UNIVERSITY OF TURKU

Comparison of Ethereum Smart Contract Vulnerability Detection Tools

Cyber Security

Master's Degree Programme in Information and Communication Technology

Department of Computing, Faculty of Technology

Master of Science in Technology Thesis

Author:

Binod Aryal

Supervisors:

Dr. Antti Hakkala

Dr. Seppo Virtanen

October 2021

Master of Science in Technology Thesis
Department of Computing, Faculty of Technology
University of Turku

Subject: Cyber Security
Programme: Master's Degree Programme in Information and Communication Technology
Author: Binod Aryal
Title: Comparison of Ethereum Smart Contract Vulnerability Detection Tools
Number of pages: 50 pages, 1 appendix pages
Date: October 2021

The thesis aims to reflect on the technical aspects of the Blockchain and Ethereum Smart Contract Vulnerabilities. The thesis has provided an in-depth overview of blockchain technologies, focused on Bitcoin, Ethereum needed to understand for vulnerabilities in blockchain. Application of cryptographic functions, consensus algorithm is explained and Blockchain security vulnerabilities are presented. A summary of public and private blockchains are shown, how these differ from each other and what are the use cases for these various blockchain application is provided. Ethereum Smart Contract are introduced and explained. The vulnerabilities present in smart contract are researched empirically.

The second half of thesis is focused on finding security flaws and vulnerabilities on Ethereum Smart contract. The attack vectors that are possible, cyber-attacks which has already happened and how can they be mitigated, if found, are presented. Analysis and comparison of popular Ethereum Smart Contract Vulnerability detection tools has been empirically studied using an automated tool called SmartBugs and the results are presented.

Keywords: Blockchain Security, Ethereum Smart Contract Vulnerabilities, Smart Contract Security tools

List of figures:

List of tables:

Acronyms:

BGP	Border Gateway Protocol
BTC	Bitcoin
DAO	Decentralized Autonomous Organization
dApps	Distributed Apps
DASP	Decentralized Application Project
DdOS	Distributed Denial of Service
DeFi	Decentralized Finance
DNS	Domain Name System
ECDSA	Elliptic Curve Digital Signature Algorithm
EVM	Ethereum Virtual Machine
FinTech	Financial Technology
HLF	HyperLedger Fabric
OWASP	Open Web Application Security Project
P2P	Peer to Peer
PoS	Proof Of Stake
PoW	Proof Of Work
RSA	Rivest–Shamir–Adleman
USD	US Dollar

Table of contents

1 Introduction

Ethereum is worlds' largest Smart Contract Blockchain Platform[1], second in the Market Capital[1]. Smart contracts hold Million Euros worth of Ether, a native cryptocurrency to Ethereum platform. Smart contracts can be executed by other smart contracts and are used for performing the transactions autonomously. The biggest security risk that Ethereum Smart Contract possess, is they are immutable by nature once deployed to the blockchain. They cannot be patched anymore, thus any vulnerabilities in the contract will cause a massive financial loss. Ethereum Smart contracts are written in a language called Solidity and compiled to bytecode that Ethereum Virtual Machine(EVM) executes[2][3]. The main objective of the research in this thesis is to figure out what smart contract vulnerabilities are, what are the causes of these vulnerabilities and what state-of-art tools exist to find and analyze these vulnerabilities.

The motivation of the research is to understand Smart Contract security which are used by Decentralized Autonomous Organizations (DAO), Decentralized financing (Defi) etc., that relies heavily, if not completely, on smart contracts of Ethereum. Thus, the author of such smart contract codes might find this research helpful to recognize and mitigate security vulnerabilities. On top of it, this research introduced the fundamental application of Cryptography in blockchain to establish a secured decentralized network. The security risks and flaws involved in blockchain has also been researched in this thesis. The research has also presented different blockchain technologies, primarily Bitcoin and Ethereum, and mentioned different cases of cyber-attacks made to the Blockchain providers or Blockchain system itself.

The researcher of the thesis, found out that the current trend of Blockchain solutions based on Ethereum and wide application of the Ethereum blockchain, has severe security implications [3]. The work of this research is based on earlier research trail of academics and scholars with the pointers they have provided for future work. The target audience for this research are security researchers, who would like to understand the security implications of Smart contracts and Smart Contract developers who would like to maintain security in the applications they developed based on Ethereum blockhain.

[1] According to coinmarketcap.com, a major website that shows daily trading volume of various cryptocurrencies.URL: https://coinmarketcap.com/, Accessed:29 September 2021

1.1 Research methodology and questions:

Two approaches have been taken for the research. One of the methods is literature and articles review related to the field of Blockchain Security and Smart Contract Vulnerabilities. From most of the articles reviewed and inspiration mostly from [3], [4] information has been gathered for further investigation on vulnerabilities of Smart Contract in Ethereum and the state-of-art tools that help to analyze these vulnerabilities. Second approach of experimentation on the analysis tools with carefully selected dataset that covers all the vulnerabilities of Ethereum Smart contracts, named as **Decentralized Application Security Project** or DASP10. Both methodologies applied in this thesis research aims at the following objectives:

1) To find vulnerabilities in blockchain and Ethereum Smart contract
2) To introduce the state-of-art of the tools available to analyze Smart Contract Vulnerabilities in Ethereum
3) To compare the tools and find out their similarities, differences, and the vulnerabilities they can detect

The thesis is structured in such a way that it will introduce all the necessary topics needed to dive into Ethereum Smart Contract Vulnerabilities which is the main topic of the research of this paper. **Chapter 2** introduces the blockchain, architecture of blockchain and cryptographic concepts used in blockchain. In **Chapter 3**, cryptocurrencies are explained with major focus on Bitcoin and Ethereum. **Chapter 3** also introduces Ethereum Virtual Machine (EVM), Solidity (A programming language used to write smart contracts) and Smart contract code is explained. **Chapter 4** delves with the summary of security issues and vulnerabilities of blockchain, their effects on Blockchain. **Chapter 4** also introduces security issues with Ethereum Smart contracts, the major focus of the research, and some of the notable attacks that have been conducted are presented. Smart Contract analysis tools are presented in **Chapter 5** and the custom data selection and criteria for tools selection are formulated. **Chapter 6** presents the results conducted using the tools and the dataset and discusses on the outcome of the results of the experiment. Finally, **Chapter 7** ends the paper with conclusion.

2 Basics of blockchain

This chapter will give the basics of blockchain technology which is needed to understand the Smart Contract vulnerabilities. Blockchain will be introduced and discussed with necessary core concepts for the fundamental understanding of the technology.

Blockchain, in the most- simplest form, is a ledger that contains all the data in form of blocks which will be then appended to the ledger forming an immutable chain of blocks, where every block has a reference to earlier blocks, so that editing any block might turn the whole blockchain to be invalid and needed to be re-computed. In the following sub-chapters, a brief overview of blockchain is provided. To avoid the confusion, cryptocurrencies such as Bitcoin and Ethereum are one application of blockchain rather than the whole concept of blockchain. Blockchain is not cryptocurrency rather the base foundational technological concept, that cryptocurrencies use to make a decentralized ledger-based currency system [5]. More in depth information related to cryptocurrencies are provided briefly in this chapter of the research.

Blockchain technologies also flirt with the idea of decentralization and distributed or peer to peer networks. Bitcoin has proven an example that a virtual digital currency, in colloquial terms cryptocurrency, can function without a need for a centralized authority. For that reason, blockchain provides a way for various parties involved in the network to validate data in blocks and maintain a network of trust that even if there are dishonest nodes and none of the network computers can be trusted, the blockchain can still be functional and immutable. Cryptographic applications in blockchain provide the opportunity to maintain the trust of the blockchain network even if the nodes are dishonest. Bitcoin, for example uses various concepts that can work as a payment protocol and even maintain anonymity of the transactions. Blockchains provide transparency and immutability, thus have many use cases including governance, smart grids, logistics, FinTech solutions, banking and security [6]. In the following chapters, a brief high-level overview is provided as an introductory information on blockchain.

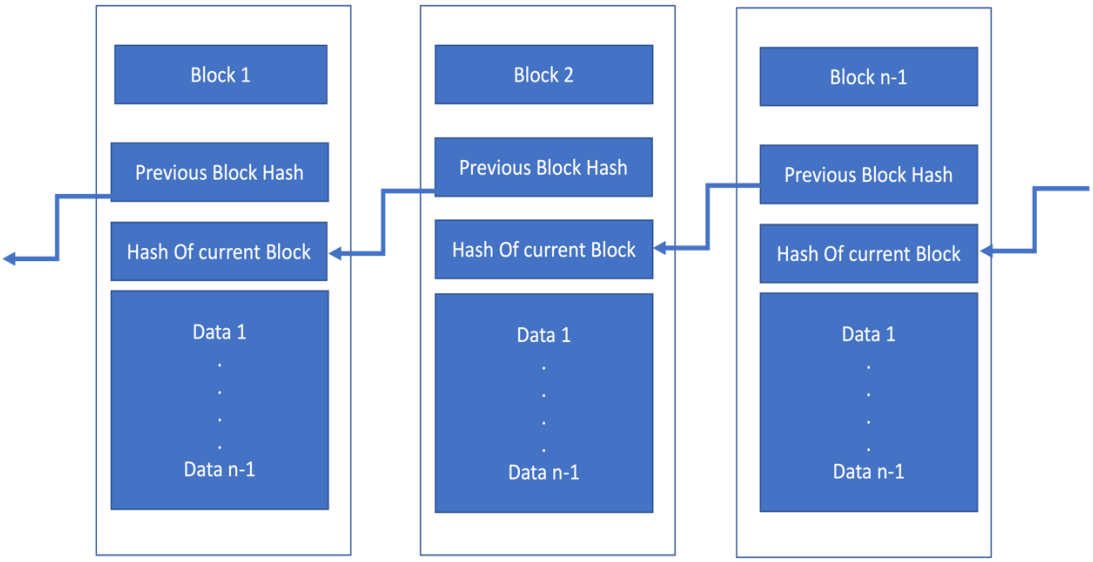


Figure 1-Blockchain

The concept of blockchain has come around into existence, around the time of rise of the internet in 1991 and the use cases of internet in every form human society from governance, banking, healthcare to content creation like media and social media. The core concept was developed from the idea of two researchers, who in their document proposed the following: "computationally practical procedures for digitally time-stamping documents so that it would be infeasible for a user either to back-date or forward-date the document"[7]. The technique was further developed to add multiple documents in form of blocks; hence the idea of block chain was formed. Chain meaning the individual blocks form a long chain each referencing to previous blocks forming an immutable chain. [8]

Bitcoin became the revolutionary concept that shaded a spotlight on the blockchain when an anonymous person, with the pseudonym of "Satoshi Nakamoto" created the idea of a digital currency that would incorporate blockchain at its core [9]. He wrote the white paper and then started the early version of bitcoin development in 2009. The idea that Bitcoin tried to achieve was to create a peer-to-peer(P2P) decentralized digital currency and have a large pool of anonymous user validate transactions called as miners on a network that can have dishonest nodes by using cryptography and other algorithms. Bitcoin became a sensational technology and inspired wide range of possibilities on decentralization and maintaining security and integrity while the member nodes of the network cannot be trusted.[9]

Ethereum came up with even better idea on the decentralization, revolving around the use of smart contact, DAOs and Turing complete blockchain programming. Ethereum's idea was created by Vitalik Buterin in 2013, which he wrote on White Paper and first released in 2015. Ethereum made development of many new concepts on Decentralized Financing (DeFi), and many cryptocurrencies (colloquial term for blockchain based digital currencies). Cryptocurrency trading started, and there are circa 1000 plus cryptocurrencies being traded with Fiat currency nowadays [2]. Hyperledger Fabric (HLF) was introduced as permissioned private blockchain solution under Linux Foundation and backed by IBM and other huge Tech Companies in December 2015. Hyperledger fabric provided industrial grade blockchain framework, that is programmable in most popular programming languages in the current Information Technology world.[10]

2.1 Blockchain Architecture

A decentralized blockchain in generally found in the following architecture:

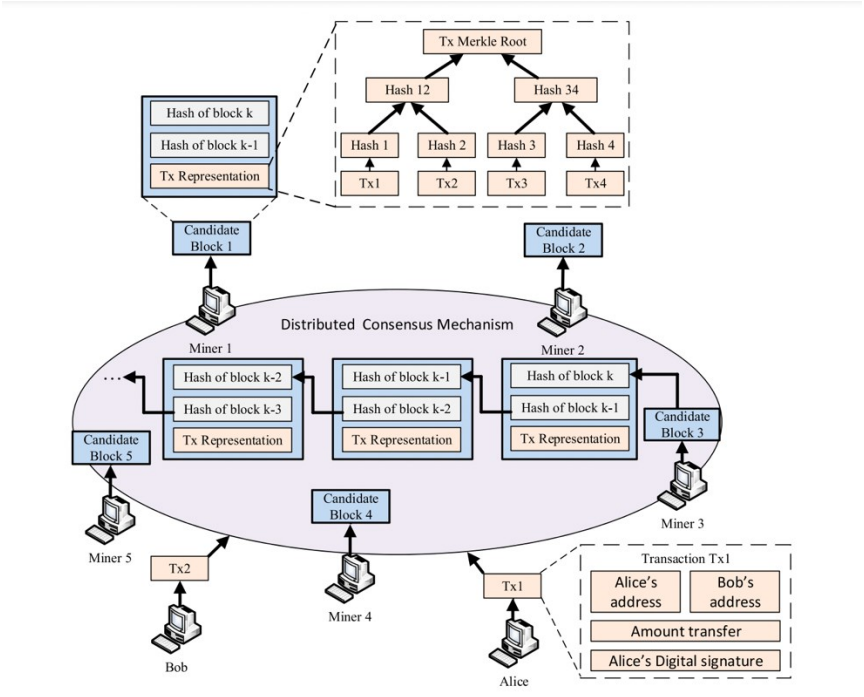


Figure 2- Blockchain Architecture [9]

In the figure 2 above, various nodes called as miners in the Blockchain terminology try to validate a block of transaction with a consensus algorithm. The transaction to be validated are coded into the

consensus algorithm and mostly the miners must solve an extremely difficult, time and energy consuming hashing algorithm to validate a block. Once the correct hash is found then the block is considered as mined and appended to the public ledger. Different Blockchain offer different incentive model as reward for successfully mined block. This reward is the main source of motivation for individuals or companies to invest on expensive computers and hardware based hashing miners etc. The complexity of the mining algorithm is also used as a method of deterrence for trying to attack the blockchain or try to be dishonest because the gain received from investment of resources in mining hardware outperforms the necessity to attack or be dishonest[2], [9].

