	Vasilis Magkoutis

Piraeus
15/01/2023

Abstract

Blockchain technology is gradually taking place in the technology spectrum and the academic community. From the various uses of the blockchain, one of the most successful and interesting ones is the Ethereum Virtual Machine. In this case, blockchain can enable smart contracts (in essence, programming code) to run on the network autonomously. This particular technology arrived fast, and multiple developers are already coding these smart contracts. However, various vulnerabilities in their code and developer practices surfaced with the broader acceptance of smart contracts. Gradually tools to identify these vulnerabilities became available, and the community focused more on the security of the smart contracts. But what about the result? How many vulnerable contracts exist on the network, are there contracts with different vulnerabilities, and are the vulnerable contracts deployed on the network decreasing? The above constitute some of the questions we will try to answer.

Acknowledgments

I want to thank my supervisor Prof. Xenakis Ch., for his consistent support and guidance during the running of this thesis. Furthermore, I thank Bolgouras V., a Ph.D. student at the University of Piraeus, for his research insights and positive feedback. Finally, I would like to thank my wife Margarita, who supported and understood my effort to finish this thesis despite being pregnant with our daughter.

Table of Content

Introduction

Constant breakthroughs in science and technology dominate modern history. One of the most important turning points in the development of technology was the invention of the computer by Alan Turing. Following it was the invention of the World Wide Web by Tim Berners-Lee, which signaled the dawn of the information era. Information technology is advancing at an increasing pace to meet society's needs. One of the latest technological breakthroughs was the creation of bitcoin in 2008 by Satoshi Nakamoto [1], thus leading to the invention of blockchain technology. Blockchain technology is used in bitcoin to solve the ever-present problem of transferring value between two individuals or entities. The invention of the blockchain made possible concepts and applications that nobody had imagined before or were impossible to achieve with the existing technologies. One such application that was made possible was the Smart Contracts, which was introduced by Vitalik Buterin in 2014 [3]. According to Vitalik, smart contracts are "systems which automatically move digital assets according to arbitrary pre-specified rules."

Blockchain

Blockchain technology is the foundation of bitcoin and other cryptocurrencies. Blockchain can be considered a public ledger that stores all the transactions on the network in a list of blocks. This public ledger grows with each new block of transactions added [4]. For securing this ledger, asymmetric cryptography gets used coupled with decentralized consensus algorithms. Some basic traits of blockchain are persistence, decentralization, audibility, and anonymity.

Blockchain allows transactions to be processed and finished without any third party, thus making it ideal to be used in various financial applications like digital assets, online payments, remittances, and more [4]. Blockchain also gets used in many other applications, such as smart contracts, public services, reputation systems, and the internet of things. Blockchain as a technology is immutable, implying that transactions cannot get altered once they are finalized. Thus, blockchain can provide reliability and integrity that can be especially attractive to certain business sectors. One of the most

important blockchain applications is smart contracts, which can be automatically executed by the miners when deployed on the network [3][4].

One of the biggest challenges that blockchain technology faces are the scalability problem. In bitcoin, for example, the size of a block is 1 MB, and a block gets mined every 10 minutes; thus, bitcoin provides a throughput of roughly seven transactions per second. However, this throughput cannot sustain high-frequency trading and other applications. Various technologies have emerged to solve this issue by providing larger space per block and faster block times, but this can lead to a vast space required to store the blockchain data, thus impacting decentralization as fewer users will be able to hold it [5].

Blockchain Architecture

As stated previously, a blockchain is a list of blocks that register transactions and is considered a public ledger. Each new block in the ledger has a unique hash. In its header, the block includes the hash of the previous block. One block can have only one previous block. In the ledger, the first block is called the genesis block and has no previous (parent) block [7].

Blocks consist of two parts the header and the body. Block header includes various information such as [4][7]:

- Block version - used to identify the specific set of rules that users must use to validate the block
- Merkle tree root hash – this is a hash of the sum of the transactions that are included in this block
- Timestamp – Usually the Unix timestamp which is the number of seconds elapsed since January 1st of 1970
- nBits – max size for a valid block hash
- Nonce – this is a 4-byte value used in the hash calculation, and it increases with each hash
- Previous or parent block – this is the 256bit hash of the previous block

A block counter and the transactions are included in the block body part. The number of transactions in each block can vary based on the block size and the size of each transaction [6].

Digital signatures based on asymmetric cryptography are used in the blockchain due to the untrusted operational environment. As a result, each user uses a pair of private and public keys. The private key is confidential and should be stored securely and safely [4]. Users use this private to sign transactions propagated throughout the network. This process is separated into two phases. In the first phase, the user builds, sings, and submits a transaction to the network. In the next phase, all the other participants in the network can validate that this user can initiate this transaction by verifying the digital signature on the transaction. Finally, the user that has successfully solved the mining problem can add a new block to the blockchain that will include this transaction [4].

Categories of blockchains

Blockchains are commonly separated into three categories: public, private, and consortium blockchains [4][8]. The blocks, transactions, pending transactions, and other records in a public blockchain should be made public and accessible to every interested party. Also, anyone can be part of the consensus mechanism. In contrast to the public blockchain, in consortium blockchain, everything is public, but in the consensus mechanism, only a few pre-select nodes participate. Lastly, private blockchains are designed to be used in organizations, and only nodes specified by the organization participate. Also, the visibility of all records is subject to control and restrictions based on the organization's needs [8].

Bitcoin

The modern digital money ecosystem uses various underline technologies and concepts. Bitcoin is a collection of such technologies and concepts[2]. The users can communicate with one another by using the bitcoin protocol. This communication occurs mostly through the internet, but other networks can substitute it. The big advantages of bitcoin technology are that it is easily accessible, available on multiple devices, has various implementations, and is open source. Bitcoin leverages digital signatures to create a digital currency that users can use like normal currency. They can buy and sell products or send money to other users. Because bitcoin is fast, secure, and widely available, it can be considered the ideal form of money for the digital online world [2].

Bitcoin has no physical form and is an intangible asset in contrast to a normal currency that has a physical form and is a tangible asset. The resulting bitcoins in the bitcoin network result from transactions that transfer value from one user to another or, more

precisely, one address to another. The user is in the custody of a key that proves ownership of an address in the bitcoin network. By owning the address, the user also owns the transactions to and from this address. Thus the resulting value from these transactions [2].

One of the biggest features of bitcoin is that it is a peer-to-peer system and is fully decentralized, thus removing the need for a centralized point of control or failure. The above feature enables the protocol users to transact with another without needing a third party [1][2]. New bitcoins get created with a process that is called mining. Every user participating in the network can mine new bitcoins. In this process, users can use their computer's processing power to solve a difficult mathematical problem. After solving the problem, that user can validate the pending transactions in the network, which awards the new bitcoins. The mining operation imitates the currency-issuance behavior of central banks in normal currencies [2].

The mining algorithm that gets solved by the users (miners) exits in the bitcoin network, and its difficulty is adjusted by the network so that no matter how much computing power gets used, the approximate needed time to solve the problem will be 10 minutes. The bitcoin network also has preset functionality that halves the mining rewards every four years. As a result, the maximum number of bitcoins that will ever be available is 21 million coins [1] [2]. The above mechanism means that bitcoin is a deflationary asset, meaning that no extra bitcoins will be created (printed) other than the expected issuance rate [2].

Consensus Algorithms

One of the biggest problems in the blockchain is reaching a consensus in a network of untrusted nodes. This problem is similar to the Byzantine Generals (BG) problem [9]. In this problem, a group of generals that commands part of the army meets a city. Then some generals prefer to attack, while others retreat. The problem is that to succeed in the attack, all the generals must attack together. This problem illustrates the difficulty of reaching a consensus in a decentralized untrusted environment. Because of the decentralized nature of the blockchain, there is no central node, but the ledger in the various nodes should be consistent. Various consensus protocols ensure consistency. Proof of work (PoW) and Proof of Stake (PoS) are the most used algorithms in cryptocurrency to achieve node consensus.

Proof of Work (PoW)

Proof of work is the consensus algorithm used by the bitcoin network [1]. In this algorithm, for one node to append a new block on the blockchain that will hold the new transactions submitted to the network, the node will have to calculate the hash of the packet header. Nevertheless, that is not enough by itself. The node will need to change a nonce value in the header and recalculate the hash until it meets a certain criterion of size. More specifically, the hash size will need to be smaller than a certain value, then the node propagates the package through the network, and after the other nodes validate it, they add it to their blockchain [1].

Proof of Stake (PoS)

Proof of stake is trying to fix one of the biggest problems of the proof of work algorithm, which is its massive energy consumption. In PoS algorithms, the users that want to add a new block to the chain need to prove that they already own several coins. This algorithm is based on the assumption that users with more coins are less likely to attack the network [4]. However, this comes with a big disadvantage: the rich users get richer; thus, various PoS implementations try to limit the staking amount or randomly determine the node to add the new block. Because PoS energy consumption is almost zero compared to the PoW algorithm, many networks started with PoW and then transitioned to PoS. Such an example is the Ethereum network which started by using Ethash [10], a PoW consensus algorithm, and is planning to move to a PoS algorithm named Casper [11].

Smart Contracts

Nick Szabo introduced the smart contracts concept in 1996 as "computerized transaction protocols that execute terms of a contract" [12]. Along with technological progress and especially the invention of blockchain technology, this concept has evolved to include more variations and implementations. As a result, smart contracts, together with blockchain technology, have, in recent years, been a major focus of the academic community. Through this focus, the mentioned technologies have been applied to multiple fields [13].

Smart contracts are a set of promises agreed upon between parties to formalize a relationship, as mentioned in [13]. Smart contracts can be thought of as the building blocks of blockchain applications. Their usage is to facilitate communication in the network, like the other blockchain rules. Smart contracts have rules and conditions

written in a programming language, and blockchain enables the automatic execution of these rules and conditions. Upon meeting the conditions in a smart contract, the blockchain automatically executes the rules preset by the smart contract [13]. Essentially smart contracts are transactions stored in blockchain; thus, by leveraging blockchain technology, no third party is needed to enforce their rules and conditions, in contrast with traditional contracts that need a third party to guarantee their fulfillment. Thus, smart contracts can be envisioned as the natural evolution of the blockchain as a transaction protocol to the blockchain as a multi-purpose tool. Smart contracts are not contracts in a legal sense but are parts of software that can enhance blockchain functionality [13].

Advantages of Smart Contracts

According to [13], the main advantages of smart contracts are:

- Reducing risks: Through the public nature, immutability, traceability, audibility, and integrity offered by the blockchain, malicious actions such as fraud are almost eliminated.
- Cutting down administration and service costs – Due to blockchain not needing a third party to enforce a contract as it will happen automatically, the administrative fees required for the meditation of a third party are thus eliminated.
- Improving the efficiency of business processes – As mentioned above, removing the need for a third party can also speed up the financial settlement process as it will take place in a peer-to-peer fashion.
- Speed and real-time updates – Smart contracts use rules programmed in a programming language and automatically executed by a computer participating in the network, thus significantly speeding up processes that otherwise would be manually done.
- Accuracy – As mentioned above, the programmed rules speed up the process and help remove manual errors.
- Lower execution risk – This happens because the execution takes place automatically in the network. So, risks like manipulation, nonperformance, and errors are greatly reduced.

- Fewer intermediaries – Smart contracts remove the need to rely on third parties to provide trust services.
- Lower cost – As stated previously, smart contracts remove the need for third parties and significantly reduce the human factor, thus effectively reducing the cost.
- New business or operational models – This is an inevitable outcome as the innovations provided by smart contracts can enable various business sectors, such as peer-to-peer renewable energy trading, automated access, vehicles, storage facilities, and more.

Smart Contracts and Data Management

One of online security and data privacy's biggest concerns is data management. Currently used cloud-based solutions, the data are stored and analyzed on centralized servers. Thus, concerns have arisen regarding trust in cloud provider security, loss of control once data are put to an external place, and lack of transparency in the handling processes [14]. However, with their peer-to-peer ability to securely transfer data between untrusted parties, smart contracts and blockchain technology can mitigate or eliminate such concerns. Furthermore, due to blockchain's immutable nature, it can provide data provenance.

Furthermore, data access can also be securely implemented due to all parties' consensus. Finally, data availability can also be extremely high due to the peer-to-peer nature. As a result, traditional DoS (Denial of Services) attacks are often harder to pull off against distributed systems than centralized ones [14].

Smart Contracts and Auditing

Another aspect of security that is a concern in smart contracts is a security auditing and risk assessment. As organizations start to use smart contracts, there should also be a process for external auditors to examine the contract's code (terms) and make an objective conclusion about the contract's compliance with the guidelines [15]. Unfortunately, such a process is not currently available because about 77% of smart contracts do not have their source code available, and only the byte code is available. Due to these circumstances, many solutions have been proposed, such as reverse engineering tools, semi-automated translation systems, and programming languages that do not need compilation or the definition of programming languages that make the execution code human-readable [15].