2.2 Underlying cryptography concepts of blockchains

This part focuses on the various implications of Security, including the challenges, problems, attack vectors, vulnerabilities, and mitigation. Blockchain heavily relies on Cryptography for maintaining security, anonymity, data integrity and uses of networking protocols for creating decentralized network. In the following chapters to come, the core cryptographic functions, consensus algorithm, decentralized networking will be mentioned. They are the building blocks of the secured blockchain, and which provides the core idea for Information security. Blockchain uses various complex cryptographic functions and consensus algorithms to maintain the system and services. These cryptographic functions are used for various purposes and can be categorized to the following manner:

2.2.1 Hashed chain storage

A cryptographic hash function is a non-reversible mathematical function, which means input will always give a fixed length output, but input cannot be reversed from output. Hash functions take arbitrary size of data in chunks and return a fixed length output in a deterministic pattern. Hashes are used in every field of computer science such as, Information Security and database storages to name a few. SHA256 is one of the most used hashing algorithm in computer science. Most blockchains solutions use Cryptographic hash functions to get the hash value of the block to be appended to the Ledger and use hash pointer to provide reference to previous block, like linked-list data structure of hashes. Hashing is at the core of blockchain ledger [11].

2.2.2 Merkle Tree

An inversed binary tree structure of hashes that are generated from hashing the block and each hash pointing to the predecessor. Merkle Tree is generated by Double SHA (SHA256 hashing algorithm) hashing technique of blocks. The pseudo-code of Merkle tree hashing algorithm can be represented as:

Merkle Root = SHA256.digest(SHA256.digest(BLOCK))

The output of the algorithm results as Merkle root hash of a block and they are referenced in the next block, making a chain of binary tree like data structure which is referred to as Merkle tree.

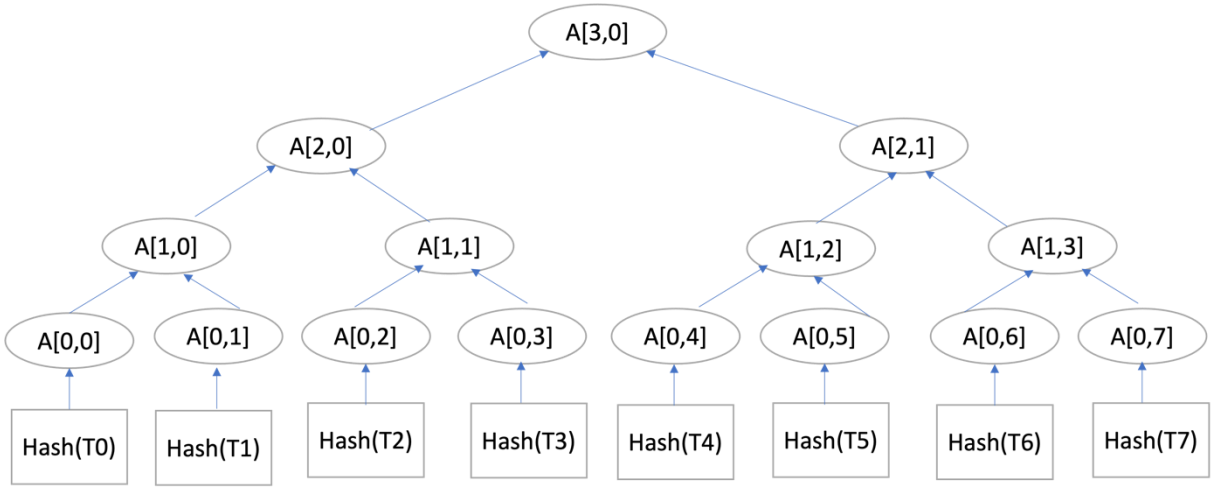


Figure 3--Merkle Tree of Leaves[10]

Here each child nodes are hashed together until one Merkle root hash is found. The parent node's Merkle hash is always a concatenated hash of its child nodes. If the data is changed in one of the child nodes, the root hash will be rendered invalid. This technic ensures the integrity of the data. Merkle tree is the root concept used in Bitcoin and slightly modified version is used in Ethereum called as Patricia Merkle tree [12], but fundamentally it represents as inverted binary tree like structure where each block in the child node is hashed to produce a unique root hash and continues to grow upward.[13]

2.2.3 Digital Signatures

Digital signature use public-key cryptography to generate keys, to ensure the validity of the signature and the data encryption/decryption. Public key cryptography works by producing a private key and public key pair, and the data signed by using a persons' private key is always checked against the public key. The public key cryptography algorithm in basic is as follows:

Encrypted Message = EncryptionFunction(Message, Public Key of recipient)
Decrypted Message = DecryptionFunction(Encrypted Message, Private Key of reciever)

Common Digital signature algorithm is RSA. RSA for example uses same function to encrypt and decrypt messages and the syntax is as follows:

Message = EncryptDecryptFunction(Data, Public Key/Private Key)

The data that has been issued using public key of the person can only be decrypted by the respective private key only. This will ensure that, in fact(mathematically), the issuer of the digital signature is the holder of the private key. Digital signatures are also used to create the wallet addresses for cryptocurrencies, thus the coins transferred to a persons' public address is only accessible by their private key and when transferring the coin, only the holder of the private key can transfer the coins. Bitcoin and ETH use Elliptic Curve Digital Signature Algorithm (ECDSA) for digital signature [14].

2.2.4 Consensus algorithm

Since blockchain is revolving around the idea of decentralization, consensus algorithm is used to maintain the trust in an environment where other nodes cannot be trusted and maintain the security, validation, and integrity of the data in the blocks. Bitcoin uses Proof of Work (PoW) and Ethereum uses Proof of Stake (PoS) as consensus algorithm. PoW uses computational resources as proof while PoS is based on the idea of staking certain Ether. Both, algorithms are designed to achieve trust in the decentralization network and reach common consensus between nodes about which blocks, and transactions are valid and what they should be based on. The upcoming Chapter 3 will provide, consensus mechanism used in BTC and ETH in details . [14]

3 Cryptocurrencies (Bitcoin, Ethereum)

In this chapter, two of the most popular cryptocurrencies are introduced. Bitcoin and Ethereum are two widely popular blockchain based cryptocurrencies. This chapter will provide information for these two cryptocurrencies, and their fundamental differences and working principles.

3.1 Bitcoin

Bitcoin was introduced by Satoshi Nakamoto, a pseudonymous character, who developed the early version of Bitcoin. He is yet unknown to the world. Bitcoin became sensational technological concept since the start.

The key problems that bitcoin was trying to solve is listed below:
1. Solves double spending problem
2. Decentralization of nodes
3. Nodes enter and exit at will, accept the latest ledger
4. Transactions are timestamped by timestamp server in chronological order
5. Bitcoin is chain of digital signatures [11]

Transaction is at the core of the blockchain, and each transaction are recorded into a block and signed with the private key of the sender and public key of the next owner. The core of the transaction as mentioned in Bitcoin Whitepaper, which describes, a cryptocurrency or a digital coin as chain of blocks of digitally signed transaction that are hashed using cryptographic hashing algorithms. The chain will be on a decentralized network, where the peers can join and leave at will and a copy public ledger will be maintained by each peer and the transactions can be verified with the digital signatures, through a process called mining. The process of mining refers to finding a computationally difficult cryptographic hash and once it is found the block is considered mined and for the miner of the transaction certain reward in form of Bitcoin (BTC) is provided [9].

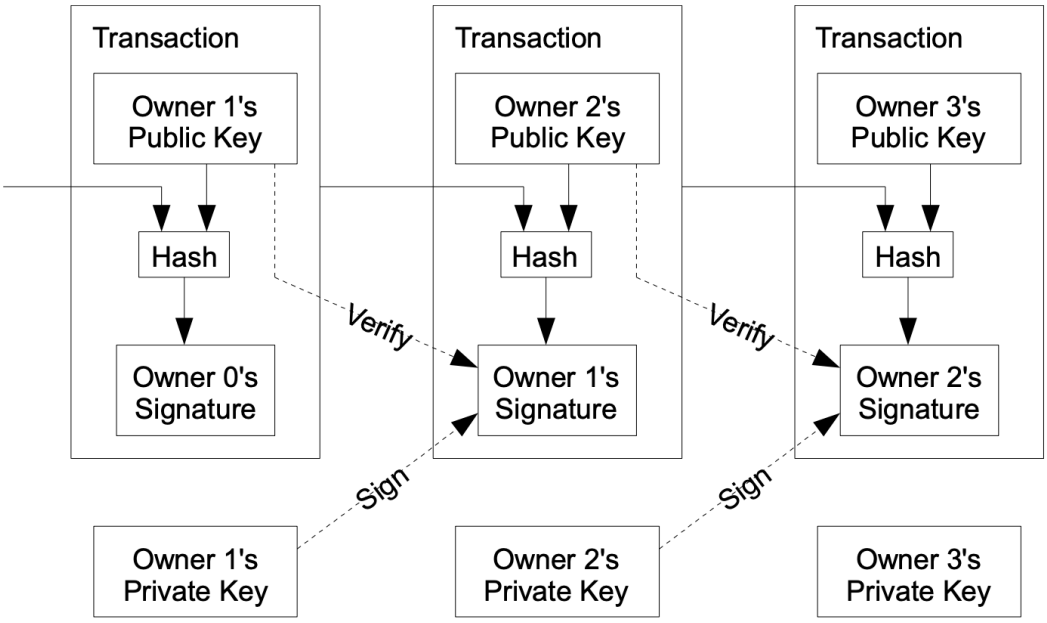


Figure 4--Bitcoin Transactions which makes the blockchain[7]

In the above figure 4, each block represents transaction that will be hashed using hashing algorithm, which is mentioned later in this document, to produce a unique hash. Each of these transactions are linked to previous transaction, thus the name blockchain. For a transaction to be valid all the previous transactions need to be valid, thus providing the integrity of the data. The aspect of security is achieved in such way that all peers maintain a public ledger which make any peer trying to modify the ledger corrupts the whole chain.

Bitcoin uses Proof of Work (PoW) as consensus algorithm. It is based on Byzantine Fault Tolerance algorithm. PoW is based on finding a hash value for a block, through a process of mining and the complexity of the hash target value increases over time[14]. This time-consuming hashing and computationally expensive mechanism is used to make sure that the blocks are mined honestly and the incentive is paid for computational power used by the node. This concept is also used to control majority attack, since it requires huge computational resource to modify the ledger and the incentive for mining is bigger than using resources to control the ledger. However, PoW is computationally very expensive and requires expensive hardware setup as well as electricity costs for mining the blocks. Blocks are mined roughly every 10 minutes, difficulty increases over time and miners are paid in Bitcoins for successfully mining a block[9][11].

3.2 Ethereum

Vitalik Buterin saw problems in Bitcoin, mostly energy efficiency and lack of Turing completeness in the transaction code, and decided to start Ethereum project to solve the computational cost by using Proof of Stake (PoS) algorithm which is computationally less expensive. Proof of Stake is discussed more in the consensus algorithm in chapter 2.2 [2]. Below are the Key Concepts from Ethereum that was unique and not available in Bitcoin:

1. Smart contracts
2. Decentralized autonomous organizations (DAOs)
3. Turing complete programming machine for blockchain.

Ethereum tried to solve the existing problem in Bitcoin to introduce a fully programmable decentralized blockchain system. Ethereum to be energy efficient, uses Proof of Stake (POS) as the consensus algorithm for the network. The idea requires the user or the nodes to stake ETH(30 ETH to become a validator) [15] and they can work as a validator, which are essentially like miners for Ethereum Network. PoS tries to address the accessibility, centralization, and scalability of the system unlike PoW which aims for complete decentralization and also ensures less computational resource requirements. PoS also penalizes the validator in case they try to act maliciously, thus motivating the validating nodes for being good actor in the network. PoS also tries to prevent Majority attack, because the amount of Stake's value needed is not worth to try to participate in such attacks [2][11].

3.3 Smart contracts with Ethereum

Smart contract, in very basic form, is a contract that exist in form of code and the contract is enforced through programming. The idea of smart contract has existed since 1991 [7][8]. Ethereum took the concept of blockchain and added a Turing complete arbitrary piece of code, called smart contract, which once uploaded to the blockchain will always be on the blockchain, and the code are immutable and continue to exist on the chain for the lifetime of blockchain. Ethereum Virtual Machine (EVM) will execute transaction based on the smart contract and the smart contract code is written mostly in Solidity (JavaScript like) or Vyper (Python like). Major number of smart contracts are written in solidity, a high-level programming language. The smart contract will be converted to Bytecodes and executed by EVM.[2][3]

Since the development of smart contract on Blockchain, Ethereum has become the pioneer, as well as backbone for many cryptocurrencies and distributed apps (dApps). The idea of smart contracts has shown a new power of blockchain, unlike Bitcoin, which only stores value, Ethereum proved that there can be transactions as well as data that could reside on the blockchain and creating a global supercomputer called EVM that can perform transactional and data computation [2]. This idea added a new development of arbitrary distributed application that can perform data computation and transactional changes autonomously. EVM can be thought as one singleton computer made of decentralized nodes that can execute arbitrary codes that are Turing complete and can be added to the blockchain and perform transactions in an automated fashion without needing any interaction from centralized authority or humans.

For this reason of immutability and fairly novel concept, smart contracts pose security threat. A buggy smart contract for reason of immutability and lifetime existence cannot be patched, which is a majority security hazard. If vulnerabilities are found, the smart contract code can be exploited to cause various kind of attacks, mostly leading to financial losses. The immutability of blockchain comes with a risk that, a smart contract even if there is bug or vulnerability, once uploaded to the network, can never be patched and reside the lifetime of blockchain. This issue leads to, and has led to, financial losses. Other blockchain network that has come up with smart contracts are EOS, Hyperledger Fabric, etc. Smart contracts have use cases in multiple areas like electronic voting, identity management, supply chain management, logistics, IoT, property management, banking and financial services, etc. [16][3]

3.3.1 EVM

EVM(Ethereum Virtual Machine), just like Java Virtual Machine, executes the bytecodes compiled from the Smart Contract code written in languages like Solidity, Vyper [3]. EVM is comprised of 140 opcodes, uses 256-bit registers that can hold 1024 items. EVM limits the call stack to 1024 EVM words (256 bits word) on the virtual machine. The stack is assigned for call return addresses, and variable assignments. EVM while executing the transactions assigns and maintains a transient memory which is a EVM word-addressed byte array of 256bits up to the maximum call stack(1024 EVM words) until there will be stack overflow[12]. Every EVM execution costs GAS, which is a term used in Ethereum, which is paid by the caller of the contract or transaction. GAS is paid in Wei, which is the smallest denomination of Ether (1 ETH = 10^{18}wei).[3]

3.3.2 Solidity

Solidity is the de-facto language for writing smart contract. Solidity resembles JavaScript or C family language and is compiled to EVM compatible bytecode for execution. Below is the picture of solidity produced bytecode and instruction set for EVM from solidity code:

```
contract Factorial {
  function fact(uint x) returns (uint y) {
    if (x == 0) {
      return 1;
    } else {
      return fact(x-1) * x;
    }
  }
}
```

Figure 5-Solidity Code[15]

```
60606040526000357
c0100000000000000
00000000000000000
00000000000000000
00000000900480631
93ddd2c1460375760
35565b005b6042600
4805050605a565b60
40518082151581526
02001915050604051
80910390f35b60006
00560006000505414
9050606b565b9056
```

```
0x00 PUSH1  0x60
0x02 PUSH1  0x40
0x04 MSTORE
0x05 PUSH1  0xe0
0x07 PUSH1  0x02
0x09 EXP
0x0a PUSH1  0x00
0x0c CALLDATALOAD
0x0d DIV
0x0e PUSH4  0x193ddd2c
0x13 DUP2
0x14 EQ
0x15 PUSH1  0x1a
0x17 JUMPI
```

```
0x18 JUMPDEST
0x19 STOP
0x1a JUMPDEST
0x1b PUSH1  0x00
0x1d SLOAD
0x1e PUSH1  0x05
0x20 EQ
0x21 PUSH1  0x60
0x23 SWAP1
0x24 DUP2
0x25 MSTORE
0x26 PUSH1  0x20
0x28 SWAP1
0x29 RETURN
```

Figure 6- Solidity(solc files) compiled to EVM byte-code[15]

In the figure 5, a contract that is written in Solidity is compiled to EVM bytecode shown in left of the figure 6. The first box of left hand is EVM byte code and two right hand boxes of figure 6 shows the instruction set that is executed by EVM. EVM is stack-based. Bytecodes contains the instruction set or OPCODES for EVM to perform the execution. EVM bytecodes are executed on stack based operations similar to Java Virtual Machine(JVM). [17] points out that, a Low Level EVM bytecode has no notion of stack frames, no type checking, no abstractions for methods or classes in comparison to JVM. Furthermore, the research also suggests that EVM is purposed for doing simple computation for efficient storage of the Smart contracts on the Blockchain of Ethereum.[17]

3.3.3 Smart Contract Code

Given below in figure 7 is an example of smart contract code written in solidity.

```
1  contract AWallet{
2      address owner;
3      mapping (address => uint) public outflow;
4
5      function AWallet(){ owner = msg.sender; }
6
7      function pay(uint amount, address recipient) returns (bool){
8          if (msg.sender != owner ||  msg.value != 0) throw;
9          if (amount > this.balance) return false;
10         outflow[recipient] += amount;
11         if (!recipient.send(amount)) throw;
12         return true;
13     }
14 }
```

Figure 7-Smart contract written in Solidity [3]

Line 1 of the contract defines the contract name "AWallet". Line 5 is the constructor of the contract. According to [3] the constructor is only called once at the time of the contract creation. The "address" seen in the contact is the variable that holds the addresses. Addresses are 160 bits and used to uniquely identify each contract in the blockchain. The "outflow" seen in the line the line 10 is a hash table that keeps track of the all the address the contract sends money to. The "function pay" at line 7 is used for sending money by the owner of the contract. Also, all the Ether that are send to this contract address is held by this contract and the amount is recorded in the variable "balance" as seen in line 8. Variable "balance" cannot be influenced by the programmer rather used to track the amount of Ether particular contracts hold at the time. At line 8 the function throws an exception with "throw" keyword if the sender is not the owner, or some ether is not transferred to the contract. At line 9 the call returns if the amount is greater than the balance the contract holds. Line 9 updates the hash table with the recipient's address to the amount being transacted. At line 11 the "send" method can fail the recipient is a contract [3][17]. The smart contract written in Solidity are comprised of following declarations:

a) Status Variable: These values get stored in the contract permanently
b) Functions: The executable part of the contract
c) Function Modifiers: Used to modify and add functionality to the contracts
d) Events: A communication interface to EVM that can notify certain events have occurred, for debugging and alerts.
e) Structures: custom data structures to group multiple data types together
f) Enumerations: Used to define custom data types and enumerate them

4 Security issues and vulnerabilities in blockchain

In this chapter, vulnerabilities that exists in Blockchain technology will be presented. These vulnerabilities are generic vulnerabilities rather than smart contract specific. For understanding vulnerabilities of smart contracts, it is necessary to understand the vulnerability and security issues of the blockchain.

Blockchain has implemented many techniques to curb the security vulnerabilities. A blockchain has few requirements to maintain the security and privacy of the system. In this chapter, a thorough analysis of Blockchain system attack vectors, Security requirements, Mitigation technic if present, future threats and most notable cyber-attacks will be given in a high level details. In the most fundamental level, a blockchain must provide following requirements to assert and secure the system:

Consistency of the ledger: Since blockchain is based on transparent public ledger system for distributed network, it must provide the consistency of the ledger across all the nodes.

Integrity and Immutability of the data: Blockchain might be running financial or other highly valuable data, so ensuring the integrity of data and immutability of the chain must be considered.

Availability of the service and data: The network must maintain the availability of service, since it is a decentralized service, the availability of the network as well as the data should be considered. If financial transactions are to be made, the network must maintain the uptime availability and processing of the transaction.

Confidentiality and anonymity: Blockchain solutions must provide confidentiality of the users, since the network and ledger are peer to peer. Some solutions focus on anonymity or pseudo-anonymity. Bitcoin ensures this by only providing a cryptographic hash of the public key of the user thus maintaining the anonymity while the trust was maintained by usage of cryptography.

4.1 Major attack vector on the Blockchain

Various Attack vectors exist in Blockchain. Some of them are directly related to other attack vectors that are common in any networked application. There are distinct attack vectors that only relates to blockchain. In the following sections, all the major attack vectors are presented:

4.1.1 Consensus attack (51% Majority attack):

This attack vector is one of the most common attack in the blockchain or distributed networked systems. As the name suggest, this kind of attacks happen, when a malicious party tries to own more than 51% of the computing resources of the network and thus trying to influence the ledger or then hijack the consensus on their favor. For this reason, incentive models in form of BTC or ETH is provided to make sure that the motivation for reward is so high that a user would not rather try to spend resources on trying to own an influence on the network. Here is an excerpt from Bitcoin Whitepaper: "The system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes." [8] The attacker who is attacking with this sort of vector, can double spend coins as well as stop confirmation of transactions but are not able to mine new coins or reverse confirmed transaction or false creation of transactions. In August 2016, Two Ethereum based blockchain solutions, Krypton and shift suffered such attack. Another great example of the 51% attack was against Bitcoin Gold in May 2018, a fork of Bitcoin, where malicious actors controlled the majority of network and performed double spending attack for several days stealing $18 million worth of Bitcoin Gold. [9][18]

4.1.2 Fork Problems

Fork in blockchain happens when two different chains of different blocks are formed while mining the blocks. This can happen intentionally or then deliberately orchestrated. To mitigate the forking issues, the consensus algorithm is designed to accept the longest valid chain, which means that the longest of the chain with valid blocks will be considered as valid and added to the ledger.

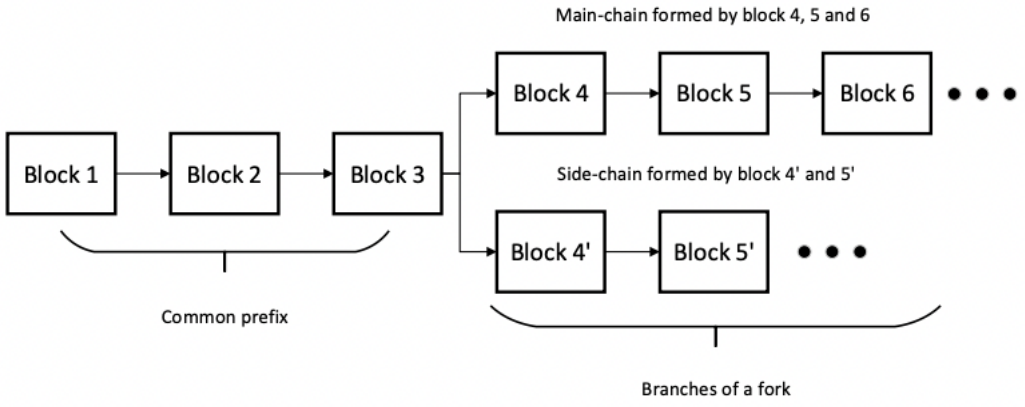


Figure 8-Blockchain Fork structure [19]

A forking problem, such as shown in the diagram above in figure 8, where the consensus is not achieved within the participating nodes, where one group of participants will accept one of the ledgers as updated version while the other group will fork a new transactional ledger both pointing to block 3 as the parent node in the above picture. This leads to existence to two ledgers independent of each other that has branched out from Block 3. Some example of fork that happened are Bitcoin and Bitcoin Cash, Ethereum and Ethereum Classic.[18]

4.1.3 Double spending attack

A double spending attack happens when a user tries to pay two entities with the same amount and hoping that both transactions will get succeeded and added to the chain. This problem can also exist if two blocks are mined at the same time. Most of the double spending attacks are possible if an entity owns 51% or more of the networks computational power and thus validating both transactions and appending in the ledger[20]. [9] mentioned in the white paper about the mitigation of double spending attack by providing a computationally heavy Proof of Work (PoW) algorithm which will validate the transaction against a public ledger that will be shared among of all of the participants in the node and only the first transaction is accepted as valid. One great example of double spending attack was done on Ethereum Classic in August 2020, a majority attack and $5.8 Million double spent.[21]

4.1.4 Network attacks

A decentralized system like blockchain is very vulnerable to network attack. Some of the network attack vectors are briefly discussed below:

DNS attack: A DNS is used as a mechanism to discover active peers in the network. When a node joins the network, it queries DNS and it gets a list of records of the IP addresses of the peers. A DNS, according to bitcoin developer guide opens wide array of attack possibilities. These attacks can range from, man in the middle attacks, stale records, cache poisoning etc.[18]

BGP hijack and spatial partitioning: Blockchain are made up of two types of nodes. Full nodes, who are the actual participant of the network, and they maintain the blockchain, validation and updates. Whereas lightweight nodes are not involved in maintaining the blockchain and they rely on full nodes to join the network. If these nodes are spatially located on the same Autonomous Systems owned by ISPs, a BGP hijacking attack is possible. [22] noticed that by hijacking fewer than 100 Border Gateway Protocol (BGP) prefixes in Bitcoin, an attacker can isolate up to 50% of the network's hash rate.

BGP hijack example case: A malicious ISP announced BGP prefixes belonging to major ISPs including Amazon, OVH, Digital Ocean, etc. and since the Bitcoin Network data is plaintext, the attacker was able to intercept the traffic routed to mining pools and was able to make 83,000 USD. Another case includes, BGP attacks launched against MyEtherWallet.com in April 2018, and the attacker were able to make 152,000 USD.[23]

Eclipse Attack: In this kind of attack, an attacker or malicious nodes can take over the Node cluster and isolate the honest nodes and feed them with the fake information and thus turning them into dishonest node as well. The solution to this is that the honest node needs to be connected to at least one honest node, but if majority of the nodes in the cluster are malicious then compromising of honest nodes might occur. [24]

DdOS Attack: Blockchain being networked application, is prone to DDOS attacks. DdOS attack in case of Blockchain like Bitcoin and Ethereum can happen if 51% network is owned by certain entity or entities. Such cases of DdOS attack has happened to Bitfinex [25]. Mempool flooding is another form of DdOS attack that has happened to bitcoin.

Case Studies: "On November 11, 2017, the Bitcoin mempool size exceeded 115k unconfirmed transactions, resulting in $700 million USD worth of transaction stall. In June 2018, again the mempool was attacked with 4,500 unconfirmed spam transactions which increased the mempool size by 45MB. The increased size led to a spike in the mining fee and legitimate users were propelled to pay higher fee to get their transactions mined "[26]

Cryptojacking:

It is a form of attack that was conducted by attackers by using victim's browser to participate in PoW, or mining, without their consent or notice. UK's National Cyber Security Centre (NCSC), on its yearly cyber security report, declared cryptojacking as a "significant threat ".[22],[23],[27]

4.2 How these affect cryptocurrencies built on blockchains

There has been many notable security attacks and incidents on blockchain, below is a table that lists the most notable attacks that has led to huge financial losses:

Incident	Type of attack	Loss
Silkroad 2, 2014 Feb	Wallet attack:	$2,700,000

MT GOX (1st), 2011	Compromised system auditors computer	$30,000.
MT GOX(2nd)	Cryptocurrency theft: Exploitation of transaction malleability as the cause	$450,000,000
Ethereum(1st),2016	TheDAO attack happened because of the bug un the code	$70,000,000
Ethereum(2nd) 2017	Multi-sig DAPP vulnerability	$30,000,000
Ethereum(3rd)2017	Another wallet bug	$275,000,000 of frozen Ether which forced a software update
MyEtherWallet	DNS hijacking attack	$152,000
Bithumb, 2018	Phishing email sent to the users to redirect to malicious URL	$30,000,000

Table 1-Notable Cyber Attacks and lossess.