Another problem derived from the organization's use of smart contracts is the new risk that comes with it and the need for both internal and external auditors to be able to monitor these. Such risks can be [15]:

- Data integrity may not be guaranteed in the blockchain.
- Unauthorized transactions might be submitted to the blockchain
- Unauthorized smart contracts may be created
- Outdated or vulnerable smart contracts may still be active

Another aspect of smart contracts is that they might be used for external auditing organizations. Because they can execute automated audit processes, thus providing real-time reporting. This smart contract use can be considered an evolution to continuous auditing [16].

Vulnerabilities in Smart Contracts

Vulnerabilities in smart contracts can be found in various aspects of the blockchain ecosystem. Those aspects include the programming language of smart contracts (Solidity). In addition, for Ethereum and other Ethereum-based networks, vulnerabilities can be found in the Ethereum Virtual Machine (EVM) and, lastly, in the underlying blockchain of the smart contracts.

We can then identify the following vulnerabilities based on the above distinct categories in smart contracts.

- Solidity
    - Call to the unknown: Primitives used in Solidity to transfer Ether or to call functions, under some circumstances, may call the fallback functions of the callee or the recipient, depending on the case. [17]
    - Exception disorders/mishandling: These disorders mainly happen when some exception is thrown. Exceptions are common in Solidity when the gas runs out, the call stack limit is reached, or the throw is called. Vulnerabilities can surface depending on how the contracts call each other and handle the exceptions.[17][18]
    - Gasless send: Sending Ether to other contracts can lead to an out-of-gas exception. Because the callee has a bound amount of gas units that only allow it to execute a small number of instructions.[17]

- Type casts: The compiler used in Solidity can detect most of the type errors. Direct calls to other smart contracts are also using types. However, the compiler will only check if the interface declared the method, not the actual variables or smart contracts passed to it.[17]
- Reentrancy: It is a common mistake in smart contracts to assume that a non-recursive function cannot be reentered before its termination. This wrong assumption is reached because of the atomicity and sequentially of transactions. Thus, the reentry of functions before they finish execution can lead to serious vulnerabilities, like the one that was exploited in the "DAO ATTACK." This attack led to a significant loss of Ether. [17][18]
- Keeping secrets or Default visibility: This is one of the biggest issues of publicly available blockchains. Solidity fields can be public, meaning other users or smart contracts can access them, or be private. Even in the case of private fields, users need to send appropriate values to miners, which will then be included in transactions on the blockchain. Thus, making it public, technics have been created to keep certain fields secret for a time to address this vulnerability. One of those techniques is timed commitments. [17][18]
- Arithmetic issues: The integer type in Solidity has an upper and lower bound. So, if an integer increases above the upper bound, it is wrapped. In some early solidity versions, this behavior was not flagging any error, thus allowing for easier exploitation.[18]
- Delegate to insecure contracts: The Delegate call is a very dangerous function as it enables the smart contract to reuse the code of the referenced contract and execute it in its context. The vulnerabilities of the reused code will be present or, even worse, in the case of untrusted content, unpredictable changes can occur, including but not limited to even changing the contract owner. [18]
- Self-destruct: This operation is a way to remove the smart contract for future blocks. It will still be present in the history of the blockchain. This function can lead to an attack vector as it is forcibly moving the remaining balance of the smart contract to a new address. Also, any

funds transferred to the address of the self-destructed contract will be permanently lost.[18]

- Tx origin: It is a global variable that stores the address of the original caller of the transaction. If this variable is used for identification or authorization in a smart contract, it can allow an attacker to run code as the contract owner. [18]
- External contract referencing: Smart contracts inevitably will need to reuse code present on other smart contracts. During the audit, it may seem secure, but if a wrong address is passed during deployment, it can lead to serious deprecations. Hard-coded external addresses may be used to avoid this situation. [18]

- EVM
  - Immutable bugs: This is one of the most impactful vulnerability categories. Smart contracts deployed on the network cannot change due to the immutable nature of the blockchain. If the smart contract performs the expected functionality, then users can trust its execution which is guaranteed by the network. The problem appears when a bug is discovered in the smart contract, as there is no direct way to patch that bug. So, the contract developers need to foresee such circumstances and create termination or update mechanisms in the contract implementation. [17]
  - Ether lost in transfer: This vulnerability is caused due to funds sent to an address that is not recoverable in any way. Also, most of the possible address combinations in the network do not belong to anyone, meaning they are orphan addresses, so funds sent to them are forever lost. In order to avoid this problem, developers need to take manual action in verifying the coerciveness of the address provided. [17]
  - Stack size limit: Every time the contract calls a new function or any external one, the call stack associated with the transaction increases by one frame. The call stack is bound to 1024 frames, and after passing this limit, an exception is thrown. This vulnerability can be combined with other vulnerabilities like "exception disorder."[17]
  - Short address/parameter issues: Parameters of smart contract functions are encoded during passing. The length of each parameter when encoded

is 32 bytes. However, because of padding added from the EVM, if the first parameter of a function is 30 bytes long, it can overflow, causing alteration of the second parameter. Which may result in severe damage.[18]

- Freezing Ether: Smart contracts are usually designed to be able to receive and send Ether. Nevertheless, if no withdrawal functionality is planned, the Ether sent to the specific smart contract will freeze as it will not be withdrawable.[18]

- Blockchain
  - Unpredictable state or Transaction order dependence: This is related to the state of the blockchain. By observing the state at any given moment, it is not guaranteed that if a user sends a transaction, this transaction will be executed using the same state as the observed one when sending the transaction. [17][18]
  - Generating randomness: One of the characteristics of the EVM is that execution of code in it is a deterministic process. All miners verifying a transaction are expected to get the same result minus malicious actors, which causes a big impact when contracts need to use random numbers (non-deterministic). One approach to address this is to use pseudo-random numbers. In this case, the initialization string is unique for each miner. One common choice for this seed is the timestamp or hash of a block that will appear in the blockchain. Since, during the run time, each miner has the same view of the blockchain, then the seed will be the same for everyone. Now timestamp or hash of the future block is determined by the transactions and the order of these transactions. Thus a malicious miner can temper this order in the block he is about to add to create a bias in the outcome of the pseudo-random generator.[17][18]
  - Time constraints: Most smart contract applications that want to determine which actions are acceptable in the current state use time constraints. One of the common time constraints to be used by smart contracts is the block timestamp. In this case, if a miner has some stake in a contract, he can slightly manipulate the timestamp of the contract he is about to mine to benefit from it. [17][18]

Background and Related work

Vulnerability Assessment toolkits

We briefly introduce the various vulnerability assessment tools that will be used in the scope of his research.

Vulnerability assessment methods

Three vulnerability assessment methods apply to smart contracts; these methods' usage depends on the expected outcome and depth of the vulnerability assessment. Namely, these methods are Static Analysis, Dynamic Analysis, and Formal Verification.[18]

Static Analysis

Static analysis has been used in Software Security since the early 2000s. The principle behind this method is that a tool scans a software program's source or compiled code, trying to match certain rules or known vulnerability footprints. Although the static analysis approach can identify possible vulnerabilities, it is a method that cannot be fully relied upon. This method is mostly used to assist in identifying vulnerabilities prior to releasing software.[19]

In Smart contract vulnerability assessments, static analysis is used to identify known vulnerabilities in bytecode and solidity source code. Slither is a tool to uses static analysis in a multilayered way in order to detect vulnerabilities, even in code optimizations, that some other tools might miss. [20] Vandal is another tool that leverages static analysis to detect vulnerabilities in smart contracts. Vandal uses a decompiler in its analysis chain to recompile bytecode to a higher-level representation to surface logic relations.[21]

Static analysis is an important vulnerability assessment method and the most widely used one.

Dynamic Analysis

Dynamic analysis is different from static analysis in that it analyzes the code while the code is executing (run-time). This method does not analyze the code base but the code executed. Such an approach makes Dynamic Analysis impervious to obfuscation and

dead code injection attempts and also highly minimizes the effect of self-modifying code on the assessment process.[22]

In smart contract vulnerability assessment, the Dynamic Analysis performed by the various tools uses symbolic analysis on the bytecode. Then, using a custom EVM implementation, the smart contract is symbolically executed using a predefined depth. This symbolic execution continues to search all possible execution paths until a vulnerability is found or the possible execution paths are exhausted. Lastly, the results of the symbolic analysis are validated using the concrete validation process.[23]

MAIAN is a tool that uses dynamic analysis to find vulnerabilities in Ethereum smart contracts.[24]

Formal Verification

The formal verification method principle is the use of mathematics to prove the correctness of various properties of a program. These properties include functional correctness, run-time safety, and others.[23] Various tools use this method to perform vulnerability assessments on smart contracts. Oyente is a tool that leverages Formal Verification to formalize the semantics of Ethereum smart contracts [25] and performs vulnerability assessment.[26]

Oyente

Oyente is one of the first vulnerability assessment toolkits on smart contracts that surfaced. By using symbolic execution and formal verification, Oyente can detect various known vulnerabilities directly from the bytecode without having access to higher-level representation [25]. Oyente identified security bugs, including Transaction Ordering Dependance, Timestamp difference, Mishandled Exceptions, and Reentrance Vulnerability [25]. When Oyente was released, it was executed on the first 19,366 smart contracts finding almost half of them (8,833) vulnerable.[25]

Osiris

Osiris is a tool that combines symbolic execution and taint analysis which is an approach of formal verification to detect integer vulnerabilities in Ethereum smart contracts [27]. Osiris can detect multiple integer bugs, such as Arithmetic bugs, Truncation bugs, and Signedness bugs. Osiris can take as input both bytecode and source code, then using symbolic analysis, it generates a Control Flow Graph and executes the different paths. Then the results are passed on to taint analysis process.[27]

Osiris was evaluated against other toolkits, such as Mithril[28] and Zeus[29]. It was found to identify truncation and signedness bugs that the other tools could not.

Methodology and Definitions

For this research, we will download the entire Ethereum blockchain in order to be able to process that locally. To download the blockchain, we will need to fetch and store the information of each block in the blockchain. Due to the large number of requests needed to download all the blocks using public infrastructure, which is rate limited, we will not cut; thus, we used a privately hosted solution. Part of the block information is the transactions included in the specific block. Each transaction consists of information like from_address, to_address, and input. In order to identify smart contracts, we will need to scan all Ethereum blockchain transactions and search for a transaction to the "null" address. After finding these transactions, we will need to verify if they are contract-creation transactions and find the address of the newly created contract. To achieve this result, we are fetching the transaction receipt. The information there includes if a contract was created and the address of the newly created contract.

In the next stage, since we already have gathered all the contracts, including the bytecode, which is present in the transaction's input field, we will use the Etherscan service to collect information on which contracts are verified and which are not. We will fetch ABI information and source solidity code for the verified contracts.

Finally, we are performing a vulnerability assessment in each of the smart contracts found in the previous step. Due to time constraints in the scope of this research, we will use only the bytecode of each smart contract. To perform the vulnerability assessment, we used four open-source tools, and the results of each tool for each contract were collected. First, a virtual machine with docker technology was used to install these tools. Next, each tool was installed there using their docket releases. Finally, a web server was set up inside the virtual machine to create an API to allow external requests to submit smart contract bytecode. Then the submitted code would be passed to the various tools, and the results would be collected and returned. This final process is the most time-consuming one, so four virtual machines were used over three months to analyze for vulnerabilities in all the found contracts.

Definitions

In this chapter, we will define the various terms used going forward.

ABI

Application Binary Interface or ABI is the translation matcher between high-level language function names and arguments into binary code incorporated in the bytecode. The corresponding ABI file is needed to retrieve the initial solidity source code from the bytecode.

Account

Accounts are like wallets; it is where the Ether coins are stored. Accounts and account balances are stored on a table and are part of the EVM state. In Ethereum, there are two types of accounts. [10]

- External Accounts: Accounts outside the scope of EVM are controlled by the owner of their public and private keys. It is assumed that persons are the owners of these accounts.
- Contract Accounts: These accounts belong to smart contracts deployed on the EVM and are controlled by those contracts.

Address

The address is a 160-bit string that is used to identify accounts. Both externally owned and contract accounts have addresses.[10]

Block

A block is the building unit of the blockchain. Blocks are used in Ethereum to store transactions submitted by external accounts and a reference to the previous block. In the context of this research, we are using the following properties of each block [10]:

- Hash: Consists of a 256-bit Keccak hash of the entire block header.
- ParentHash: The hash of the parent (previous) block.
- Timestamp: The Unix timestamp of the block creation date.
- Number: Incremental value of the total number of previous blocks. Genesis block has the number zero.
- Transactions: Array containing the transactions submitted to the block.

Contract

An informal term can mean code run in the EVM and is associated with an account or an address associated with a contract account.

EVM

Ethereum Virtual Machine (EVM) is a simple virtual machine that forms the key part of the execution layer responsible for running the EVM code associated with a contract account.

EVM Code

It is the bytecode representation of Solidity code that can be natively executed in the Ethereum Virtual Machine.[10]

EVM Assembly

Human readable representation of EVM code.[10]

Gas

In order to use the Ethereum network to make any transaction, a computation cost is needed. Gas is that cost and is exclusively paid in Ether and represented in Wei.[10]

Solidity

The high-level language is currently used to write code that is then compiled into bytecode. The whole Ethereum community supports Solidity.[17]

Transaction

The transaction is a single instruction cryptographically signed by an external to the Ethereum network scope account. The sender of a transaction cannot be a smart contract. A transaction can transfer the Ethereum balance between two accounts or include a message input. When a transaction is included in a block and executed, a transaction receipt is created. In the context of this research, we are using the following properties of transactions [10]:

- BlockHash: Hash of the block in which the transaction was included.
- BlockNumber: Number of the block in which the transaction was included
- Hash: Consists of a 256-bit Keccak hash of the transaction
- Gas: Total amount of gas spent to execute the transaction message and all other contract messages resulting from it.
- GasPrice: The price paid in Wei per gas unit for the cost resulting from the transaction execution.
- Nonce: Incremental value equal to the number of transactions submitted by the sender.

- From: The address of the sender who submitted the transaction.
- To: The destination account of the transaction.
- Input: Instruction message submitted with the transaction. For contract-generating transactions, bytecode is present in this field.
- TransactionIndex: Index of the transaction in the block it was included.
- Value: Number of Wei transferred to the transaction recipient.

Transaction Receipt

Transaction receipt consists of the state alteration that resulted after the transaction execution. It includes the status of the transaction, success or failure. It also includes contract-related information if the transaction created a new smart contract. In the context of this research, we are using the following properties of transactions receipts [10]:

- BlockHash: Hash of the block in which the receipt transaction was included.
- BlockNumber: Number of the block in which the transaction of the receipt was included
- TransactionHash: The hash of the transaction of the receipt.
- GasUsed: Total amount of gas spent to execute the transaction message and all other contract messages resulting from it.
- From: The address of the sender who submitted the transaction.
- To: The destination account of the transaction.
- TransactionIndex: Index of the transaction in the block it was included
- ContractAddress: The address of the smart contract that was interacted with by the transaction. The newly created contract address for a contract generation transaction.
- Status: After the Byzantium fork transaction receipt provides a status of false when failed and true in success.

Wei

Wei represents the smallest possible division of Ether. One Wei equals 10^{-18}Ether.

Methodology

This chapter will analyze the steps taken and the difficulties encountered in this research process. Below, figures 1 and 2 give a graphical representation of the implemented

process. There were multiple steps involved in achieving the goal of this thesis. In particular, they are:

1. Obtaining the blockchain block information from the network.
2. Scan the blocks for possible contract creation transactions.
3. Obtain transaction receipts for possible contract creation transactions.
4. Identify non-valid contracts, and find verified contracts and their ABI.
5. Identify tools to be used for vulnerability assessment.
6. Integrate the tools into a virtual machine.
7. Create an API to handle submitted vulnerability assessment requests.
8. Extract useful results

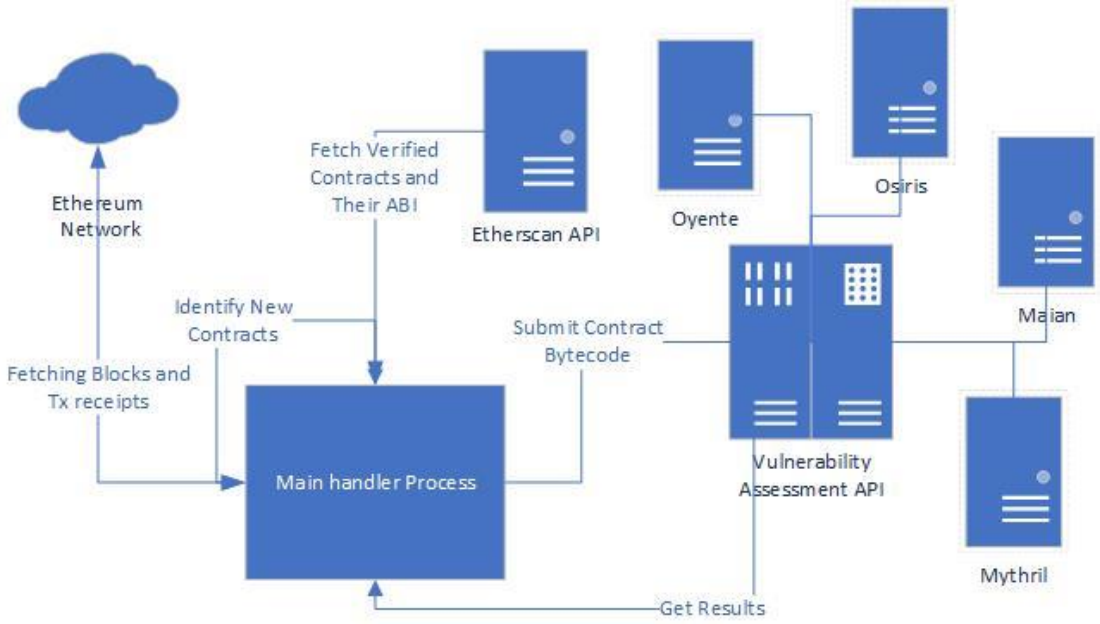


Figure 1: High-level Overview

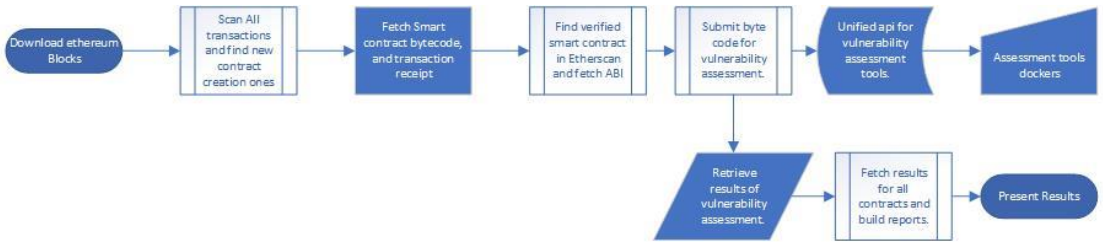


Figure 2: Internal Process Flow

Obtain the blockchain block information from the network.

The first step was to obtain the blocks of the blockchain and store them locally for offline processing. For this purpose, an Ethereum network full node is needed to query and obtain blocks. However, since the public infrastructure is being throttled and thus

is not usable, we used a private Ethereum full node hosted service. There is, of course, the option to host an Ethereum node, but it has high technical, computational, and storage requirements. Also, due to the huge size of the blockchain, requiring more than one Terabyte of storage, compression was used to store the blocks into files locally. With compression, the total amount of storage occupied was approximately 360 Gigabytes.

The obtained blocks were stored locally in files containing JSON objects with the following structure:

```
{
        "blocknumber": { blockinformation },
        ...
}
```

Figure 3: Local Block Information Storage Structure

An example of block information is presented below:


```
{
  "difficulty": "2856555416664183",
  "extraData": "0x7575706f6f6c2e636e2d32",
  "gasLimit": 12451224,
  "gasUsed": 12338462,
  "hash":
    "0x411e667e75045da1b4a33460b224d8bccc5198450eeecfc574a7ea3673cfd5c9",
  "logsBloom":
    "0x1637e9e14977025a49f289228c311ee26c610c418cd640f08c49a8586046d9c1845090
    875056a1805605c586521a4d03cb34950e8f0204907b61c447512416eb138019f48847c48
    77a27a16f1d805e6d0218bfa62764f2e444cae7eda86019109a20564602019042030d8fe2
    a88e1a4c0230fb43042a0451a8dd26933f802804693ad0c34b3a40405d5102a3a8ab4ed80
    e54e8cb61608d3b97c03cceb1774c9193d32ca061e5a94e46a6f3b09a04993e239c30cfd9
    82e040c06fd9a0ae83e0d4e1a406320c3a2574ca126625a8429cccfe58d24fadca399142e
    ae076d7ea6400fc112da0101b10349a2ca6510dc4028237a50230c1b1c27444a00025a826
    101c",
  "miner": "0xD224cA0c819e8E97ba0136B3b95ceFf503B79f53",
  "mixHash":
    "0xe857efaf34a5735576fd3bee5a16782c1d46ce0ab05b633b8475b3fd8a1726a1",
  "nonce": "0x7d1b59406c7c4404",
  "number": 10782001,
  "parentHash":
    "0x045dea6d898dab54669fba32a0f2fba6f0af7b5313021eae7936cd4096d4d806",
  "receiptsRoot":
    "0x07e12194f697c7c7d1c95fef898b2495669a09fd80836e5dbba5e41b2b0927d3",
  "sha3Uncles":
    "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
  "size": 49129,
  "stateRoot":
    "0x80e8a3ef72c708e77f3da0e056bd2226285a0938f02bd55f99ee240716492f89",
  "timestamp": 1599052321,
  "totalDifficulty": "17165177682045200186539",
  "transactions": [],
  "transactionsRoot":
    "0x39eff5a9aacbead9061fb75faeb5b6ba66769889d8a89cee5af6935f2123d069",
  "uncles": []
}
```


Figure 4: Block Information

Most of the information inside a block was analyzed previously. This research's helpful and important information is the number, timestamp, and transactions array.

Scan the blocks for possible contract creation transactions.

Having obtained the blockchain blocks, we can identify possible contract-creation transactions. The most important characteristic of a contract creation transaction is that the destination of the transactions is the "null" address. That does not mean that all transactions with destination the "null" address are contract-creating transactions, that whoever will be the focus in future steps. For this step, we scan the entire blockchain and look for a transaction to the "null" address. Then we store them locally for future

processing. For transactions, we have not used compression as the storage required was not in the scale of the blockchain. Also, compression inherently adds overhead to computations and data processing.


```
{
  "blockHash":
"0xaebf48d354504e006ce6344c213712d3b7f72218a61200c9364a793108f124b8",
  "blockNumber": 1000050,
  "from": "0x7FA200646216300e3dEc61925aEf2bCb5cb08904",
  "gas": 1000000,
  "gasPrice": "50000000000",
  "hash":
"0xa8b3733cb556f942c8ce9b2fca536e8d364c4535fe7dc0b95151cdd7eaf3a313",
  "input":
"0x6060604052600261010860005055604051610156380380610156833981016040528
05160805160a05191909201919080838381516001908101815560009600160a060020
a033216906002906003839055918352506101026020526040822055 5b8251811015610
0eb57828181518110156100025790602001906020020151600160a060020a031660026
0005082600201610100811015610002579090016000508190555080600201610102600
0506000858481518110156100025790602001906020020151600160a060020a0316815
2602001908152602001600020600050819055506001016100605 65b816000600050819
0555050505080610105600050819055506101 0f6201518042049056 5b6101075550505
0506031806101256000396000f3003660008037602060003660003473273930d21e01e
e25e4c219b63259d214872220a261235a5a03f21560015760206000f30000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000100000000000000000000000000000000000000
00e3dec61925aef2bcb5cb08904",
  "nonce": 4,
  "to": null,
  "transactionIndex": 0,
  "value": "0",
  "type": 0,
  "v": "0x1b",
  "r":
"0x987332e524aa75db80c7f7f56c80af842d181037f7c3b30d438fe9ed42bb50d4",
  "s":
"0x3af456bbf67bea65d72324f3d817469a16aadf6bcb27fd783f6d66c4e2b2277c"
}
```


Figure 5: Example of contract creation Transaction

In the above JSON formatted text, we can see the information contained in a contract-creating transaction. The most notable fields are the "input," which contains the bytecode of the newly created contract. Next, the "to" field sets the recipient of the transactions, which in this case is the "null," and finally, the "hash," which will be used to fetch the transaction receipt. Other important fields that we will be keeping for this research include "blockNumber," "from," "gas," "gasPrice," "nonce," "transactionIndex," "value," "type," "v," "r," "s."

Another important thing to notice is the starting bytes of the bytecode "0x6060604052". These starting bytes make the free memory pointer initialization and are how every solidity program starts. This sequence corresponds to the following EVM instructions "PUSH 0x60", "PUSH 0x40", and "MSTORE." In solidity version v0.4.22 and later, this initialization sequence changes to "0x6080604052". This sequence is important to help us identify valid solidity code and, thus, valid smart contracts.

Obtain transaction receipts for possible contract creation transactions.