Cryptocurrency world has suffered huge losses from cyber-attack. The main motive in this kind of cyber-attack are mostly financial and design flaws of the blockchain.

4.3 Security issues with smart contracts based on existing taxonomies

Literature [3] has done a great work to analyze vulnerabilities in Ethereum smart contract and has provided a taxonomy based on the findings. A lot of vulnerabilities arise from the Solidity compiled bytecode. Even though contracts written on solidity looks like functions, EVM does not support functions and solidity functions are compiled as a unique signature [3]. If the signature matches a function on a contract, then it jumps to that part of code otherwise it will jump to fallback function. A fallback function is anonymous function taking no arguments and can be arbitrarily programmed. The taxonomy from [3] is shown below.

Level	Cause of vulnerability
Solidity	Call to the unknown
	Gasless send
	Exception disorders
	Type casts
	Reentrancy
	Keeping secrets
EVM	Immutable bugs
	Ether lost in trasfer
	Stack size limit
Blockchain	Unpredictable state
	Generating randomness
	Time constraints

Table 2-Taxonomy by [2]

The taxonomy shown in Table 2 classifies vulnerabilities found in Ethereum Smart contracts to three levels. The classification suggests that different vulnerabilities in Ethereum Smart Contact are from Programming Language Solidity or from flaws in EVM and some vulnerabilities directly relatable to the Blockchain. Each of the vulnerabilities from the taxonomy are explained in the following part.

4.3.1 Vulnerabilities in Solidity

The following vulnerabilities happen because of the flaws of the de-facto programming language solidity. Being a new language and similarities to C family languages, a developer might fall into some quirks that are not noticeable and can lead to attacks and one great example of the attack is The DAO hack. The vulnerabilities directly relating to solidity are explained below:

Call to unknown: When a function which doesn't exist in contract is called, it will lead to a fallback function, which is a special function that can be arbitrarily programmed, doesn't have names and arguments. This quirk can lead to arbitrary code execution by attacker.[3]

Gasless Send: Because Ethereum has a maximum gas limits and gas limit exception occurs when there is gas exhaustion. Gas is consumed with every execution of code and a contract can run into gas exhaustion. This behavior can be sometimes exploited to alter the state of contract by having inexpensive fallback function in case of exception and send Ether without paying gas. [3]

Exception Disorders: Some of the low-level code doesn't throw exception rather false or true, which if the developer hasn't well considered might make the contract vulnerable. Solidity has situations where

an exception like call-stack reached limit or gas is exhausted or throw command is executed. The way Solidity handles execution leads to vulnerability as exceptions are not predictable and sometimes handled just as true or false.[3]

Type casts: Solidity typecasting can lead programmers to false belief that it is error prone and might lead to unexpected vulnerabilities in the run-time. Solidity compiler does not perform error checks, except for few cases like assigning integer value to string as an example. Also at the runtime exception are not thrown in case of type errors and can lead to vulnerable behaviors.[3]

Reentrancy: This is the most well-known vulnerability that caused the DAO hack where $50 Million was stolen from a Contract named DAO [28]. Reentrancy happens when an attacker calls the vulnerable function multiple times, without the previous call being completed. In EVM the caller of the contract can perform message-calls to another contract, which can be performed by calling a function in another contract or sending Ether [3], [17]. Literature [17] mentions that the caller of the contract can perform arbitrary execution, even a call to other contract before returning the control to the caller function. If a contract is not reentrancy safe, the attacker can perform this sort of attack. [3] mentions that even when a non-recursive function is called, the attacker can re-enter the calling function multiple times because of the fallback mechanism of the EVM code. An example code from [17] shows the reentrancy which is shown below:

```
1 function withdraw() public {
2   uint balance = accounts[msg.sender];
3   msg.sender.call.value(balance)();
4   accounts[msg.sender] = 0;
5 }
```

Figure 9- Reentrancy Vulnerable code [15]

In figure 9, the contract has a reentrancy bug that allows the caller to call the line 3 code multiple times since the account is assigned value of 0 in line 4. The simple solution to this is just to move line 4 to line 3 shown in the figure 10 below:

```
1 function withdraw() public {
2   uint balance = accounts[msg.sender];
3   accounts[msg.sender] = 0;
4   msg.sender.call.value(balance)();
5 }
```

Figure 10-Reentrancy code bug fix [15]

Ethereum has no way as [3], [17]suggests to solve reentrancy. The only way is to beware of this quirk of Solidity and test the code before deployment that it is reentrancy safe.

Keeping Secrets: Contract fields can be public or private. Private field being private doesn't guarantee secrecy in Solidity code. Since the code are stored in the blockchain, which is a public ledger. There is no secrecy of the variables even they are declared as private.[3]

4.3.2 Vulnerabilities in EVM

There are other vulnerabilities that comes from the EVM. These vulnerabilities from the taxonomy are briefly discussed in the following part:

Immutable bugs: The smart contract by nature, once uploaded to the blockchain is immutable and will exists for the lifetime of blockchain. If there are any bugs in the contract, they can't be patched. This is a major security hazard and also the main reason that every contract that is deployed must be ensured that they are non-vulnerable. Once deployed to the chain, they cannot be patched and will remain vulnerable, and programmers have to just anticipate that this issue will be fixed. Many buggy contracts have been exploited but [3] explains that for TheDAO hack, the countermeasure was taken has a hard-fork rolling back to the previous commit and recovering the funds. This was against, for some of the Ethereum community, against the idea of "code is law" principle of blockchain and Ethereum Classic was formed as the disagreement. [28]

Ether lost in transfer: Ethereum addresses are of 160-bit sequence. If Ether is transferred to orphan address, that are not associated with any wallet or contract, then the transferred Ethereum will get lost.

Orphan addresses in this context means that the address is valid but does not belong to any user or any contract. Lost ethers cannot be recovered. A mistake in the address while sending ether can also go to wrong account and never be recovered. An user involving in the transactions needs to manually be sure of the address of the contract or the address of the recipient is valid.[3]

Stack size limit: The stack of EVM has size of 1024 frames, which was vulnerable to exception disorder and stack size limit vulnerabilities. Each time a code is executed the stack grows by 1. When it reaches the max size of 1024 there will be stack overflow throwing an exception. [3] mentions that until October 2016 an adversary could create almost full stack call to the victim's contract and an exception is thrown. If the victim's contract has not handled the exception well the attack would be successful. But the hard fork of Ethereum has addressed this issue.[2]

4.4 Vulnerabilities in Blockchain

Some vulnerabilities in the taxonomy are related to the blockchain itself. A brief discussion of the vulnerabilities is provided in the following:

Generating randomness:
Because of EVM's bytecodes Deterministic and thus true randomness is hard to achieve. To gain randomness, some contract creators use pseudo-random number generator which is based on the timestamp of the block that will happen in the future, as future blocks are not predictable. Miners are still under control of what goes to the blockchain, and they can see what pseudo-random numbers are used. A malicious miner can try to craft a block that could bias pseudo-random generations.[3]

Unpredictable state:

The contract's state changes with transaction, so it is hard to predict the state of contract when the transaction is being executed. There is no preservation of order in the transactions, a transaction send first still could be added to block later. Moreover, miners have choice over transaction ordering to add to the block. Another unpredictable state happens when there is a fork in the block and until the consensus is achieved which fork will be appended to the chain.[3]

Time constraints:

Time constraints are implemented as block-stamps, and they are agreed by all miners. Timestamp are shared by all the contract in a block and they can be exploited by malicious miners by choosing a suitable timestamp for the contract they are mining.

After the work [3] more vulnerabilities have emerged and DASP 10 suggests ten vulnerabilities found in Ethereum smart contracts. The DASP 10 vulnerabilities are show below:

No	Vulnerability	Vulnerability mentioned by Atzei et al.
1	Access Control	-
2	Arithmetic	-
3	Bad Randomness	Generating Randomness
4	Denial of service	-
5	Front running	Unpredictable State
6	Reentrancy	Call to unknown Reentrancy
7	Short addresses	-
8	Time manipulation	Time Constraints
9	Unchecked low level calls	Gasless Send Exception Disorders
10	Other	-

Table 3-DASP 10 Vulnerabilites mapping to Atzei Et al. [26]

Notable Attacks

Below are listed the example of well-known attack that happened to Ethereum smart contract [29].

- DAO hack: Because of reentrancy vulnerabilities, attacker drained 3.6 million ether into another contract [30]
- Parity Multisig Wallet hack:
  Vulnerability found in the contract of Parity Multisig led to stealing of 150,00= ETH [31]

5 Smart Contract analysis

In this chapter, necessary groundwork needed for the smart contract analysis are presented. Research questions, criteria for tools and smart contract for testing selection are discussed. They are the basis of the results which will be shown in chapter 6.

Investigation from earlier articles suggests that there are multiple variety of tools that are widely used in detection of smart contract vulnerabilities. In this section of the thesis, a list of ten popular tools is chosen to be tested. There was a perfect tool called SmartBugs [32] which was well suited for the experiment and research on the Vulnerable Ethereum smart contracts and analysis tools from the vulnerability taxonomies provided in earlier section. The main motivation of the research is to understand the tools used in Smart contract vulnerability detection, what are their strengths and how do they compare to each other in terms of vulnerability finding and time performance. Analysis of smart contract vulnerabilities are done with the help of Smartbugs which helps to gather all popular tools in one place. Smartbugs was well suited tool for this part of the research which saved time to rewrite a new tool and reinventing the wheel.

Research Questions:

1. What kind of vulnerabilities exist in Ethereum Smart contracts?
2. What Kind of tools exist to detect vulnerabilities?
3. How those vulnerabilities can be mapped to vulnerability taxonomies?
4. How can the tools be compared to each other in various terms?

5.1 Custom dataset selection

A set of 32 different smart contracts written in Solidity from various categories were chosen for the analysis. These data represent ten of the vulnerabilities from DASP10. The following table shows the category of the vulnerable contract and number of contracts chosen for the research.

All the contracts shown in Table 4 are selected from each of the vulnerabilities that are classified in DASP10 vulnerability index. The reason for this selection is to find what tools cover which of the vulnerabilities, so that an idea of use cases of tool can be derived. Furthermore, the coverage from all the tools on these contracts will give an estimation of what can be done to find and secure Smart contracts while development.

No	Category	Contracts Selected
1	Access Control	2
2	Arithmetic	5
3	Bad Randomness	2
4	Denial of service	3
5	Front running	2
6	Reentrancy	8
7	Short addresses	1
8	Time manipulation	3
9	Unchecked low level calls	3
10	Other	3
	Total:	32

Table 4-Selected contracts for research

5.2 Criteria for tools selection

The tools that were selected for the experiment and their availability is given in table 5. The tools selected for the research as shown in Table 5 are widely used symbolic execution, static analysis tool, except for Solhint which is a utility for linting solidity code. While other tools are also available, the tools given in the Table 5 are commonly used tools and recommended by various Ethereum Smart Contract security enthusiasts and developers. These tools are also already supported by SmartBugs, which was another reason for the selection. The criteria of selection were, a) easy integration with SmartBugs and b) favorability by security enthusiasts in Ethereum Development community. With the selected tools and the contracts, SmartBugs was used to execute the analysis and the results produced were analyzed. The results will be presented in Chapter 6 that follows.

No.	Name of the tool	What does it do?	Availability
1	HoneyBadger	Find Honey Pot in Ethereum contract	https://github.com/christoftorres/HoneyBadger
2	Maian	Prodigal, suicidal and greedy contract detection	https://github.com/MAIAN-tool/MAIAN
3	Manticore	Symbolic execution of EVM binaries and smart contracts	https://github.com/trailofbits/manticore/
4	Mythril	Smart contact disassembler and analysis	https://github.com/ConsenSys/mythril-classic
5	Osiris	Symbolic execution based on Oyente	https://github.com/christoftorres/Osiris
6	Oyente	Symbolic execution	https://github.com/melonproject/oyente
7	Securify	Static analysis	https://github.com/eth-sri/securify
8	Slither	Static analysis	https://github.com/crytic/slither
9	SmartCheck	Static analyis	https://github.com/smartdec/smartcheck
10	Solhint	Linting utility for Solidity	https://github.com/protofire/solhint

Table 5-Tools selected for the research

6 Presentation of the results

In this chapter, the result from the smart contact analysis done with the help of SmartBugs on custom datasets against widely used tools are presented. The finding from the experiment of the analysis is discussed.

The research was conducted with the help of SmartBugs with modification for the need of the research. A set of 32 vulnerable smart contract was chosen for the research. Each of the contract were selected from various categories of Vulnerabilities to meet the needs of the experiment. The data sets have been carefully selected to meet each of the vulnerabilities of the DASP 10.

6.1.1 SmartBugs

SmartBugs is an automation tool, that provides a common place for running varieties of symbolic execution and static analysis tools available for Ethereum Vulnerabilities. It is an automated tools runner, which does not do the analysis itself, rather bundles the available tools and runs test on the contracts supplied and produces the result output as a JSON or Sarif file. The outputted result files need to be analyzed separately. It can be used to collect data on smart contract vulnerabilities. SmartBugs is written in Python3. SmartBugs is extensible. The tools needed to be used and supported by the framework can be added. SmartBugs uses Docker Images of tools that are supported and runs them in a sequential manner against the provided contracts. SmartBugs is easy to use and can perform bulk analysis of contracts. Also, the authors have provided already large set of datasets covering many different Ethereum Vulnerabilities. SmartBugs also already comes with a huge list of widely used tools like, Mythril, Maian and Slither to name a few popular ones [32].

On the source code of SmartBugs, time log was added to measure the time taken by each tool for analyzing the contracts. The code added is appended to the Appendix 1. The following figure shows the time taken to perform an analysis by all the tools in the updated SmartBugs software and output window from the SmartBugs.