The next step will combine the information received from the previous steps into a new data structure. The new data structure will be used and populated in the following step. In this step, we are fetching transaction receipts from the network. As stated before, the transaction receipt contains important information, including the resulting contract address and the transaction result (success or failure). All information from the block, transaction, and transaction receipt can be considered important metadata of contract creation. Thus we store all this information in the following form:

```
{
      "contractAddress": { contractMetadata },
      ...
}
```

Figure 6: Local Contract Information Storage Structure


```
{
  "transactionHash":
"0xa8b3733cb556f942c8ce9b2fca536e8d364c4535fe7dc0b95151cdd7eaf3a313",
  "blockHash":
"0xaebf48d354504e006ce6344c213712d3b7f72218a61200c9364a793108f124b8",
  "blockNumber": 1000050,
  "logs": [],
  "contractAddress": "0xD90E6f0674B2DF67f0ef171C6b963B183424E32f",
  "effectiveGasPrice": 50000000000,
  "cumulativeGasUsed": 205847,
  "from": "0x7fa200646216300e3dec61925aef2bcb5cb08904",
  "gasUsed": 205847,
  "logsBloom":
"0x00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000",
  "status": true,
  "transactionIndex": 0,
  "type": "0x0"
}
```


Figure 7: Transaction Receipt Information

In the JSON text of figure 7, we can see an example of a transaction receipt regarding the transaction in the previous step. We can identify that the newly created contract address is "0xD90E6f0674B2DF67f0ef171C6b963B183424E32f" and that the status is true, which signifies that the transaction execution and, thus, the creation of the contract has succeeded. Other important information that will be kept from the transaction receipt and will get used later include "transactionHash," "blockNumber," "effectiveGasPrice," "cumulativeGasUsed," "from," "gasUsed," "transactionIndex," "type." The values overwrite similar fields to the transaction in the transaction receipt. Figure 8 below is an example of the result stored object after this step.


```
{
  "0xD90E6f0674B2DF67f0ef171C6b963B183424E32f": {
    "blockNumber": 1000050,
    "contractAddress": "0xD90E6f0674B2DF67f0ef171C6b963B183424E32f",
    "cumulativeGasUsed": 205847,
    "effectiveGasPrice": 50000000000,
    "from": "0x7fa200646216300e3dec61925aef2bcb5cb08904",
    "gasUsed": 205847,
    "transactionHash":
"0xa8b3733cb556f942c8ce9b2fca536e8d364c4535fe7dc0b95151cdd7eaf3a313",
    "transactionIndex": 0,
    "type": "0x0",
    "status": true,
    "gas": 1000000,
    "gasPrice": "50000000000",
    "input": "0x6060604052...",
    "nonce": 4,
    "value": "0",
    "v": "0x1b",
    "r":
"0x987332e524aa75db80c7f7f56c80af842d181037f7c3b30d438fe9ed42bb50d4",
    "s":
"0x3af456bbf67bea65d72324f3d817469a16aadf6bcb27fd783f6d66c4e2b2277c",
  }
}
```


Figure 8: Combined Contract Information from a block, a transaction, and a receipt.

Identify non-valid contracts, find verified contracts and their ABI

After fetching the initial data from the blockchain, we will not be fetching additional ones as we already have everything we need. Therefore, this section will scrap all the imported contracts to obtain meaningful statistics. The statistics will include the total found contract count, the total successfully created contracts and the number of contracts for some specific solidity versions.

Before diving into the specific statistics, we would like to explain a bit about the initial part of the input of a contract. By observing, someone will notice that the vast majority of smart contract code starts with either "0x6060604052" or "0x6080604052" this sequence of bytes (or opcodes, if you will,) initiates the memory pointer. This process is the initialization of every solidity program. Nevertheless, we notice that there are two variations in it. According to solidity compiler documentation, "0x60" was the address to the free memory pointer for solidity versions up to 0.4.21 [30]. After v0.4.22 and up to the latest version (0.8.17 at the time of writing), the free pointer initialization points to "0x80". This change allowed for more system-owned memory size before the program-owned memory started [31]. In figure 9, there is a representation of the bytecode prologue in opcode:

```
00:  6080 PUSH1 0x80
02:  6040 PUSH1 0x40
04:  52   MSTORE
```

Figure 9: Opcodes in bytecode prologue

The main point of this step is to find which smart contracts have verified their source code by uploading their Solidity ABI file on Etherscan. For this purpose, we will query every smart contract found in the previous steps through the Etherscan API to find any uploaded ABI file. If the ABI file is submitted, then the smart contract is verified, and we store the ABI file contents locally. Through the contents of the ABI file, we can later reconstruct the initial solidity source code of the smart contract. After scanning all smart contracts, we found 4,602,424 smart contract creation transactions, 3,877,891 of which were successful*.

Metric	Value
Total Identified Smart Contract Creation Tx	4,602,424
Total Successful Smart Contract Creation Txs*	3,877,891
Verified Smart Contracts	708,937
Duplicate contracts created	0
Prior to Solidity v0.4.21, smart contracts	2,635,913 (129,557 verified)
After Solidity v0.4.22 smart contracts	1,717,743 (409,696 verified)
Other smart contracts	248,768 (169,684 verified)

Figure 10: Results after scanning the blockchain for contracts

*Smart contract transactions before the Byzantium hard fork [32] do not include the status field in their receipts. This field was introduced with EIP-658. [33]

Since we have considered the difference in the initialization code, we will now present the initial statistics on the smart contracts retrieved from the blockchain.

For solidity v0.4.22, we found 1,717,743 smart contracts, of which 409,696 are verified on Etherscan. However, due to the high computational requirements for vulnerability assessment, we will review only this category of smart contracts in this research. The other categories will be the target of future work.

For solidity v0.4.21 and before, we found 2,635,913 smart contracts, of which 129,557 are verified on Etherscan. Also, we found 248,768 smart contracts that do not comply with the standard solidity initialization code, and further examination of their origin is needed. As mentioned before, these categories of smart contracts will be the subject of future work.

Construct solidity code

Identify tools to be used for vulnerability assessment.

It is time to start planning for the vulnerability assessment process. In this process, we will use open-source smart contract vulnerability assessment tools to identify vulnerabilities in most smart contracts found in the previous steps.

For the process's simplicity and the assessment's uniformity, we searched for tools that can provide results with only Ethereum bytecode input. Another prerequisite was that these tools have docker implementations. Finally, we tried to cover various vulnerabilities with the selection of tools and different tools to spot different vulnerabilities, as the purpose of this research is not to evaluate the various tools. Below we are going to present these tools and sample results.

Oyente

As mentioned previously, Oyente is a tool developed to help in the pre-deployment mitigation of vulnerabilities. It uses symbolic execution by representing the symbolic input values as symbolic expressions. Using symbolic execution, Oyente can statically reason the program path by path. This approach is less expensive computationally than dynamic execution, which would require a simulation of the execution environment.[34] The vulnerabilities discovered by this tool and used in our research

are "callstack," "reentrancy," "time_dependency," "integer_overflow," "integer_underflow," and lastly, "money_concurrency." We chose this tool as it is one of the most commonly used and recognized. Figure 11 is an example result for Oynete that was collected during our research.[26]


```
"oyente": {
      "vulnerabilities": {
        "callstack": false,
        "reentrancy": true,
        "time_dependency": false,
        "integer_overflow": [],
        "integer_underflow": [],
        "money_concurrency": true
      },
      "evm_code_coverage": "95.8"
    },
```


Figure 11: Example of Oyente results

Osiris

Like with Oyente, we have already mentioned Osiris before. This tool is expanding upon Oyente's detection abilities. Osiris combines three components to analyze the smart contract bytecode for vulnerabilities: Symbolic Analysis, Taint Analysis, and Integer Error Detection. [27] The vulnerabilities discovered by this tool and used in our research are "callstack," "reentrancy," "modulo," "division," "signedness," "underflow," "time_dependency," "overflow," "money_concurrency" and lastly "truncation." We chose this tool as it is one of the most commonly used tools and provides a wide range of detection abilities. Figure 12 shows an example result for Osiris that was collected during our research.

```
"osiris": {
      "execution_paths": "50",
      "callstack": false,
      "reentrancy": true,
      "modulo": false,
      "division": false,
      "execution_time": "20.5002510548",
      "signedness": false,
      "evm_code_coverage": "58.0",
      "underflow": "\nOpcode: SUB\nInput: [Iv, Ia_store_6]\nOutput: [Iv
+\n1157920892373161954235709850086879078532699846656405640394575840079131
29639935*\nIa_store_6]",
      "time_dependency": false,
      "assertion_failure": false,
      "timeout": true,
      "overflow":   "\nOpcode:   MUL\nInput:   [32,   Id_12]\nOutput:
[32*Id_12]\nOpcode: ADD\nInput: [128, 32 + 32*Id_12]\nOutput: [160 +
32*Id_12]\nOpcode: ADD\nInput: [32, 196 + 32*Id_12]\nOutput: [228 +
32*Id_12]\nOpcode: ADD\nInput: [32, 228 + 32*Id_12]\nOutput: [260 +
32*Id_12]\nOpcode: ADD\nInput: [10000000000000000, Ia_store_6]\nOutput:
[10000000000000000   +   Ia_store_6]\nOpcode:   MUL\nInput:   [32,
Id_12]\nOutput:   [32*Id_12]\nOpcode:   ADD\nInput:   [32,   164   +
32*Id_12]\nOutput:   [196   +   32*Id_12]\nOpcode:   ADD\nInput:   [32,
32*Id_12]\nOutput: [32 + 32*Id_12]",
      "money_concurrency": true,
      "truncation": false
    },
```


Figure 12: Example of Osiris results

Mythril

Mythril is a security analysis tool used to detect vulnerabilities in EVM bytecode. It detects vulnerabilities in Ethereum and other EVM-compatible blockchains. It uses symbolic execution, SMT solving, and taint analysis to detect security vulnerabilities.[28] Mythril provides information for the found vulnerabilities, including description, detection function, severity, and others. We decided to use this tool because it leverages SMT solving, a technique not used by the previous tools. Figure 13 is an example result for Mythril that was collected during our research.

```
"mythril": {
      "error": null,
      "issues": [
        {
          "address": 0,
          "contract": "MAIN",
          "description": "An assertion violation was triggered.\nIt is
possible to trigger an assertion violation. Note that Solidity assert()
statements should only be used to check invariants. Review the
transaction trace generated for this issue
and either make sure your program logic is correct, or use require()
instead of assert() if your goal is to constrain user inputs or enforce
preconditions. Remember to validate inputs from both callers (for
instance, via passed arguments) and callees (for instance, via return
values).",
          "function": "fallback",
          "max_gas_used": 0,
          "min_gas_used": 0,
          "severity": "Medium",
          "sourceMap": 0,
          "swc-id": "110",
          "title": "Exception State"
        }
      ],
      "success": true
    },
```


Figure 13: Mythril results example

MAIAN

MAIAN is another tool that uses symbolic analysis to extract execution traces from smart contracts. MAIAN can check for prodigal contracts, suicidal contracts, and greedy contracts. Prodigal is the contract that can leak Ether, given a certain execution path. Suicidal are contracts that can be killed following a certain execution path. Lastly, greedy contracts can accept Ether but have no way to send it out. We chose this tool as it provides information for vulnerabilities not covered by the other previously used tools. Figure 14 shows an example result for MAIAN that was collected during our research.

```
"maian": {
      "suicidal": Check if contract is SUICIDAL
    [] Contract address: 0xaFFECAFEAFfECaFEaFFecAfEAFfecAfEAffEcaFE
    [] Contract bytecode: 6080604052600436106100d057600000000...
    [] Bytecode length: 10418
    [] Blockchain contract: False
    [] Debug: False
    [-] The code does not contain SUICIDE instructions, hence it is not
vulnerable
      "prodigal": Check if contract is PRODIGAL
    [] Contract address: 0xaFFECAFEAFfECaFEaFFecAfEAFfecAfEAffEcaFE
    [] Contract bytecode: 6080604052600436106100d0576000357c0100000...
    [] Bytecode length: 10418
    [] Blockchain contract: False
    [] Debug: False
    [] Search with call depth: 1: 1111111
    [] Search with call depth: 2
1122222221222222212222222122222221222222212222222
    [] Search with call depth: 3:
1122233333332333333323333333233333332333333312222333333323333333233333332333333
3233333331222223333333233333332333333312222223333333
    [+] No prodigal vulnerability found,
      "greedy": 'Check if contract is GREEDY
    [] Contract address: 0xaFFECAFEAFfECaFEaFFecAfEAFfecAfEAffEcaFE
    [] Contract bytecode: 6080604052600436106100d0576000357c01000...
    [] Bytecode length: 10418
    [] Debug: False
    [-] Contract can receive Ether
    [-] No lock vulnerability found because the contract cannot receive
Ether'
    }
  }
```

Figure 14: Maian results example

Integrate the tools into a virtual machine.

As mentioned previously, we preferred tools with docker versions to allow for a uniform and easy integration. So, after selecting the tools, we are now installing them

on a virtual machine that will allow us to create a common API to call the tools and retrieve the results. Debian 5.10 was used as the operating system of the virtual machine. Steps to prepare a virtual machine for the installation of the tools and the API creation:

- Install Docker for Debian
- Install PHP. (Version 7.4.30 was used)
- In php.ini, we removed the "disabled_functions" to allow the "shell_exec" method.
- Added www-data in sudoers with the privilege to execute all commands without a password.

Although the above steps compromise the security of the virtual machine, it is necessary to allow for a unified integration and remote execution through the API.
Following the virtual machine's preparation, we will download and run the following docker images. We will also configure the instances to run persistently through restarts of the virtual machine:

- luongnguyen/oyente
- christoftorres/Osiris
- smartbugs/maian
- mythril/myth

For each of these images, the command sequence executed was:

1. sudo docker pull luongnguyen/oyente
2. sudo docker run -dit --restart always --name oyente luongnguyen/oyente
3. sudo docker start oyente

```
eval@scevalvm:~$ sudo docker ps
CONTAINER ID   IMAGE                  COMMAND       CREATED       STATUS       PORTS     NAMES
670b4bf9fa46   smartbugs/maian        "bash"        3 weeks ago   Up 11 days             maian
894900e46a12   christoftorres/osiris  "/bin/bash"   3 weeks ago   Up 11 days             osiris
0ba68012f4de   luongnguyen/oyente     "/bin/bash"   3 weeks ago   Up 11 days             oyente
eval@scevalvm:~$
```

Figure 15: Running containers of Vulnerability Assessment Tools

In figure 15, we can see the running containers. We notice that Mythril is not on the list. That is because mithril will get started from the beginning every time a smart contract is submitted for evaluation and will terminate after the evaluation is finished.

Create an API to handle submitted vulnerability assessment requests

Now that the docker containers are inside the virtual machine, we will create a simple API implementation in PHP. The purpose is to evaluate the submitted bytecode as requested by the API call and tools and return the results in JSON format. Because most tools do not include remote capabilities, we write the bytecode in a file inside the docker containers, perform the assessment, and retrieve the result. The sample code from the implementation for Oyente is below the complete code for the API resides in Appendix A.

```
// Write bytecode.
shell_exec("sudo docker container exec oyente bash -c \"echo {$bytecode} > /oyente/oyente/contract.sol\"");
// Execute Assessment.
shell_exec("sudo docker container exec oyente bash -c \"python /oyente/oyente/oyente.py -s /oyente/oyente/contract.sol -j -t 5000\"");
// Retrieve result.
$output = shell_exec("sudo docker container exec oyente bash -c \"cat /oyente/oyente/contract.sol.json\"");
```

Figure 16: Vulnerability assessment API Oyente handle

One of the main characteristics of the API implementation is that the call to the endpoint can control which of the tools present on the platform will perform the vulnerability assessment. Also, through the call to the API, we can specify if the submitted code is bytecode or solidity code.

Having reached this point, everything else is automated, and we are waiting for the process to finish. The vulnerability assessment process is the lengthiest one. It is estimated that for the approximately ~1.7 million contracts targeted in this research, it took three months. The setup was a desktop PC with Intel core I7 6700K, 32 GBs of Ram, and SSD storage. Six clones of the vulnerability assessment VM were running on the computer, and the application was load-balancing between them to fetch the results.

Extract useful results

Finally, the last step after collecting the vulnerability assessment results for all the targeted smart contracts is processing them. We will process these results and produce useful metrics and statistics to understand the current security state for on-chain contracts.

Below we are presenting some questions we will try to answer by processing the collected results.

1. The number of vulnerabilities detected and contracts vulnerable to each one of them.
2. The number of total vulnerable contracts detected, with additional info for contracts with more than one vulnerability.
3. The number of vulnerable contracts detected per tool. Also, additional info for vulnerable contracts detected by multiple tools.
4. How many verified smart contracts are vulnerable?
5. Smart contracts with at least one critical vulnerability detected.

Discussion and Results

Since we have already analyzed the step we have taken to lead us here, we will present and elaborate on the results collected from the vulnerability assessment process. Based on the scope of smart contracts we focused on in this research, we have identified and evaluated a total of 1.587.700 smart contracts, of which 1.574.790 have succeeded in being created. Of this total amount, 356.554 smart contracts have been verified in the Etherscan platform, and 344.496 verified smart contracts succeeded and were created. Therefore, only the successfully created smart contracts will be considered and considered in the results.

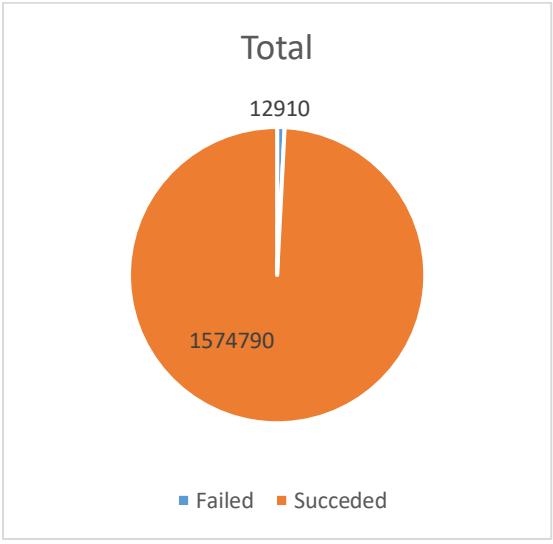


Figure 17: Total Succeeded Smart Contract Creation Transactions

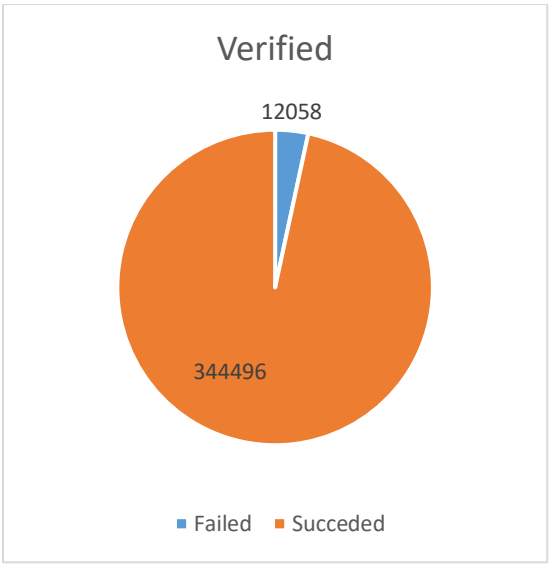


Figure 18: Verified Succeeded Smart Contract Creation Transactions

Results Per Tool

Following the initial count, we will present the vulnerabilities found per tool. Oyente can identify six vulnerabilities. Oyente found 125598 contracts with call stack vulnerability for the assessed smart contracts, 70355 of which had verified contract code. Identified contracts with reentrancy vulnerability were 146, with 36 having verified code, and 4573 contracts had Time Dependency vulnerability, of which 4570 had verified code. Finally, the money concurrency vulnerability had 800 smart contracts, of which 456 were verified. Oyente did not identify any integer overflow or underflow vulnerability for the assessed contracts.

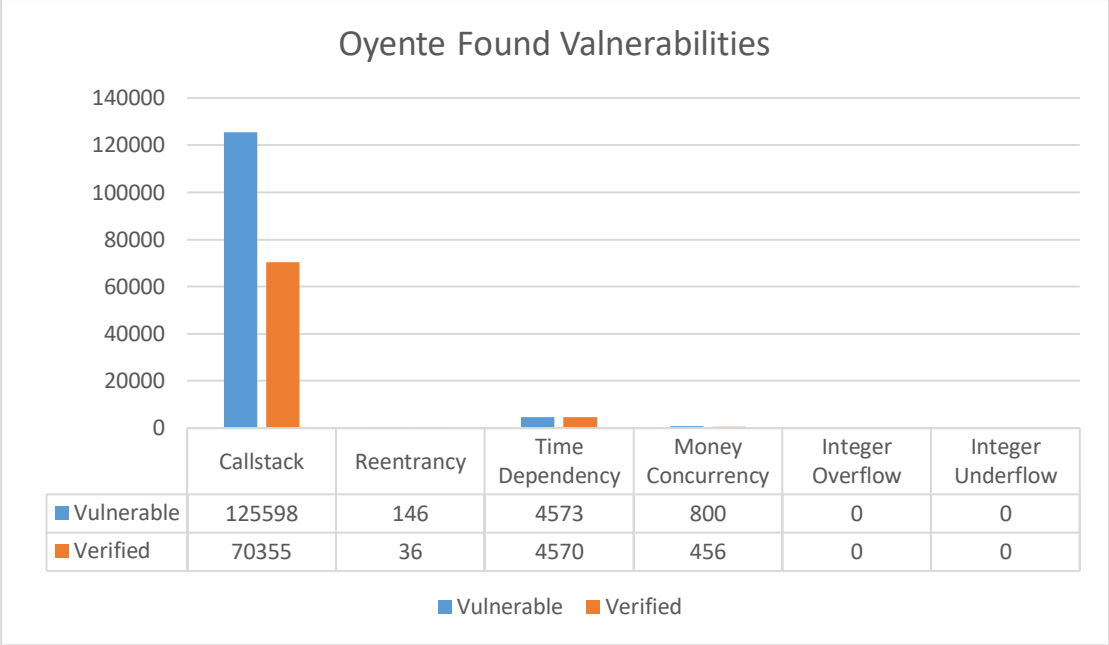
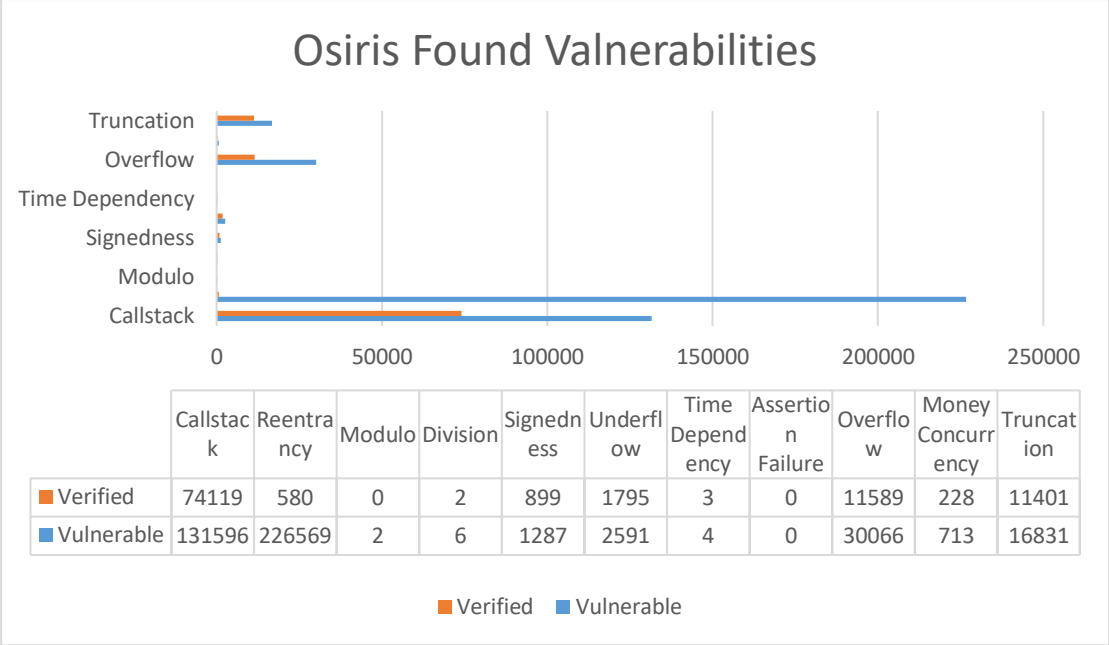


Figure 19: Oyente Found Vulnerabilities

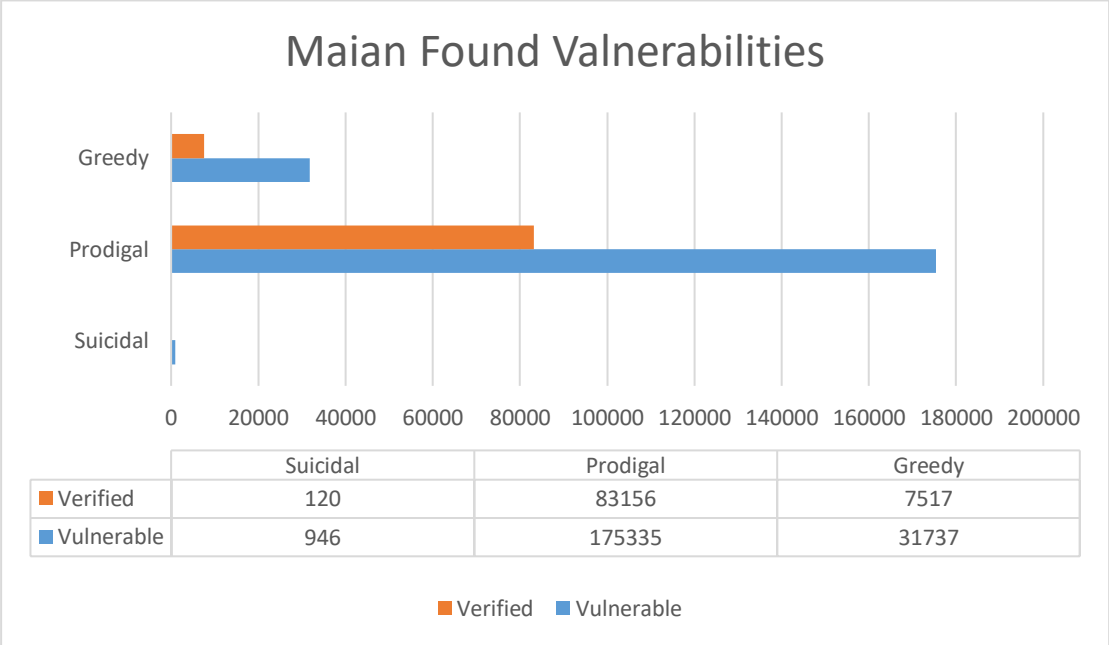
Next, we are going to present results for the Osiris tool. For callstack vulnerability, 131596 smart contracts were found vulnerable, and 74119 were verified. For reentrancy vulnerability, 226569 smart contracts were found vulnerable, 580 of which are verified. Two smart contracts were found vulnerable for modulo vulnerability, 0 of which are verified. Six smart contracts were found vulnerable for division vulnerability, 2 of which are verified. 1287 smart contracts were found vulnerable for signedness vulnerability, and 899 were verified. For integer underflow vulnerability, 2591 smart contracts were found vulnerable, 1795 of which are verified. Four smart contracts were found vulnerable to Time Dependency vulnerability, 3 of which are verified. 0 smart contracts were found vulnerable to Assertion Failure vulnerability, and 0 were verified. For Integer Overflow vulnerability, 30066 smart contracts were found vulnerable, 11589 of which are verified. 713 smart contracts were found vulnerable to Money Concurrency vulnerability, and 228 were verified. Finally, 16831 smart contracts were found vulnerable to Truncation vulnerability, 11401 of which are verified.