6.1.2 Outcome Of the Test

```
--------------------------------------------------------------------------------

Done [351/352, 0:03:11]: dataset/test_dataset/simple_dao.sol [maian] in 0:06:36
Analysing file [351/352]: dataset/test_dataset/simple_dao.sol [securify]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2

----Average Time and Total calculations----------------------------------

Tool: conkas, Total calculations: 32, Average Time: 42.238749999999996 seconds
Tool: mythril, Total calculations: 32, Average Time: 104.30718750000001 seconds
Tool: osiris, Total calculations: 32, Average Time: 12.3603125 seconds
Tool: slither, Total calculations: 32, Average Time: 2.0865624999999994 seconds
Tool: oyente, Total calculations: 32, Average Time: 8.032499999999999 seconds
Tool: solhint, Total calculations: 32, Average Time: 3.0803124999999993 seconds
Tool: smartcheck, Total calculations: 32, Average Time: 793.62625 seconds
Tool: honeybadger, Total calculations: 32, Average Time: 45.271875 seconds
Tool: manticore, Total calculations: 32, Average Time: 302.7618750000001 seconds
Tool: maian, Total calculations: 32, Average Time: 597.8053124999999 seconds
Tool: securify, Total calculations: 32, Average Time: 176.38624999999996 seconds

--------------------------------------------------------------------------------

pas terrible
Done [352/352, 0:00:00]: dataset/test_dataset/simple_dao.sol [securify] in 0:00:09
Analysis completed.
It took 1114m 44s to analyse all files.
```

Figure 11-Output of Average from the analysis

In the above figure 11 of SmartBugs output in terminal, the average time consumptions and total calculations part was added to the original code and the source can be found later in the appendix. Other lines are the output of SmartBugs itself, which show the file being analyzed, the tool used for analysis and time take for each execution of the tool. The following figure 12 shows the time taken to perform analysis by all the tools in the updated SmartBugs software. Below is a bar chart of the average time taken by each tool output:

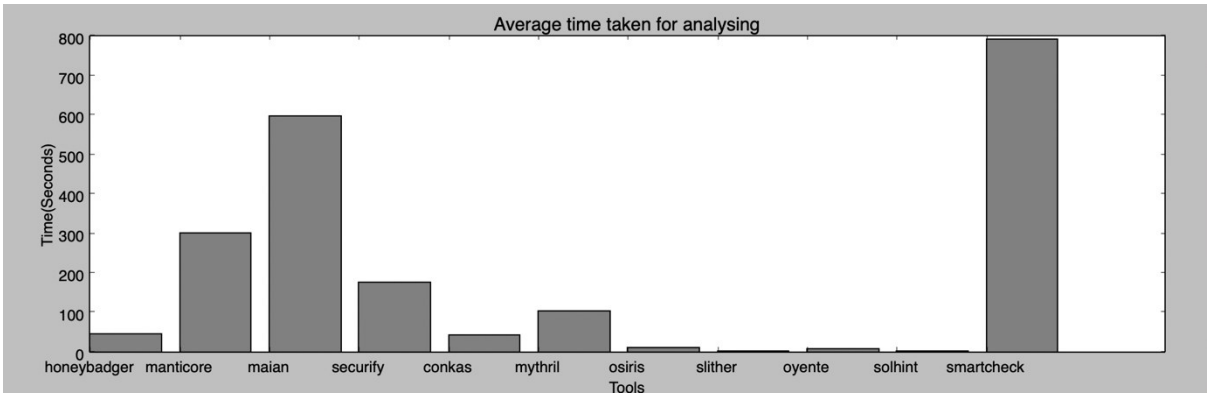


Figure 12-Bar chart of average time taken by tools

From the above figure, SmartCheck, Maian and Manticore took the longest time to analyze the contract. The reasons for this will be clarified in the section where each tool will be mapped to vulnerabilities. The data produced from the SmartBugs was in Json Format and Sarif format. For simplicity, json data was used. Python and manual analysis were used to analyze the data and produce the output. The plot graph was also drawn using python Matplotlib library. The analysis of the data was manual process. And python was used later to work on the data and produce the result. The comparison matrix from the outcome is shown in the following matrix:

	securify	**maian**	**manticore**	**smartcheck**	**solhint**	**honeybadger**	**slither**	**mythril**	**conkas**	**osiris**
crypto_roulette	0	0	0	0	0	1	11	3	8	1
dos_address	0	0	0	0	1	0	4	2	2	0
dos_number	0	0	0	4	0	0	5	4	1	0
dos_simple	0	0	0	0	0	0	1	1	0	0
ERC20	0	0	0	0	68	0	7	0	0	0
eth_tx_order_dependence_minimal	1	0	0	0	0	0	4	4	4	1
ether_lotto	1	0	3	7	1	0	7	5	4	1
etherpot_lotto	0	0	0	34	95	0	14	8	5	1
governmental_survey	0	0	0	0	30	0	13	7	12	2
insecure_transfer	0	0	0	1	2	0	5	1	0	1
integer_overflow_1	0	0	1	4	7	0	2	1	1	1

king_of_the_ether_throne	0	0	0	21	0	0	12	NaN	12	1
Lottery	0	0	3	20	36	0	12	1	3	1
lotto	1	1	0	3	15	0	7	5	4	1
lottopollo	0	0	2	0	18	0	4	3	6	1
mishandled	0	0	2	0	4	0	2	2	2	1
name_registrar	0	0	4	1	0	0	5	0	0	1
odds_and_evens	1	0	1	10	30	1	7	9	4	1
open_address_lottery	0	0	5	17	2	1	9	4	5	1
overflow_simple_add	0	0	1	0	0	0	2	1	1	0
parity_wallet_bug_1	0	0	0	0	276	0	0	0	0	0
parity_wallet_bug_2	0	2	0	50	241	0	52	16	0	2
random_number_generator	0	0	1	0	9	0	0	0	0	0
reentrance	1	0	10	0	16	0	10	4	3	1
Reentrancy_cross_function	0	0	3	5	1	0	7	4	2	0
reentrancy_bonus	0	0	0	6	0	0	7	3	2	0
reentrancy_dao	1	0	3	5	0	0	5	5	5	1
reentrancy_insecure	1	0	1	0	1	0	5	3	2	0
reentrancy_simple	1	0	5	10	16	0	7	4	3	1
short_address_example	0	0	3	7	18	0	4	1	0	1
simple_dao	1	0	10	0	13	0	6	4	3	1
spank_chain_payment	0	0	0	0	680	0	120	2	7	0

Table 6-Total number of vulnerability results produced by various tools against the dataset

Color Code Yellow: Data considered as anomaly

The above-mentioned experiment was done on the data produced by SmartBugs which were in json format. There was lot of manual analysis of the data. A separate test was made to look at the output of

the data directly by analysis tools by slight modification on the code of SmartBugs. The original code of Smartbugs did not have such an output. SmartBugs tool was modified to look at the terminal logs from the tools directly before it parses to the parser for data collection. This process gave more clarity on each tool than the data that was produced from SmartBugs which needed data analyzing and nested json parsing in some case. The contract selected for this purpose was SimpleDao, as reentrancy is one of the biggest vulnerabilities of Ethereum smart contract.

The sample contract of simple DAO is posted below:

```
/*
 * @source: http://blockchain.unica.it/projects/ethereum-
survey/attacks.html#simpledao
 * @author: -
 * @vulnerable_at_lines: 19
 */

pragma solidity ^0.4.2;

contract SimpleDAO {
  mapping (address => uint) public credit;

  function donate(address to) payable {
    credit[to] += msg.value;
  }

  function withdraw(uint amount) {
    if (credit[msg.sender]>= amount) {
      // <yes> <report> REENTRANCY
      bool res = msg.sender.call.value(amount)();
      credit[msg.sender]-=amount;
    }
  }

  function queryCredit(address to) returns (uint){
    return credit[to];
  }
}
```

Figure 13-SimpleDAO contract code

The contract in Figure 13 has Reentrancy vulnerability. The contract was taken from sample contracts provided in Smartbugs. The contract has Reentrancy vulnerability on the yellow marker where the amount is deducted after calling the function. This is the major cause of Reentrancy as the fallback may be exploited by attacker many times emptying the contract. This is the famous TheDAO hack. The contract above has been run and analyzed tool- by-tool basis. A discussion on the tool and their performance are provided. A terminal screenshot is also added as they give more clear idea on the state-of-art tools and their abilities.

Conkas:

```
binod@Binods-MacBook-Pro smartbugs % python3 smartBugs.py --tool all --file dataset/reentrancy/sim
ple_dao.sol
Analysing file [0/11]: dataset/reentrancy/simple_dao.sol [conkas]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
Analysing /dataset/reentrancy/simple_dao.sol:SimpleDAO...
Vulnerability: Reentrancy. Maybe in function: withdraw(uint256). PC: 0x237. Line number: 19.
Vulnerability: Integer Overflow. Maybe in function: donate(address). PC: 0x1bc. Line number: 13.
If a = 1
and b = 115792089237316195423570985008687907853269984665640564039457584007913129639935
Vulnerability: Unchecked Low Level Call. Maybe in function: withdraw(uint256). PC: 0x237. Line num
ber: 19.

----Average Time and Total calculations----------------------------------

Tool: conkas, Total calculations: 1, Average Time: 5.55 seconds

--------------------------------------------------------------------------
```

Figure 14-Conkas Output

Conkas has found Reentrancy, Integer Overflow, Unchecked Low Level calls. Verbosity is good and line numbers are clarified. Average time is also 5.5 seconds which is satisfactory.

Mythril:

```
Analysing file [1/11]: dataset/reentrancy/simple_dao.sol [mythril]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
{"error": null, "issues": [{"address": 444, "code": "credit[to] += msg.value", "debug": "storage_k
eccac_1461501637330902918203684832716283019655932542975_&\n1461501637330902918203684832716283019655
5932542975_&\n1461501637330902918203684832716283019655932542975_&\ncalldata_SimpleDAO_4: 0xa8795cd
3fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe\ncallvalue: 0x57f25cd0000000000000000000
0000000000000000000000000000000000000000\ncalldata_SimpleDAO_0: 0x362a9500000000000000000000000000000
0000000000000000000000000000\ncalldatasize_SimpleDAO: 0x4\n", "description": "A possible integer o
verflow exists in the function `fallback`.\nThe addition or multiplication may result in a value h
igher than the maximum representable integer.", "filename": "/dataset/reentrancy/simple_dao.sol",
"function": "fallback", "lineno": 13, "title": "Integer Overflow ", "type": "Warning"}, {"address"
: 567, "code": "msg.sender.call.value(amount)()", "debug": "", "description": "This contract execu
tes a message call to the address of the transaction sender. Generally, it is not recommended to c
all user-supplied addresses using Solidity's call() construct. Note that attackers might leverage
reentrancy attacks to exploit race conditions or manipulate this contract's state.", "filename": "
/dataset/reentrancy/simple_dao.sol", "function": "withdraw(uint256)", "lineno": 19, "title": "Mess
age call to external contract", "type": "Warning"}, {"address": 567, "code": "msg.sender.call.valu
e(amount)()", "debug": "", "description": "The return value of an external call is not checked. No
te that execution continue even if the called contract throws.", "filename": "/dataset/reentrancy/
simple_dao.sol", "function": "withdraw(uint256)", "lineno": 19, "title": "Unchecked CALL return va
lue", "type": "Informational"}, {"address": 648, "code": "credit[msg.sender]-=amount", "debug": ""
, "description": "The contract account state is changed after an external call. Consider that the
called contract could re-enter the function before this state change takes place. This can lead to
 business logic vulnerabilities.", "filename": "/dataset/reentrancy/simple_dao.sol", "function": "
withdraw(uint256)", "lineno": 20, "title": "State change after external call", "type": "Warning"}]
, "success": true}

----Average Time and Total calculations----------------------------------

Tool: conkas, Total calculations: 1, Average Time: 5.55 seconds
Tool: mythril, Total calculations: 1, Average Time: 11.72 seconds
```

Figure 15-Mythril output

While Mythril found out Integer Overflow, Reentrancy and Unchecked low level calls, the verbosity of the output is not human-readable rather json programmable. This will have an added overhead to smart contract developers to try to parse the JSON. The messages are meaningful despite being hard to analyze without help of JSON parser. Time taken is satisfactory.

Osiris:

```
Analysing file [2/11]: dataset/reentrancy/simple_dao.sol [osiris]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
INFO:root:Contract /dataset/reentrancy/simple_dao.sol:SimpleDAO:
INFO:symExec:Running, please wait...
INFO:symExec:    ============ Results ===========
INFO:symExec:      EVM code coverage:      99.7%
INFO:symExec:      Arithmetic bugs:        True
INFO:symExec:       └> Overflow bugs:      True
/dataset/reentrancy/simple_dao.sol:SimpleDAO:13:5
credit[to] += msg.value
^
INFO:symExec:       └> Underflow bugs:     True
/dataset/reentrancy/simple_dao.sol:SimpleDAO:20:7
credit[msg.sender]-=amount
^
INFO:symExec:       └> Division bugs:      False
INFO:symExec:       └> Modulo bugs:        False
INFO:symExec:       └> Truncation bugs:    False
INFO:symExec:       └> Signedness bugs:    False
INFO:symExec:      Callstack bug:          True
/dataset/reentrancy/simple_dao.sol:SimpleDAO:19:18
msg.sender.call.value(amount)()
^
INFO:symExec:      Concurrency bug:        False
INFO:symExec:      Time dependency bug:    False
INFO:symExec:      Reentrancy bug:         True
/dataset/reentrancy/simple_dao.sol:SimpleDAO:19:18
msg.sender.call.value(amount)()
^
INFO:symExec:    --- 3.58133101463 seconds ---
INFO:symExec:    ====== Analysis Completed ======

  .oooooo.                o8o              o8o
 d8P'   `Y8b               `"'              `"'
888       888  .oooo.o oooo   oooo d8b oooo    .oooo.o
888       888 d88(   "8 `888   `888""8P `888   d88(   "8
888       888 `"Y88b.    888    888       888   `"Y88b.
`88b     d88' o.  )88b   888    888       888   o.  )88b
 `Y8bood8P'  8""888P' o888o d888b     o888o 8""888P'

----Average Time and Total calculations--------------------------------

Tool: conkas, Total calculations: 1, Average Time: 5.55 seconds
Tool: mythril, Total calculations: 1, Average Time: 11.72 seconds
Tool: osiris, Total calculations: 1, Average Time: 6.16 seconds

------------------------------------------------------------------------
```

Figure 16-Osiris Output

Osiris Detected Arithmetic Bugs, Callstack Bug and Reentrancy bug in the contract. Callstack bug is something new that was not previously in Conkas or Mythril. Also, the output looks very readable. The verbosity looks very good. The true false representation makes it easier for anyone to follow and what to expect for. However, there was noticeable time difference from what Osiris reported and what was recorded via the code that was added to SmartBugs. This also suggests that there is extra overhead of few seconds in the execution and finalization of report from SmartBugs. This is reasonable time overhead.