	Callstack	Reentrancy	Modulo	Division	Signedness	Underflow	Time Dependency	Assertion Failure	Overflow	Money Concurrency	Truncation
Verified	74119	580	0	2	899	1795	3	0	11589	228	11401
Vulnerable	131596	226569	2	6	1287	2591	4	0	30066	713	16831

Figure 20: Osiris Found Vulnerabilities

Next, we are going to present results for the Maian tool. For suicidal vulnerability, 946 smart contracts were found vulnerable, and 120 were verified. For prodigal vulnerability, 175335 smart contracts were found vulnerable, 83156 of which are verified. Finally, a greedy vulnerability was found in 31737 smart contracts, 7517 of which are verified.

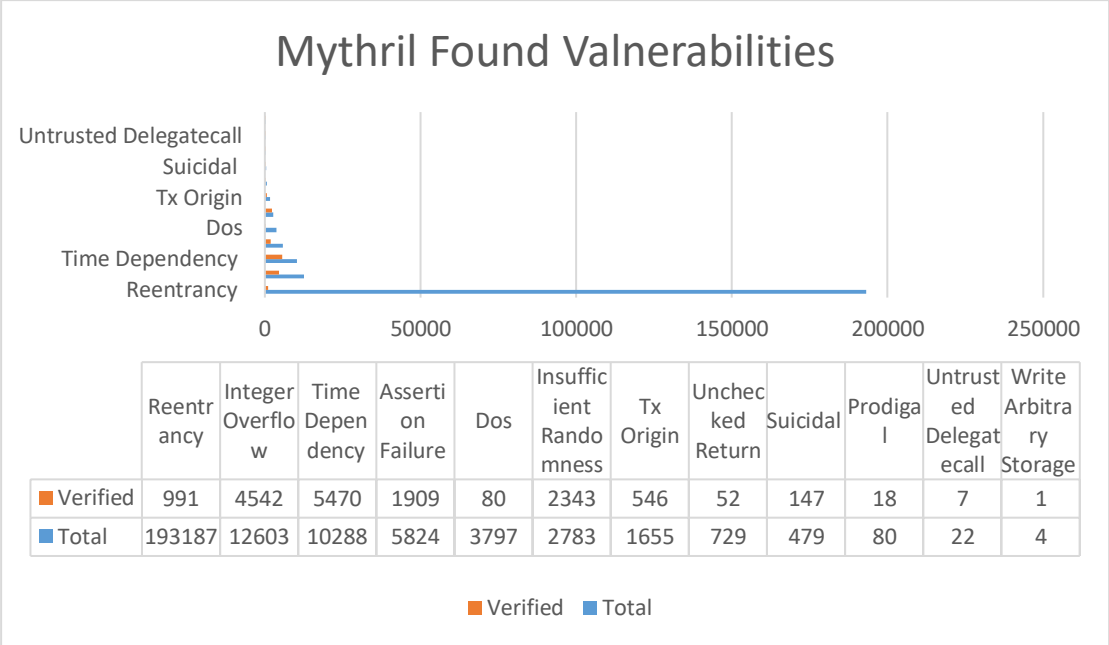


	Suicidal	Prodigal	Greedy
Verified	120	83156	7517
Vulnerable	946	175335	31737

Figure 21: Maian Found Vulnerabilities

Lastly, we will present the results for the Mythril tool. 193187 smart contracts were found vulnerable for reentrancy vulnerability, and 991 were verified. For integer

overflow vulnerability, 12603 smart contracts were found vulnerable, and 4542 were verified. 10288 smart contracts were found vulnerable to time dependency vulnerability, and 5470 were verified. 5824 smart contracts were found vulnerable for assertion failure vulnerability, 1909 of which are verified. 3797 smart contracts were found vulnerable for dos vulnerability, 80 of which are verified. 2783 smart contracts were found vulnerable to insufficient randomness vulnerability, and 2343 were verified. 1655 smart contracts were found vulnerable for the Tx origin vulnerability, and 546 were verified. 729 smart contracts were found vulnerable to unchecked return vulnerability, and 52 were verified. For suicidal vulnerability, 479 smart contracts were found vulnerable, 147 of which are verified. For prodigal vulnerability, 80 smart contracts were found vulnerable, 18 of which are verified. For untrusted delegatecall vulnerability, 22 smart contracts were found vulnerable, 7 of which are verified. Finally, four smart contracts were found vulnerable to the "writing to arbitrary storage" vulnerability, 1 of which is verified.



	Reentrancy	Integer Overflow	Time Dependency	Assertion Failure	Dos	Insufficient Randomness	Tx Origin	Unchecked Return	Suicidal	Prodigal	Untrusted Delegatecall	Write Arbitrary Storage
Verified	991	4542	5470	1909	80	2343	546	52	147	18	7	1
Total	193187	12603	10288	5824	3797	2783	1655	729	479	80	22	4

Figure 22: Mythril Found Vulnerabilities

Results Per Vulnerability

Following the initial per tool results, we will present the total number of smart contracts vulnerable for each detected vulnerability and how many were verified.

1. 369391 smart contracts were found vulnerable to reentrancy vulnerability, and 1225 were verified.

2. For prodigal vulnerability, 175414 smart contracts were found vulnerable, and 83174 were verified.
3. For callstack vulnerability, 131626 smart contracts were found vulnerable, and 74134 were verified.
4. 36304 smart contracts were found vulnerable to integer overflow vulnerability, and 13690 were verified.
5. For greedy vulnerability, 31737 smart contracts were found vulnerable, and 7517 were verified.
6. 16831 smart contracts were found vulnerable to truncation vulnerability, and 11401 were verified.
7. 10586 smart contracts were found vulnerable to time dependency vulnerability, and 7746 were verified.
8. 4380 smart contracts were found vulnerable to assertion failure vulnerability, and 1549 were verified.
9. 3130 smart contracts were found vulnerable for dos vulnerability, 69 of which are verified.
10. For integer underflow vulnerability, 2591 smart contracts were found vulnerable, 1795 of which are verified.
11. 1507 smart contracts were found vulnerable for insufficient randomness vulnerability, and 1201 were verified.
12. For suicidal vulnerability, 1372 smart contracts were found vulnerable, 231 of which are verified.
13. 1287 smart contracts were found vulnerable for signedness vulnerability, and 899 were verified.
14. 979 smart contracts were found vulnerable to money concurrency vulnerability, and 477 were verified.
15. 543 smart contracts were found vulnerable for unchecked return vulnerability, 51 of which are verified.
16. 444 smart contracts were found vulnerable for Tx origin vulnerability, and 194 were verified.
17. For untrusted delegatecall vulnerability, 21 smart contracts were found vulnerable, and six were verified.
18. Six smart contracts were found vulnerable for division vulnerability, 2 of which are verified.

19. For write-to arbitrary storage vulnerability, three smart contracts were found vulnerable, 1 of which is verified.
20. Two smart contracts were found vulnerable for modulo vulnerability, 0 of which are verified.

Figure 23 is a graphical representation of the top 5 vulnerabilities identified overall.

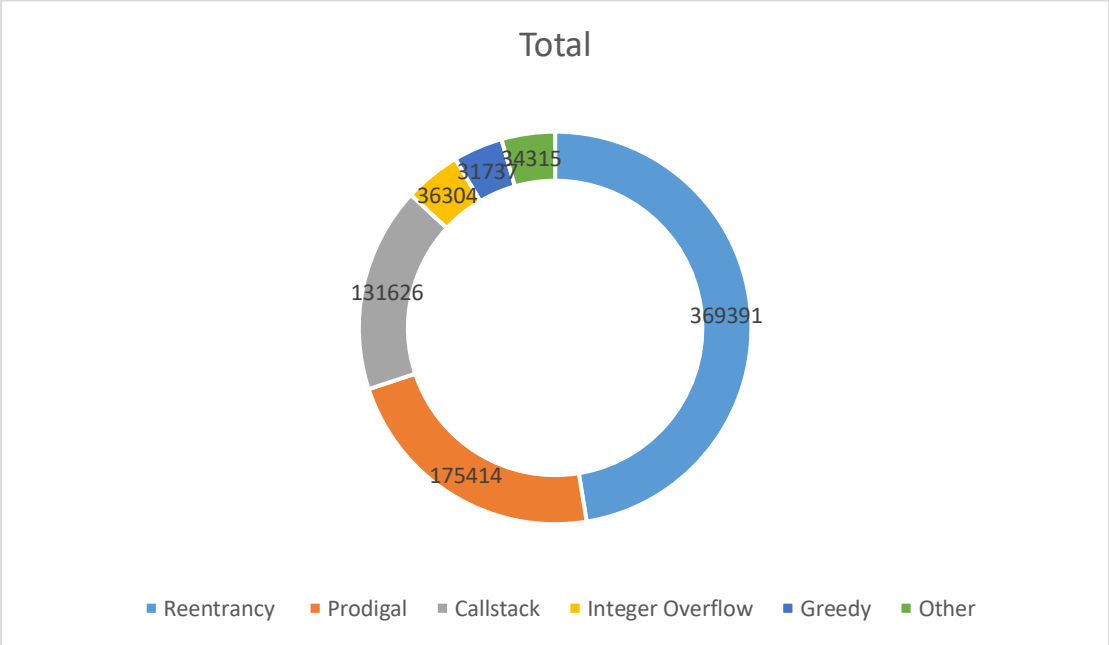


Figure 23: Top 5 Vulnerabilities Detected

Figure 24 is a graphical representation of the top 5 vulnerabilities identified for verified contracts.

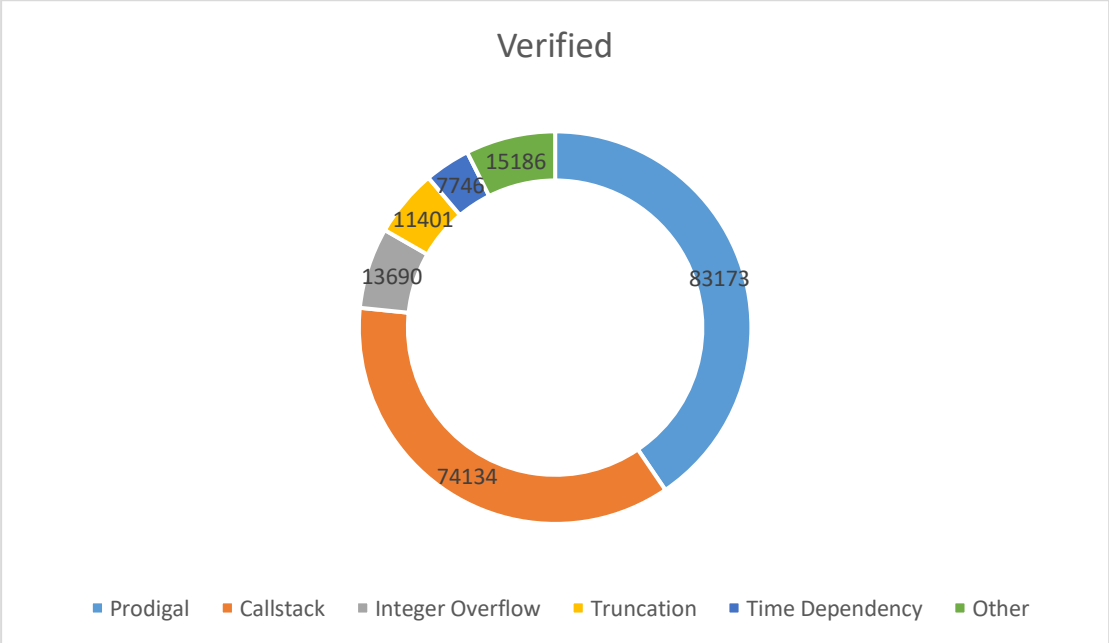


Figure 24: Top 5 Vulnerabilities detected for verified contracts

An important statistic is how many contracts have one vulnerability, two or more vulnerabilities.

1. A total of 683950 smart contracts have one vulnerability, of which 160796 are verified.
2. A total of 40487 smart contracts have two vulnerabilities, of which 19403 are verified.
3. A total of 4118 smart contracts have three vulnerabilities, of which 1826 are verified.
4. A total of 305 smart contracts have four vulnerabilities, of which 52 are verified.
5. A total of 35 smart contracts have five vulnerabilities, of which six are verified.
6. Nineteen smart contracts have six vulnerabilities, of which two are verified.

In figure 25, we can see the statistical data for the total contracts in a pie chart.

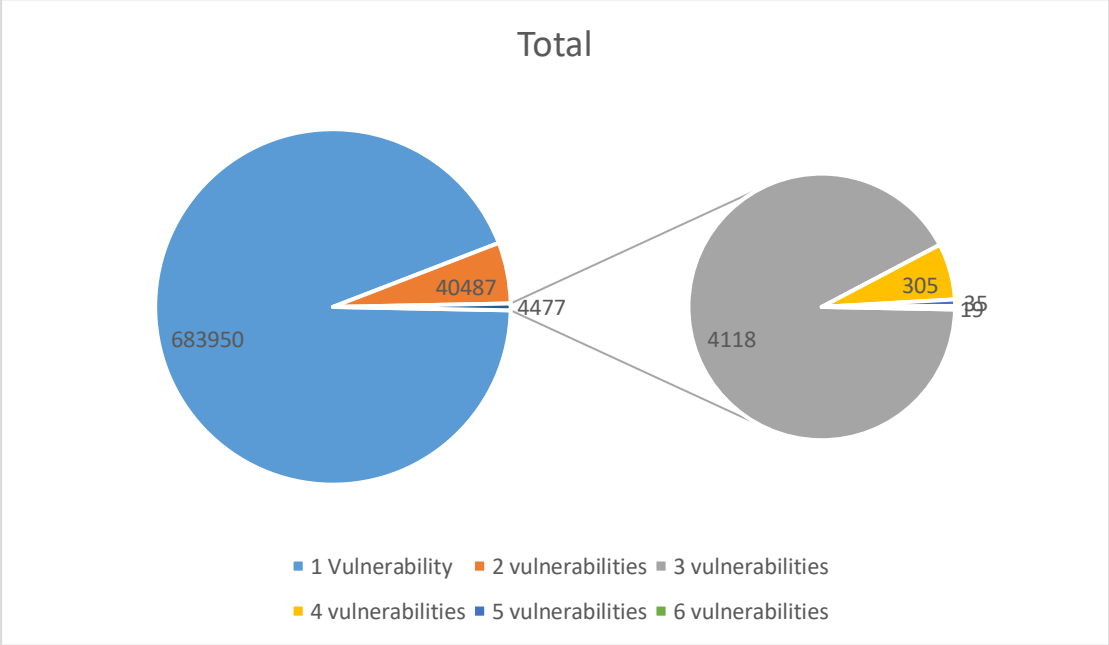


Figure 25: Number of Vulnerabilities found per contract

In Figure 26, we can see the statistical data for the verified contracts in a pie chart.

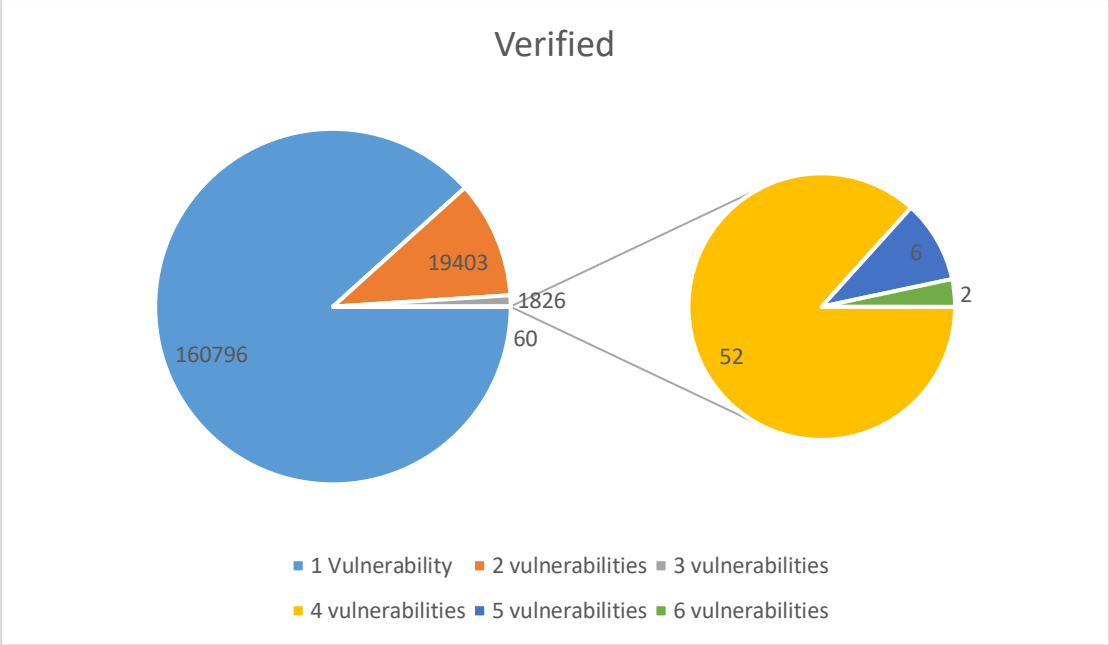
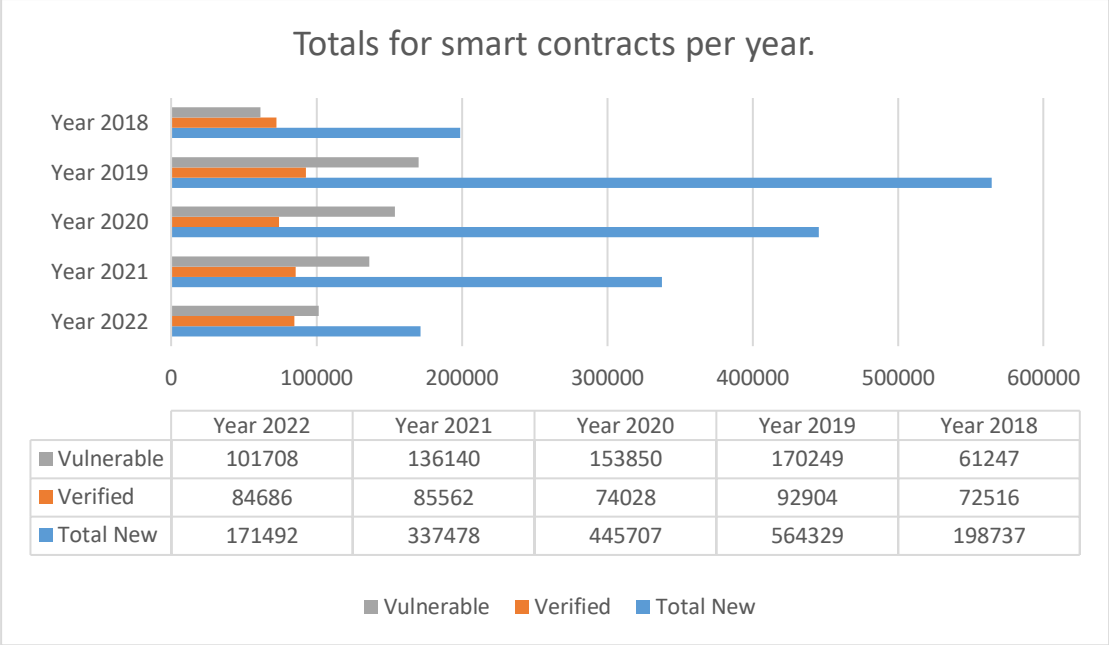


Figure 26: Number of Vulnerabilities found per Verified Contract

Vulnerable Contracts per Year

One of the most interesting pieces of information extracted from the results is the number of new vulnerable smart contracts created each year. For this reason, we will present the new vulnerable contracts created yearly since 2018, which was the first release of the solidity v0.4.22 compiler. For 2022, a total of 171492 smart contracts were created, 84686 of the created smart contracts have their code verified, and 101708 were found to have at least one vulnerability. For 2021, a total of 337478 smart contracts were created, 85562 of the created smart contracts have their code verified, and 136140 were found to have at least one vulnerability. For 2020, a total of 445707 smart contracts were created, 74028 of the created smart contracts have their code verified, and 153850 were found to have at least one vulnerability. For the year 2019, a total of 564329 smart contracts were created, 92904 of the created smart contracts have their code verified, and 170249 were found to have at least one vulnerability. For the year 2018, a total of 198737 smart contracts were created, 72516 of the created smart contracts have their code verified, and 61247 were found to have at least one vulnerability.



	Year 2022	Year 2021	Year 2020	Year 2019	Year 2018
Vulnerable	101708	136140	153850	170249	61247
Verified	84686	85562	74028	92904	72516
Total New	171492	337478	445707	564329	198737

Figure 27: Number of new / verified / vulnerable contracts per year

Another interesting metric to look at per year is the factor between vulnerable contracts and total created contracts. Also, another metric is the factor of the verified contracts and total created contracts.

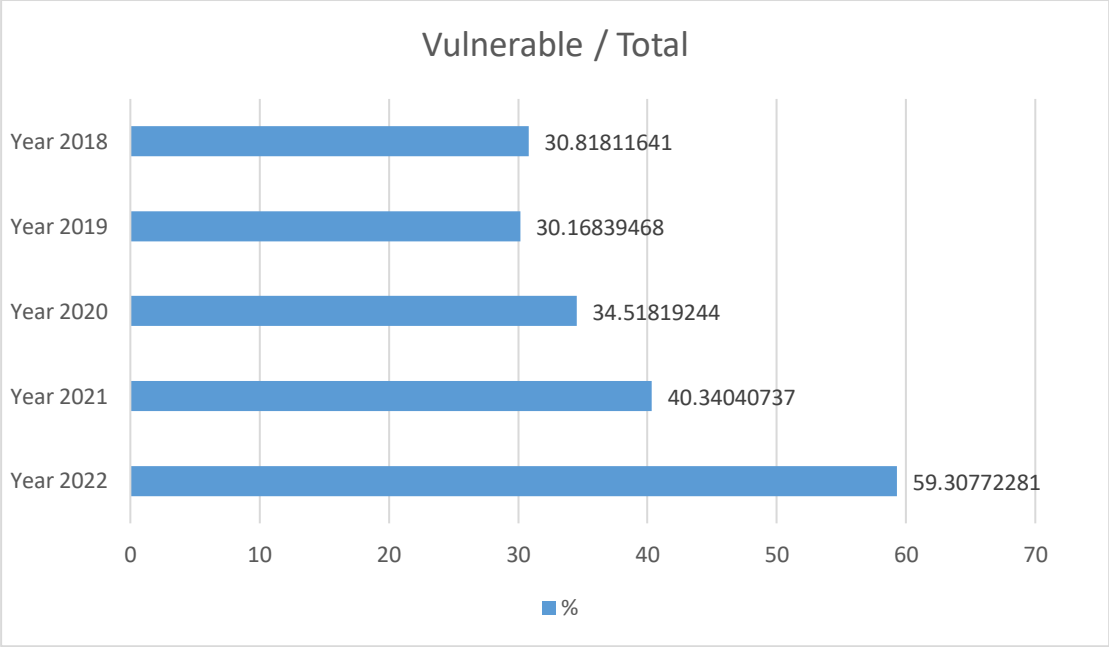


Figure 28: Percentage of vulnerable to new contracts per year

From figure 28 statistics, we can see that the percentage of vulnerable smart contracts vs. total smart contracts created is increasing year after year. Which might result from less importance on security, naïve developer practices, or other reasons.