Slither:

```
Analysing file [3/11]: dataset/reentrancy/simple_dao.sol [slither]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
INFO:Slither:Compilation warnings/errors on /dataset/reentrancy/simple_dao.sol:
/dataset/reentrancy/simple_dao.sol:12:3: Warning: No visibility specified. Defaulting to "public".

  function donate(address to) payable {
  ^ (Relevant source part starts here and spans across multiple lines).
/dataset/reentrancy/simple_dao.sol:16:3: Warning: No visibility specified. Defaulting to "public".

  function withdraw(uint amount) {
  ^ (Relevant source part starts here and spans across multiple lines).
/dataset/reentrancy/simple_dao.sol:19:7: Warning: Unused local variable.
      bool res = msg.sender.call.value(amount)();
      ^------^
/dataset/reentrancy/simple_dao.sol:24:3: Warning: No visibility specified. Defaulting to "public".

  function queryCredit(address to) returns (uint){
  ^ (Relevant source part starts here and spans across multiple lines).
/dataset/reentrancy/simple_dao.sol:24:3: Warning: Function state mutability can be restricted to v
iew
  function queryCredit(address to) returns (uint){
  ^ (Relevant source part starts here and spans across multiple lines).

INFO:Detectors:
Reentrancy in SimpleDAO.withdraw (/dataset/reentrancy/simple_dao.sol#16-22):
        External calls:
        - res = msg.sender.call.value(amount)() (/dataset/reentrancy/simple_dao.sol#19)
        State variables written after the call(s):
        - credit (/dataset/reentrancy/simple_dao.sol#20)
Reference: https://github.com/trailofbits/slither/wiki/Detectors-Documentation#reentrancy-vulnerab
ilities
INFO:Detectors:
SimpleDAO.donate (/dataset/reentrancy/simple_dao.sol#12-14) should be declared external
SimpleDAO.withdraw (/dataset/reentrancy/simple_dao.sol#16-22) should be declared external
SimpleDAO.queryCredit (/dataset/reentrancy/simple_dao.sol#24-26) should be declared external
Reference: https://github.com/trailofbits/slither/wiki/Detectors-Documentation#public-function-tha
t-could-be-declared-as-external
INFO:Detectors:
Detected issues with version pragma in /dataset/reentrancy/simple_dao.sol:
        - pragma solidity^0.4.2 (/dataset/reentrancy/simple_dao.sol#7): it allows old versions
Reference: https://github.com/trailofbits/slither/wiki/Detectors-Documentation#incorrect-version-o
f-solidity
INFO:Detectors:
Low level call in SimpleDAO.withdraw (/dataset/reentrancy/simple_dao.sol#16-22):
        -res = msg.sender.call.value(amount)() /dataset/reentrancy/simple_dao.sol#19
Reference: https://github.com/trailofbits/slither/wiki/Detectors-Documentation#low-level-calls
INFO:Slither:/dataset/reentrancy/simple_dao.sol analyzed (1 contracts), 6 result(s) found

----Average Time and Total calculations---------------------------------

Tool: conkas, Total calculations: 1, Average Time: 5.55 seconds
Tool: mythril, Total calculations: 1, Average Time: 11.72 seconds
Tool: osiris, Total calculations: 1, Average Time: 6.16 seconds
Tool: slither, Total calculations: 1, Average Time: 2.2 seconds

------------------------------------------------------------------------
```

Figure 17-Slither Output

Slither has extra information that are very helpful in form of warning. Low Level Calls and Reentrancy were detected, and the color coding makes things more informational. The verbosity is good, while readability is satisfactory. The performance is good too.

Oyente:

```
Analysing file [4/11]: dataset/reentrancy/simple_dao.sol [oyente]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
WARNING:root:You are using evm version 1.8.2. The supported version is 1.7.3
WARNING:root:You are using solc version 0.4.25, The latest supported version is 0.4.19
INFO:root:contract /dataset/reentrancy/simple_dao.sol:SimpleDAO:
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM Code Coverage:                    99.7%
INFO:symExec:     Integer Underflow:                    False
INFO:symExec:     Integer Overflow:                     True
INFO:symExec:     Parity Multisig Bug 2:                False
INFO:symExec:     Callstack Depth Attack Vulnerability:  True
INFO:symExec:     Transaction-Ordering Dependence (TOD): False
INFO:symExec:     Timestamp Dependency:                 False
INFO:symExec:     Re-Entrancy Vulnerability:            True
INFO:symExec:/dataset/reentrancy/simple_dao.sol:13:5: Warning: Integer Overflow.
    credit[to] += msg.value
Integer Overflow occurs if:
    credit[to] = 1
INFO:symExec:/dataset/reentrancy/simple_dao.sol:19:18: Warning: Callstack Depth Attack Vulnerabili
ty.
      bool res = msg.sender.call.value(amount)()
INFO:symExec:/dataset/reentrancy/simple_dao.sol:19:18: Warning: Re-Entrancy Vulnerability.
      bool res = msg.sender.call.value(amount)()
INFO:symExec:   ====== Analysis Completed ======

----Average Time and Total calculations--------------------------------

Tool: conkas, Total calculations: 1, Average Time: 5.55 seconds
Tool: mythril, Total calculations: 1, Average Time: 11.72 seconds
Tool: osiris, Total calculations: 1, Average Time: 6.16 seconds
Tool: slither, Total calculations: 1, Average Time: 2.2 seconds
Tool: oyente, Total calculations: 1, Average Time: 7.41 seconds

------------------------------------------------------------------------
```

Figure 18-Oyente Output

Oyente has similar clarity to Osiris, messages shown as true/false. Oyente was able to detect Reentrancy, Callstack Depth and Integer Overflow. The verbosity and time consumption are also good.

Solhint:

```
Analysing file [5/11]: dataset/reentrancy/simple_dao.sol [solhint]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
/dataset/reentrancy/simple_dao.sol:10:3: Expected indentation of 4 spaces but found 2 [Error/indent]
/dataset/reentrancy/simple_dao.sol:12:3: Expected indentation of 4 spaces but found 2 [Error/indent]
/dataset/reentrancy/simple_dao.sol:13:5: Expected indentation of 8 spaces but found 4 [Error/indent]
/dataset/reentrancy/simple_dao.sol:14:3: Expected indentation of 4 spaces but found 2 [Error/indent]
/dataset/reentrancy/simple_dao.sol:16:3: Expected indentation of 4 spaces but found 2 [Error/indent]
/dataset/reentrancy/simple_dao.sol:17:5: Expected indentation of 8 spaces but found 4 [Error/indent]
/dataset/reentrancy/simple_dao.sol:19:7: Expected indentation of 12 spaces but found 6 [Error/indent]
/dataset/reentrancy/simple_dao.sol:20:7: Expected indentation of 12 spaces but found 6 [Error/indent]
/dataset/reentrancy/simple_dao.sol:21:5: Expected indentation of 8 spaces but found 4 [Error/indent]
/dataset/reentrancy/simple_dao.sol:22:3: Expected indentation of 4 spaces but found 2 [Error/indent]
/dataset/reentrancy/simple_dao.sol:24:3: Expected indentation of 4 spaces but found 2 [Error/indent]
/dataset/reentrancy/simple_dao.sol:25:5: Expected indentation of 8 spaces but found 4 [Error/indent]
/dataset/reentrancy/simple_dao.sol:26:3: Expected indentation of 4 spaces but found 2 [Error/indent]

13 problems

----Average Time and Total calculations----------------------------------

Tool: conkas, Total calculations: 1, Average Time: 5.55 seconds
Tool: mythril, Total calculations: 1, Average Time: 11.72 seconds
Tool: osiris, Total calculations: 1, Average Time: 6.16 seconds
Tool: slither, Total calculations: 1, Average Time: 2.2 seconds
Tool: oyente, Total calculations: 1, Average Time: 7.41 seconds
Tool: solhint, Total calculations: 1, Average Time: 2.96 seconds

------------------------------------------------------------------------
```

Figure 19-Solhint Output

Solhint is a linting tool for Solidity, and as the result shows that it has been complaining about the indentation. As a linting tool it can help for consistency of writing code, but Solhint does not seem to be particularly useful for detecting vulnerability.

Smartcheck:

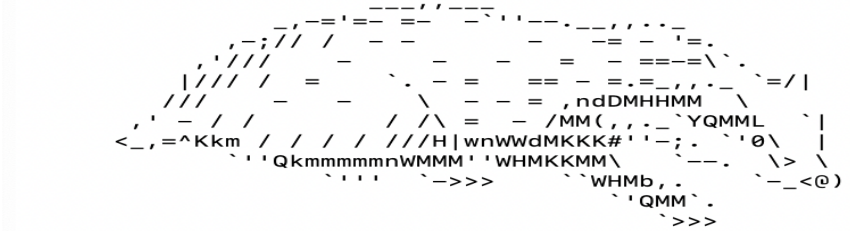
```
Done [6/11, 0:00:31]: dataset/reentrancy/simple_dao.sol [solhint] in 0:00:03
Analysing file [6/11]: dataset/reentrancy/simple_dao.sol [smartcheck]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2


----Average Time and Total calculations----------------------------------

Tool: conkas, Total calculations: 1, Average Time: 5.55 seconds
Tool: mythril, Total calculations: 1, Average Time: 11.72 seconds
Tool: osiris, Total calculations: 1, Average Time: 6.16 seconds
Tool: slither, Total calculations: 1, Average Time: 2.2 seconds
Tool: oyente, Total calculations: 1, Average Time: 7.41 seconds
Tool: solhint, Total calculations: 1, Average Time: 2.96 seconds
Tool: smartcheck, Total calculations: 1, Average Time: 1800.29 seconds

------------------------------------------------------------------------
```

Figure 20-Smartcheck output

There was no particular output to terminal from smartcheck but the time consumption is surprisingly too long than average. It is not verbose at all, time consuming and later in the discussion it will be provided information on what Smartbugs was able to collect from smartcheck.

Honeybadger:

```
Analysing file [7/11]: dataset/reentrancy/simple_dao.sol [honeybadger]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
INFO:root:Contract /dataset/reentrancy/simple_dao.sol:SimpleDAO:
INFO:symExec:Running, please wait...
INFO:symExec:    ============ Results ===========
INFO:symExec:     EVM code coverage:       99.7%
INFO:symExec:     Money flow:              True
INFO:symExec:     Balance disorder:        False
INFO:symExec:     Hidden transfer:         False
INFO:symExec:     Inheritance disorder:    False
INFO:symExec:     Uninitialised struct:    False
INFO:symExec:     Type overflow:           False
INFO:symExec:     Skip empty string:       False
INFO:symExec:     Hidden state update:     False
INFO:symExec:     Straw man contract:      False
INFO:symExec:     --- 3.55458784103 seconds ---
INFO:symExec:    ====== Analysis Completed ======
```



```
 _    _                                  ____                        _
| |  | |                                |  _ \                      | |
| |__| | ___  _ __   ___ _   _| |_) | __ _  __| | __ _  ___ _ __
|  __  |/ _ \| '_ \ / _ \ | | |  _ < / _` |/ _` |/ _` |/ _ \ '__|
| |  | | (_) | | | |  __/ |_| | |_) | (_| | (_| | (_| |  __/ |
|_|  |_|\___/|_| |_|\___|\__, |____/ \__,_|\__,_|\__, |\___|_|
                          __/ |                    __/ |
                         |___/                    |___/
```

```
----Average Time and Total calculations--------------------------------

Tool: conkas, Total calculations: 1, Average Time: 5.55 seconds
Tool: mythril, Total calculations: 1, Average Time: 11.72 seconds
Tool: osiris, Total calculations: 1, Average Time: 6.16 seconds
Tool: slither, Total calculations: 1, Average Time: 2.2 seconds
Tool: oyente, Total calculations: 1, Average Time: 7.41 seconds
Tool: solhint, Total calculations: 1, Average Time: 2.96 seconds
Tool: smartcheck, Total calculations: 1, Average Time: 1800.29 seconds
Tool: honeybadger, Total calculations: 1, Average Time: 5.99 seconds

------------------------------------------------------------------------
```

Figure 21-Honeybadger Output

Honeybadger specializes in finding honeypots in smart contracts. Honeypots in smart contract are designed to scam victims by making a contract code look vulnerable and making the attacker believe that if they send some ether first the contract can be exploited. It can be a good tool for detecting honeypots by security researchers on the Ethereum blockchain.