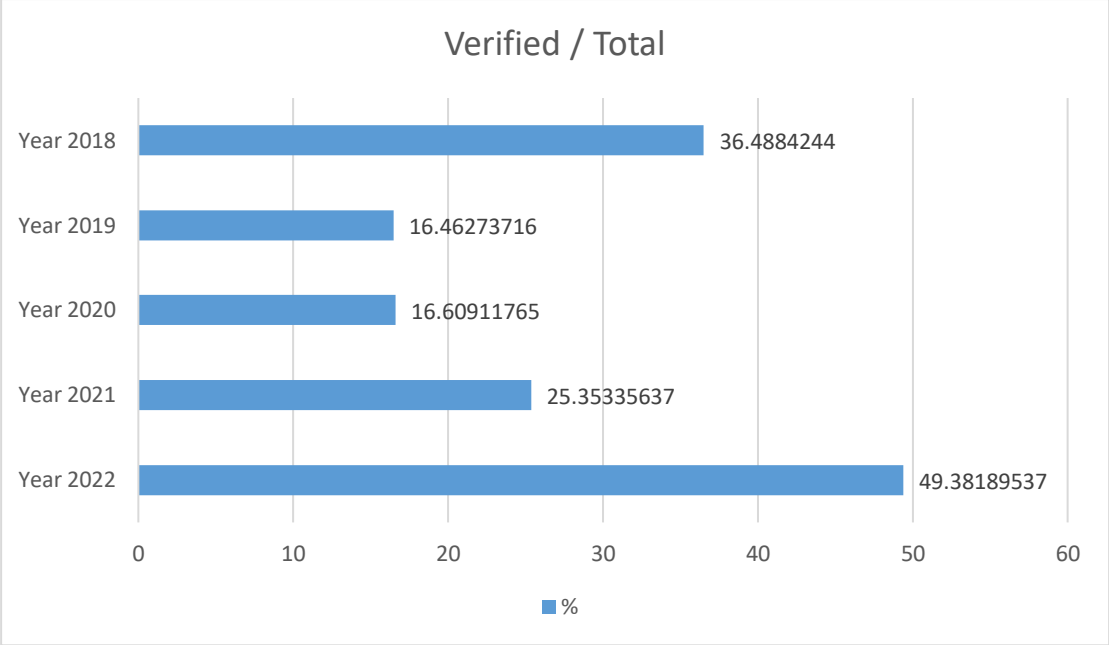


Figure 29: Percentage of verified new contracts per year

In figure 29 statistics, we can see that the percentage of verified smart contracts vs. total smart contracts created is increasing year after year. Which can be interpreted as a trend for more developers and projects going open source.

Conclusion

As seen in the previous chapters, security in Ethereum smart contracts is an ongoing pursuit with a small reach. Although some vulnerabilities were discovered some years ago, we can still see smart contracts containing them being deployed on the network. Because blockchain technology and the cryptocurrency industry are nascent, there is not as much attention and awareness of security and vulnerabilities. There is also a rush to develop new ideas and get products out faster. All the factors mentioned above help can explain the results presented in this research. There should be more focus on security. At least there should be an effort to avoid already known vulnerabilities and all solidity developers to become more aware of security best practices. We suggest communities create repositories containing vulnerable contracts and vulnerability assessment tools to check a smart contract on the go. These tools will help both users and investors. We also suggest that Layer 1 networks provide a stacking-like mechanism for smart contract creators and users, where it will be possible for a user to get reimbursed in case of fraud, which will dissuade vulnerable code from being reused again and again.

Finally, we will continue the research to expand to all smart contracts on the Ethereum network. Other research targets include assessing other EVM-compatible blockchains and creating an online free and up-to-date vulnerability assessment toolkit.

References

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system, 2008. Available online: https://bitcoin.org/bitcoin.pdf. (Accessed April 1st, 2022).

[2] Antonopoulos A.M. Mastering Bitcoin: Unlocking Digital Cryptocurrencies O'Reilly Media, Inc. (2014)

[3] V. Buterin, "A next-generation smart contract and decentralized application platform," white paper, 2014. Available online: https://github.com/ethereum/wiki/wiki/White-Paper (Accessed April 2nd, 2022).

[4] Zheng, Zibin & Xie, Shaoan & Dai, Hong-Ning & Chen, Xiangping & Wang, Huaimin. (2017). An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. 10.1109/BigDataCongress.2017.85.

[5] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in Proceedings of International Conference on Financial Cryptography and Data Security, Berlin, Heidelberg, 2014, pp. 436–454.

[6] NRI, "Survey on blockchain technologies and related services," Tech. Rep., 2015. [Online]. Available: https://www.meti.go.jp/english/press/2016/pdf/0531_01f.pdf

[7] D. Lee Kuo Chuen, Ed., Handbook of Digital Currency, 1st ed. Elsevier, 2015. [Online]. Available: http://EconPapers.repec.org/RePEc:eee:monogr:9780128021170

[8] V. Buterin, "On public and private blockchains," 2015. [Online]. Available: https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/

[9] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 4, no. 3, pp. 382–401, 1982.

[10] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger" Ethereum Project Yellow Paper, 2014.

[11] V. Zamfir, "Introducing Casper the friendly ghost," Ethereum Blog URL: https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost, 2015.

[12] Szabo, N. (1996). Smart contracts: building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought, 16, 18, p-2

[13] Jani, Shailak. (2020). Smart Contracts: Building Blocks for Digital Transformation. 10.13140/RG.2.2.33316.83847.

[14] Khan, S.N., Loukil, F., Ghedira-Guegan, C. et al. Blockchain smart contracts: Applications, challenges, and future trends. Peer-to-Peer Netw. Appl. 14, 2901–2925 (2021). https://doi.org/10.1007/s12083-021-01127-0

[15] Andrés, Javier & Lorca, Pedro. (2021). On the impact of smart contracts on auditing. The International Journal of Digital Accounting Research. 21. 155-181. 10.4192/1577-8517-v21_6.

[16] Rozario, A. M., & Thomas, C. (2019). Reengineering the audit with blockchain and smart contracts. Journal of Emerging Technologies in Accounting, 16(1), 21-35. https://doi.org/10.2308/jeta-52432

[17] Atzei, Nicola & Bartoletti, Massimo & Cimoli, Tiziana. (2017). A Survey of Attacks on Ethereum Smart Contracts (SoK). 164-186. 10.1007/978-3-662-54455-6_8.

[18] Zhou H, Milani Fard A, Makanju A. The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support. Journal of Cybersecurity and Privacy. 2022; 2(2):358-378. https://doi.org/10.3390/jcp2020019

[19] Chess, B., & McGraw, G. (2004). Static analysis for security. IEEE security & privacy, 2(6), 76-79.

[20] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 2019, pp. 8-15, DOI: 10.1109/WETSEB.2019.00008.

[21] Brent, L., Jurisevic, A., Kong, M., Liu, E.; Gauthier, F., Gramoli, V., Holz, R., Scholz, B. Vandal: A scalable security analysis framework for smart contracts. arXiv 2018, arXiv:1809.03981.

[22] Bayer, U., Moser, A., Kruegel, C., & Kirda, E. (2006). Dynamic analysis of malicious code. Journal in Computer Virology, 2(1), 67-77.

[23] Praitheeshan, P., Pan, L., Yu, J., Liu, J., & Doss, R. (2019). Security analysis methods on Ethereum smart contract vulnerabilities: a survey. arXiv preprint arXiv:1908.08605.

[24] MAIAN: an automatic tool for finding trace vulnerabilities in Ethereum smart contracts, https://github.com/MAIAN-tool/MAIAN

[25] Luu, L., Chu, D. H., Olickel, H., Saxena, P., & Hobor, A. (2016, October). Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (pp. 254-269).

[26] Melon Project, Oyente, 2016, https://github.com/melonproject/oyente.

[27] Torres, C. F., Schütte, J., & State, R. (2018, December). Osiris: Hunting for integer bugs in Ethereum smart contracts. In Proceedings of the 34th Annual Computer Security Applications Conference (pp. 664-676).

[28] ConsenSys, Mythril, 2017, https://github.com/ConsenSys/mythril-classic.

[29] Goel, S., Dhawan, M., Sharma, S., & Kalra, S. ZEUS: Analyzing Safety of Smart Contracts.

[30] Solidity version 0.4.21 documentation. Online at: https://docs.soliditylang.org/en/v0.4.21/miscellaneous.html. Retrieved November 28th, 2022.

[31] Solidity version 0.4.22 documentation. Online at: https://docs.soliditylang.org/en/v0.4.22/miscellaneous.html. Retrieved November 28th, 2022.

[32] Ethereum Byzantium hardfork details. Online at: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-609.md. Retrieved November 28th, 2022.

[33] Ethereum EIP-658 details. Online at: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-658.md. Retrieved November 28th, 2022.

[34] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[35] Nikolić, Ivica & Kolluri, Aashish & Sergey, Ilya & Saxena, Prateek & Hobor, Aquinas. (2018). Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. 653-663. 10.1145/3274694.3274743.

Table of Figures

Appendix A

Below is the complete code for the API on the virtual machine hosting the various vulnerability assessment tools.

```
<?php
$result = array();
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $bytecode = $_POST['code'];
        $verified = $_POST['verified'];
        $all = false;
        if (!empty($_POST["all"]) && $_POST["all"] === 'true') {
                $all = true;
        }
        $oyente = false;
        if (!empty($_POST["oyente"]) && $_POST["oyente"] === 'true') {
                $oyente = true;
        }
        $osiris = false;
        if (!empty($_POST["osiris"]) && $_POST["osiris"] === 'true') {
                $osiris = true;
        }
        $mythril = false;
        if (!empty($_POST["mythril"]) && $_POST["mythril"] === 'true') {
                $mythril = true;
        }
        $maian = false;
        if (!empty($_POST["maian"]) && $_POST["maian"] === 'true') {
                $maian = true;
        }
        if ($all === true || $oyente === true) {
                $output = shell_exec("sudo docker container exec oyente bash -c \"echo {$bytecode} >
/oyente/oyente/contract.sol\"");
                if ($verified == 'true') {
                $output = shell_exec("sudo docker container exec oyente bash -c \"python /oyente/oyente/oyente.py -s
/oyente/oyente/contract.sol -j -t 5000\"");
                } else {
                $output = shell_exec("sudo docker container exec oyente bash -c \"python /oyente/oyente/oyente.py -s
/oyente/oyente/contract.sol -b -j -t 5000\"");
                }
                $output = shell_exec("sudo docker container exec oyente bash -c \"cat
/oyente/oyente/contract.sol.json\"");
                shell_exec("sudo docker container exec oyente bash -c \"rm /oyente/oyente/contract.sol\"");
                shell_exec("sudo docker container exec oyente bash -c \"rm /oyente/oyente/contract.sol.json\"");
                $result['oyente'] = json_decode($output);
        }

        if ($all === true || $osiris === true) {
                $output = shell_exec("sudo docker container exec osiris bash -c \"echo {$bytecode} >
/root/osiris/contract.sol\"");
                if ($verified == 'true') {
                $output = shell_exec("sudo docker container exec osiris bash -c \"python /root/osiris/osiris.py -s
/root/osiris/contract.sol -j -glt 20\"");
                } else {
                $output = shell_exec("sudo docker container exec osiris bash -c \"python /root/osiris/osiris.py -s
/root/osiris/contract.sol -b -j -glt 20\"");
                }
                $output = shell_exec("sudo docker container exec osiris bash -c \"cat
/root/osiris/contract.json.evm.disasm\"");
                shell_exec("sudo docker container exec osiris bash -c \"rm /root/osiris/contract.*\"");
                $result['osiris'] = json_decode($output);
        }

        if ($all === true || $mythril === true) {
                $output = shell_exec("sudo docker run mythril/myth analyze -c '{$bytecode}' -o json --max-depth 10 --
parallel-solving --execution-timeout 15");
                $result['mythril'] = json_decode($output);
        }

        if ($all === true || $maian === true) {
                $result['maian'] = array();
                $output = shell_exec("sudo docker container exec maian bash -c \"echo {$bytecode} >
/MAIAN/tool/contract.sol\"");
                $result['maian']['suicidal'] = shell_exec("sudo docker container exec maian bash -c \"python3
/MAIAN/tool/maian.py -b /MAIAN/tool/contract.sol -c 0\"");
                $result['maian']['prodigal'] = shell_exec("sudo docker container exec maian bash -c \"python3
/MAIAN/tool/maian.py -b /MAIAN/tool/contract.sol -c 1\"");
                $result['maian']['greedy'] = shell_exec("sudo docker container exec maian bash -c \"python3
/MAIAN/tool/maian.py -b /MAIAN/tool/contract.sol -c 2\"");
                shell_exec("sudo docker container exec maian bash -c \"rm /MAIAN/tool/contract.sol\"");
        }

        echo json_encode($result);
} else {
 echo "No Post Request";
}
?>
```

Appendix B

Below there are various links to the GitHub repository of the project code, found smart contracts and vulnerability assessment results, and final statistic results.

GitHub:

https://github.com/stultusmundi/scat

Raw Results:

https://drive.google.com/drive/folders/1-CXmS0H_5GTEfE10olYexLfL90B9Iizv?usp=sharing

Vulnerability Assessment VM:

https://drive.google.com/drive/folders/1-CXmS0H_5GTEfE10olYexLfL90B9Iizv?usp=sharing

Final computed statistics:

https://drive.google.com/drive/folders/1-CXmS0H_5GTEfE10olYexLfL90B9Iizv?usp=sharing

Articles Mentioned:

https://drive.google.com/drive/folders/1-CXmS0H_5GTEfE10olYexLfL90B9Iizv?usp=sharing