Manticore:

```
Analysing file [8/11]: dataset/reentrancy/simple_dao.sol [manticore]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
2021-09-23 16:47:25,652: [26] m.main:INFO: Registered plugins: DetectReentrancyAdvanced, DetectInv
alid, DetectSuicidal, DetectDelegatecall, DetectUninitializedStorage, DetectExternalCallAndLeak, D
etectUnusedRetVal, DetectIntegerOverflow, DetectUninitializedMemory, DetectEnvInstruction, DetectR
eentrancySimple
2021-09-23 16:47:25,654: [26] m.main:INFO: Beginning analysis
2021-09-23 16:47:25,686: [26] m.e.manticore:INFO: Starting symbolic create contract
2021-09-23 16:47:29,474: [26] m.e.manticore:INFO: Starting symbolic transaction: 0
2021-09-23 16:47:39,327: [360] m.e.detectors:WARNING: Reachable external call to sender
2021-09-23 16:47:41,939: [360] m.e.detectors:WARNING: Potential reentrancy vulnerability
2021-09-23 16:47:42,482: [360] m.e.detectors:WARNING: Reentrancy multi-million ether bug
2021-09-23 16:47:46,992: [360] m.e.detectors:WARNING: Returned value at CALL instruction is not us
ed
2021-09-23 16:47:47,232: [26] m.e.manticore:INFO: 5 alive states, 5 terminated states
2021-09-23 16:47:49,558: [26] m.e.manticore:INFO: Starting symbolic transaction: 1
2021-09-23 16:48:22,599: [1855] m.e.detectors:WARNING: Reachable external call to sender
2021-09-23 16:48:33,134: [1855] m.e.detectors:WARNING: Potential reentrancy vulnerability
2021-09-23 16:48:34,170: [1855] m.e.detectors:WARNING: Reentrancy multi-million ether bug
2021-09-23 16:48:35,121: [1860] m.e.detectors:WARNING: Reachable external call to sender
2021-09-23 16:48:36,416: [1853] m.e.detectors:WARNING: Potentially reading uninitialized storage
2021-09-23 16:48:48,356: [1860] m.e.detectors:WARNING: Potential reentrancy vulnerability
2021-09-23 16:48:50,176: [1860] m.e.detectors:WARNING: Reentrancy multi-million ether bug
2021-09-23 16:49:00,078: [1855] m.e.detectors:WARNING: Returned value at CALL instruction is not u
sed
```

Figure 22-Manticore output

```
----Average Time and Total calculations----------------------------------

Tool: conkas, Total calculations: 1, Average Time: 5.55 seconds
Tool: mythril, Total calculations: 1, Average Time: 11.72 seconds
Tool: osiris, Total calculations: 1, Average Time: 6.16 seconds
Tool: slither, Total calculations: 1, Average Time: 2.2 seconds
Tool: oyente, Total calculations: 1, Average Time: 7.41 seconds
Tool: solhint, Total calculations: 1, Average Time: 2.96 seconds
Tool: smartcheck, Total calculations: 1, Average Time: 1800.29 seconds
Tool: honeybadger, Total calculations: 1, Average Time: 5.99 seconds
Tool: manticore, Total calculations: 1, Average Time: 379.88 seconds

----------------------------------------------------------------------------
```

Figure 23-Manticore Time consumption

The output of manticore was too big but the warnings was repeated and only a small portion of the screenshot has been taken. Manticore was also able to detect External Calls and reentrancy vulnerability. The output has verbosity but too detailed. The time consumption was one of the largest after Smartcheck.

Maian:

```
Analysing file [9/11]: dataset/reentrancy/simple_dao.sol [maian]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
/usr/local/lib/python2.7/dist-packages/web3/main.py:130: DeprecationWarning: Python 2 support is e
nding! Please upgrade to Python 3 promptly. Support will end at the beginning of 2018.
  category=DeprecationWarning,
/usr/local/lib/python2.7/dist-packages/web3/main.py:130: DeprecationWarning: Python 2 support is e
nding! Please upgrade to Python 3 promptly. Support will end at the beginning of 2018.
  category=DeprecationWarning,

================================================================================================
==
[ ] Compiling Solidity contract from the file /dataset/reentrancy/simple_dao.sol ...  Done
[ ] Connecting to PRIVATE blockchain emptychain  . ESTABLISHED
[ ] Deploying contract ..........................................................................
................................................................................................
................................................................................................
.......................... confirmed at address: 0x9E536236ABF2288a7864C6A1AfaA4Cb98D464306
[ ] Contract code length on the blockchain :    1590  : 0x6080604052600436106100615760...
[ ] Contract address saved in file: ./out/SimpleDAO.address
[ ] Check if contract is SUICIDAL

[ ] Contract address   : 0x9E536236ABF2288a7864C6A1AfaA4Cb98D464306
[ ] Contract bytecode  : 6080604052600436106100615760003 57c0100000000000000...
[ ] Bytecode length    : 1588
[ ] Blockchain contract: True
[ ] Debug              : False

[-] The code does not contain SUICIDE instructions, hence it is not vulnerable
/usr/local/lib/python2.7/dist-packages/web3/main.py:130: DeprecationWarning: Python 2 support is e
nding! Please upgrade to Python 3 promptly. Support will end at the beginning of 2018.
  category=DeprecationWarning,
/usr/local/lib/python2.7/dist-packages/web3/main.py:130: DeprecationWarning: Python 2 support is e
nding! Please upgrade to Python 3 promptly. Support will end at the beginning of 2018.
  category=DeprecationWarning,

================================================================================================
==
[ ] Compiling Solidity contract from the file /dataset/reentrancy/simple_dao.sol ...  Done
[ ] Connecting to PRIVATE blockchain emptychain  . ESTABLISHED
[ ] Sending Ether to contract 0x9e536236abf2288a7864c6a1afaa4cb98d464306  ........ tx[0] mined  Se
nt!
[ ] Deploying contract ...... confirmed at address: 0x9E536236ABF2288a7864C6A1AfaA4Cb98D464306
[ ] Contract code length on the blockchain :    1590  : 0x6080604052600436106100615760...
[ ] Contract address saved in file: ./out/SimpleDAO.address
[ ] The contract balance: 44   Positive balance
[ ] Check if contract is PRODIGAL

[ ] Contract address   : 0x9E536236ABF2288a7864C6A1AfaA4Cb98D464306
[ ] Contract bytecode  : 6080604052600436106100615760003 57c0100000000000000...
[ ] Bytecode length    : 1588
[ ] Blockchain contract: True
[ ] Debug              : False

[ ] Search with call depth: 1   : 11
[ ] Search with call depth: 2   : 122122
[ ] Search with call depth: 3   : 1233212332
[+] No prodigal vulnerability found
/usr/local/lib/python2.7/dist-packages/web3/main.py:130: DeprecationWarning: Python 2 support is e
nding! Please upgrade to Python 3 promptly. Support will end at the beginning of 2018.
  category=DeprecationWarning,
```

Figure 24-Maian Output

```
----Average Time and Total calculations----------------------------------

Tool: conkas, Total calculations: 1, Average Time: 5.55 seconds
Tool: mythril, Total calculations: 1, Average Time: 11.72 seconds
Tool: osiris, Total calculations: 1, Average Time: 6.16 seconds
Tool: slither, Total calculations: 1, Average Time: 2.2 seconds
Tool: oyente, Total calculations: 1, Average Time: 7.41 seconds
Tool: solhint, Total calculations: 1, Average Time: 2.96 seconds
Tool: smartcheck, Total calculations: 1, Average Time: 1800.29 seconds
Tool: honeybadger, Total calculations: 1, Average Time: 5.99 seconds
Tool: manticore, Total calculations: 1, Average Time: 379.88 seconds
Tool: maian, Total calculations: 1, Average Time: 383.0 seconds

--------------------------------------------------------------------------
```

Figure 25-Maian Time consumption

Maian specializes in finding suicidal, prodigal and lock vulnerable contracts. It did not detect any of those in the test contract. The output is verbose. Time consumption was quite big compared to other tools.

Securify:

```
Done [10/11, 0:04:21]: dataset/reentrancy/simple_dao.sol [maian] in 0:06:23
Analysing file [10/11]: dataset/reentrancy/simple_dao.sol [securify]
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2
ANTLR runtime and generated code versions disagree: 4.9.2!=4.7.2


----Average Time and Total calculations----------------------------------

Tool: conkas, Total calculations: 1, Average Time: 5.55 seconds
Tool: mythril, Total calculations: 1, Average Time: 11.72 seconds
Tool: osiris, Total calculations: 1, Average Time: 6.16 seconds
Tool: slither, Total calculations: 1, Average Time: 2.2 seconds
Tool: oyente, Total calculations: 1, Average Time: 7.41 seconds
Tool: solhint, Total calculations: 1, Average Time: 2.96 seconds
Tool: smartcheck, Total calculations: 1, Average Time: 1800.29 seconds
Tool: honeybadger, Total calculations: 1, Average Time: 5.99 seconds
Tool: manticore, Total calculations: 1, Average Time: 379.88 seconds
Tool: maian, Total calculations: 1, Average Time: 383.0 seconds
Tool: securify, Total calculations: 1, Average Time: 8.28 seconds

--------------------------------------------------------------------------
```

Figure 26-Securify time consumption

Securify did not produce any output on this experiment run. Just as a quick testing tool, Securify would not qualify. However, looking at the Json Result from Smartbug, there was no mentioning of Reentrancy or other bugs.

After going through the output and analyzing them on tool-by-tool basis. Four tools, Slither, Conkas, Osiris and Mythril were found out to be able to detect popular DASP10 vulnerabilities. They were chosen to be compared with the values from the Table 6 to a graph to find the results visually. The color-coded part from table 6 was removed from the cell's value on Slither tool because of assumed anomaly in the data. The result can be seen in the following:

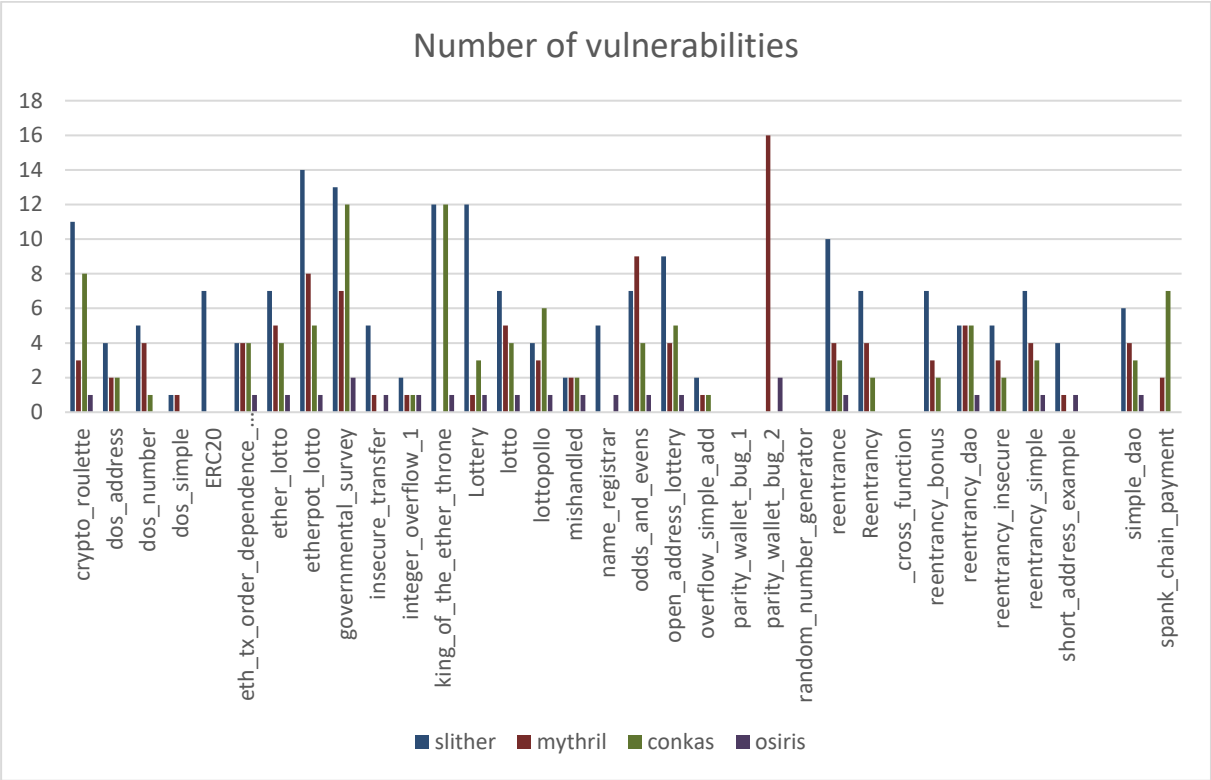


Figure 27-Results found by selected tools on each contract

6.2 Discussion of the results

Table 6 provides the data mapped to each contract used in the test datasets to the tools. Below will be discussion on the results of the tools based on the matrix and separate test-runs and the reasons for varying number of results to the different contracts.

Osiris was able to detect Reentrancy, overflow bugs, callstack bugs in the dataset. Osiris could not detect DOS address bugs, on ERC20 contract. However, in the Governmental Survey Contract, Osiris was able to detect, overflow bugs and time -dependency bugs. From the analysis of data, Osiris can be used to figure out Reentrancy, Callstack, Time Dependency and Overflow Bugs. The average time taken for Osiris is also relatively low.

SolHint has highest number of findings in the results. The reason for this is SolHint is a linter tool that checks for linting errors in the source code of the Solidity contracts. SolHint cannot be considered a true Symbolic Execution tool but helps Smart Contract developers to have a consistent formatting of their code and can be considered as a visualization Tool. Also, the average time taken by Solhint is also considerably low.

Honeybadger, on the other hand specializes in finding honey pots in the Ethereum contracts. In the test dataset, Honebadger was able to detect on three contracts: open address lottery, crypto lottery, odd and evens contracts hinting signs of honeypot. Honeypots in smart contract, which is created to look like a vulnerable contract, so the user can drain the contract if they send certain ether amount of contract first. It can be a great deal for scanning contracts that are designed by malicious developers with an intention to scam others. Honeybadger is relatively less time consuming.

Conkas, was able to detect different types of vulnerabilities in the contracts. Conkas was able to detect, Time Manipulations, Unchecked Low-Level Calls, Transaction Ordering Dependence, and Integer Overflow. Conkas for example in governmental survey contract, despite showing 12 vulnerabilities, they were repetition of the vulnerability it was detected mentioned earlier but on different line numbers. Conkas also suggests the line number in the contract where the vulnerabilities occur in the contract.

Maian is able to detect lock, prodigal and suicidal vulnerable contracts. In, Lotto contract Maian detected prodigal vulnerable while in Parity wallet 2 contract both suicidal and prodigal was true. No lock vulnerable contract was detected in the dataset. Maian also took relatively long time in comparison to other tools and the reason could be the algorithm to detect these vulnerabilities Maian can detect are computationally time consuming. Time performance did not feel satisfactory for the number of results produced.

Manticore, from the data matrix shows highest count of vulnerability (ten) in reentrance contract, but most of them are repeating in multiple lines, some of them as a suggestion of potential bugs or then warning. Manticore was good to detect reentrancy bug, gave suggestions about uninitialized storage, warning over Timestamp instructions, Invalid instructions, External calls to sender on mishandled contract. Manticore also showed line numbers and the code that the issue raised from. Manticore was also relatively time consuming too, but was good to suggest potential bugs, give warnings etc.

Smartcheck listed the data in different way compared to other tools. While Smartcheck was able to detect and show different problems on different line of the code and content, the result name was mostly non-verbose but rather presented as, Solidity Visibility or Solidity upgrade to 50, Solidity Revert

Require etc. Smartcheck's error name were not self-explanatory compared to other tools. Smartcheck also took the longest time in average compared to all other tools.

Mythril detected highest on Parity wallet 2 contract. The vulnerabilities that were found were Integer Overflow, exceptional state, Integer underflow in the Parity Wallet contract. Mythril was also able to detect call to external contract, Unchecked call return value and Transaction order dependence. The data showed line number, type of the issue and description. Mythril had highly verbose output and relatively low execution average time.

Securify was able to detect mostly from the categories of reentrancy and missing input validation. Securify did not have results on most of the contracts. On reentrany contract it had found the reentrancy vulnerabilities. Something new that was seen in securify compared to other contract was missing input validation on SimpleDao contract. Securify had one of the highest time consumptions on average.

Slither, had a highly verbose output, it also labelled the impact and confidence of the vulnerabilities, mapping to the line numbers etc. Slither was able to detect vulnerabilities of Re-entrancy, unchecked low level calls, time dependence and the labelling and overall performance and vulnerability coverage was great. The average time taken was relatively low.

Based on the above result of vulnerability mapping matrix, Silther, Mythril and conkas were able to cover maximum vulnerable contracts. There were some tools that took exceptionally long compared to other tools for example Smartcheck. The reason for time long time consumption could not be understood. Solhint specialized in other use cases like checking the contract syntax for solidity and Honebadger specialized in suicidal contracts.

Ethereum smart contract because of their relatively new style of programming and some design flaws of solidity, many vulnerabilities have been identified. Some of these vulnerabilities has led to financial losses, for example DAO hack, Parity wallet hack etc. Major vulnerabilities include Reentrancy, Call to unknown, suicidal contracts etc. Some of these contracts still might exist in the Blockchain, as once the vulnerable contract once deployed to blockchain will prevail forever for the lifetime of Ethereum Blockchain. Hopefully, someday these issues will be mitigated by some clever people designing and developing the blockchain code.

The existing vulnerabilities resulting from some security breaches and hacks are classified in DASP10 vulnerability lists, which will continue to evolve as more vulnerabilities will be found. Earlier researches such as [3] had come up with a vulnerability taxonomies and while other researches point to DASP 10 as the latest vulnerability classification. The result from the tools and their analysis through a common

tool SmartBugs is useful for smart contract developers to analyze their contracts, before the contract will be uploaded to the Blockchain where the vulnerable contract will exist forever. On top of that, tools like Smartbugs have removed the trouble for developers to worry about downloading many tools and run their code to scan for vulnerabilities. The result suggests that each tools have their own peculiarity and needs to be used to cover as many vulnerabilities as possible in the smart contract and to keep the financial aspect of the smart contracts on Blockchain secured.

Apart from the tools that exist to detect vulnerabilities can also be used by malicious attackers to scan vulnerable smart contract, which will continue to exist on the Ethereum Blockchain. This can bring, and has brought, financial losses to the people who have invested in the Blockchain. This stays a major issue for Ethereum blockchain, as this is a double-edged sword to the security that Blockchain provides on the integrity of the data and public ledger, while leaving the vulnerable contracts to exist forever. Probably, someday there will be found some smart solution to tackle these issues and hopefully these tools will help to bring safe secure Smart contracts in the future. From the test following conclusion was drawn by the researcher:

Tool	Usefulness as a quick vulnerability checking tool	Verbosity	Time
Conkas	Can detect some vulnerability from DASP 10 like Reentrancy, LL calls, Integer Overflow. Useful as a quick command line tool	Good and human-readable	Low
Mythril	Covers many vulnerabilities of DASP 10 and useful tool	Output not easily human-readable. Json parsing needed	Low
Osiris	Able to detect popular DASP vulnerabilities and usable as a quick command line tool	Very verbose output and easy to grasp	Low
Slither	Detects popular DASP 10 and useful as a command line tool too.	Verbosity Ok but color-coding for warning, info and errors helful	Low
Oyente	Detects popular DASP 10 vulnerabilities and useful as a quick command line tool	Very readable output. True/false labelling of vulnerabilities	Low
Solhint	Useful as a linter but not to detect vulnerabilities	Ok	Low
Smartcheck	No output on the test run	No output in the test run	Exceptionally high
Honeybadger	Good tool for detecting honeypots	Verbose and true false like output. Easily Readable	Low
Manticore	Detects popular DASP 10 vulnerabilities. Useful as a quick command line tool too	Verbose but complicated to read and huge output	High
Maian	Good tool for detecting suicidal, prodigal and lock vulnerable contracts	Verbose	High
Securify	No output on test run	No output on test run	Low

Table 7- Viewpoint from the experiment

7 Conclusion

Blockchain has been pondered by many institutions as a revolutionary technology that will bring potential benefits to industries, governance, and financial institutions. The concept of blockchain provides a possibility of decentralized or governed model of doing transactions of many sorts. Bitcoin initially created the hype because of massive surges in price and making early bearers and investors a huge gain. Ethereum, as they claim, came up with a new revolutionary concept of Turing Complete programmable blockchain giving birth to plethora of cryptocurrencies. HLF joined the league as an industrial blockchain. Many other blockchain technologies and concepts have been born now. Blockchain primarily using cryptography and other algorithms of computer science provides integrity and confidentiality of data where none of the peers in the network or decentralized system can be trusted. However, any system is vulnerable, and thus is blockchain.

In this research, an overview related to blockchain including the history, uses of cryptography, consensus algorithm was thoroughly researched based on other literature and articles, thus providing a ground up information. Security vulnerability in blockchain as-a-whole was presented. Empirically, the major focus of research was Ethereum Smart contract and the vulnerabilities that has led to massive financial losses and tools available for analyzing vulnerabilities. In the research, an introduction to smart contract, vulnerabilities classified by research literatures was reviewed and analyzed. Also, tools available to detect the vulnerabilities was studied and presented. An empirical study of ten popular vulnerability detection tools were studied with the help of automation tool called SmartBugs. The study found out that SmartBugs is well suited tool for developers for choosing many tools in one place and do an automated test to find vulnerabilities in Smart contracts. However, these tools cannot prevent vulnerable contract that are already deployed. The tools were compared against time consumption and the amount of detection they can do on a curated test data set of DASP10 vulnerabilities.

The conclusion from the research is that tools like SmartBugs and other symbolic execution tools will help find out vulnerable contracts at the time of development, but they will not help mitigate the security issue once the contracts are deployed to Ethereum. Hopefully, Ethereum team will figure out a solution someday.

References

[1] J. S. Kat Tretina, “Top 10 Cryptocurrencies In October 2021.” https://www.forbes.com/advisor/investing/top-10-cryptocurrencies/.

[2] Buterin and Vitalik, “Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform,” *Etherum*, no. January, pp. 1–36, 2014, [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper.

[3] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on Ethereum smart contracts (SoK),” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 10204 LNCS, no. July, pp. 164–186, 2017, doi: 10.1007/978-3-662-54455-6_8.

[4] A. Dika, “Ethereum Smart Contracts: Security Vulnerabilities and Security Tools,” *(Master’s thesis, NTNU)*, no. December, 2017, [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2479191/18400_FULLTEXT.pdf%0Ahttp://hdl.handle.net/11250/2479191.

[5] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck, “Blockchain,” *Bus. \& Inf. Syst. Eng.*, vol. 59, no. 3, pp. 183–187, 2017.

[6] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, “Blockchain challenges and opportunities: A survey,” *Int. J. Web Grid Serv.*, vol. 14, no. 4, pp. 352–375, 2018.

[7] S. Haber and W. Scott Stornetta, “How to time-stamp a digital document,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 537 LNCS, pp. 437–455, 1991, doi: 10.1007/3-540-38424-3_32.

[8] D. Bayer, S. Haber, and W. S. Stornetta, “Improving the Efficiency and Reliability of Digital Time-Stamping,” *Seq. II*, pp. 329–334, 1993, doi: 10.1007/978-1-4613-9323-8_24.

[9] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” 2008, [Online]. Available: https://bitcoin.org/bitcoin.pdf.

[10] S. Aggarwal and N. Kumar, “Hyperledger☆,” *Adv. Comput.*, vol. 121, pp. 323–343, 2021, doi: 10.1016/bs.adcom.2020.08.016.

[11] M. Di Pierro, “What is the blockchain?,” *Comput. Sci. \& Eng.*, vol. 19, no. 5, pp. 92–95, 2017.

[12] G. Wood, “Ethereum: a secure decentralised generalised transaction ledger,” *Ethereum Proj. Yellow Pap.*, pp. 1–32, 2014.

[13] F. Tschorsch and B. Scheuermann, “Bitcoin and beyond: A technical survey on decentralized digital currencies,” *IEEE Commun. Surv. Tutorials*, vol. 18, no. 3, pp. 2084–2123, 2016, doi: 10.1109/COMST.2016.2535718.

[14] R. Zhang, R. Xue, and L. Liu, “Security and privacy on blockchain,” *ACM Comput. Surv.*, vol. 52, no. 3, 2019, doi: 10.1145/3316481.

[15] Everett Muzzy, “What Is Proof of Stake?,” 2020. https://consensys.net/blog/blockchain-explained/what-is-proof-of-stake/.

[16] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, “Smart contracts vulnerabilities: A call for blockchain software engineering?,” *2018 IEEE 1st Int. Work. Blockchain Oriented Softw. Eng. IWBOSE 2018 - Proc.*, vol. 2018-Janua, no. March, pp. 19–25, 2018, doi: 10.1109/IWBOSE.2018.8327567.

[17] L. Brent *et al.*, “Vandal: A Scalable Security Analysis Framework for Smart Contracts,” pp. 1–28, 2018, [Online]. Available: http://arxiv.org/abs/1809.03981.

[18] M. Saad *et al.*, “Exploring the Attack Surface of Blockchain: A Systematic Overview,” pp. 1–30, 2019, [Online]. Available: http://arxiv.org/abs/1904.03487.

[19] K. Wang, Y. Wang, and Z. Ji, “Defending Blockchain Forking Attack by Delaying MTC Confirmation,” *IEEE Access*, vol. 8, pp. 113847–113859, 2020, doi: 10.1109/ACCESS.2020.3000571.

[20] M. Rosenfeld, “Analysis of Hashrate-Based Double Spending,” pp. 1–13, 2014, [Online]. Available: http://arxiv.org/abs/1402.2009.

[21] Don Yu, “Coinbase’s perspective on the recent Ethereum Classic (ETC) double spend incidents,” 2020. https://blog.coinbase.com/coinbases-perspective-on-the-recent-ethereum-classic-etc-double-spend-incidents-1fd19ef215f3.

[22] M. Apostolaki, A. Zohar, and L. Vanbever, “Hijacking Bitcoin: Routing Attacks on Cryptocurrencies,” *Proc. - IEEE Symp. Secur. Priv.*, pp. 375–392, 2017, doi: 10.1109/SP.2017.29.

[23] A. Greenberg, “Hacker Redirects Traffic From 19 Internet Providers to Steal Bitcoins,” 2014. https://www.wired.com/2014/08/isp-bitcoin-theft/.

[24] E. Heilman, A. Kendler, A. Zohar, T. Hebrew, M. S. R. Israel, and S. Goldberg, “Eclipse Attacks on Bitcoin ’ s Peer-to-Peer Network This paper is included in the Proceedings of the,” *SEC’15 Proc. 24th USENIX Conf. Secur. Symp.*, pp. 129–144, 2015, [Online]. Available: https://www.usenix.org/node/190891.

[25] M. Saad, M. T. Thai, and A. Mohaisen, “POSTER: Deterring DDoS attacks on blockchain-based cryptocurrencies through mempool optimization,” *ASIACCS 2018 - Proc. 2018 ACM Asia Conf. Comput. Commun. Secur.*, pp. 809–811, 2018, doi: 10.1145/3196494.3201584.

[26] B. E. Guide, “Report: Bitcoin (BTC) Mempool Shows Backlogged Transactions, Increased Fees if so?,” 2018. .

[27] M. Saad, A. Khormali, and A. Mohaisen, “End-to-End Analysis of In-Browser Cryptojacking,” 2018, [Online]. Available: http://arxiv.org/abs/1809.02152.

[28] Vitalik Buterin, “Hard Fork,” 2016. .

[29] B. C. Gupta, “Analysis of Ethereum Smart Contracts - A Security Perspective,” no. May, 2019.

[30] D. Spiegel, “Understanding the dao attack.” .

[31] Santiago Palladino, “The Parity Wallet Hack Explained.”

[32] A. Pedro Cruz Monteiro, J. Fernando Peixoto Ferreira, and J. António Madeiras Pereira Supervisor, “A Study of Static Analysis Tools for Ethereum Smart Contracts Information Systems and Computer Engineering Examination Committee,” no. October, 2019.

Appendix 1: Own Code

Code added to Smartbugs:

```
dict = {}
def calculate_average_time(tool,time):
    if tool in dict:
        dict[tool].append(time)
    else:
        dict[tool] = [time]

    print('\n----Average Time and Total calculations----------------------------------\n')
    for k,v in dict.items():
        print("Tool: "+k+", Total calculations: "+str(len(v))+", Average Time: "+str(sum(v)/len(v))+" seconds")
    print('\n-----------------------------------------------------------------------------\n')
```