Analysis of Ethereum Smart Contracts - A Security Perspective

A thesis submitted in partial fulfillment of the requirements

for the degree of Master of Technology

by

Bishwas C Gupta

17111011

under the guidance of

Prof. Sandeep K. Shukla

to the

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

May 2019

Statement of Thesis Preparation

1. Thesis Title: Analysis of Ethereum Smart Contracts - A Security Perspective
2. Degree for which thesis is submitted: Master of Technology
3. The "Thesis Guide" was referred to for preparing the thesis.
4. Specifications regarding thesis format have been closely followed.
5. The contents of the thesis have been organized based on the guidelines.
6. The thesis has been prepared without resorting to plagiarism.
7. All sources used have been cited appropriately.
8. The thesis has not been submitted elsewhere for a degree.

Bishwas Gupta

Name: Bishwas C Gupta

Roll No.: 17111011

Department of Computer Science and Engineering

May 2019

CERTIFICATE

It is certified that the work contained in the thesis titled **"Analysis of Ethereum Smart Contracts - A Security Perspective"** has been carried out under my supervision by **Bishwas C Gupta** and that this work has not been submitted elsewhere for a degree.

Prof. Sandeep K. Shukla
Professor
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
Kanpur, 208016.

May 2019

ABSTRACT

Name of student: **Bishwas C Gupta** Roll no: **17111011**

Degree for which submitted: **Master of Technology**

Department: **Department of Computer Science and Engineering**

Thesis title: **Analysis of Ethereum Smart Contracts - A Security Perspective**

Name of Thesis Supervisor: **Prof. Sandeep K. Shukla**

Month and year of thesis submission: **May 2019**

Ethereum is the second most valuable cryptocurrency, just after Bitcoin. The biggest difference between Bitcoin and Ethereum is the ability to write smart contracts - small programs that sit on the blockchain. As the contracts are on the blockchain, they become immutable making them attractive for various decentralised applications (or dApps) like e-governance, healthcare management and data provenance.

However, the biggest advantage of smart contracts - their immutability also poses the biggest threat from a security standpoint. This is because any bug found in the smart contract after deployment cannot be patched. Recent attacks like the DAO attack and the Parity attack have caused massive monetary losses. In such a scenario it becomes imperative to develop and interact with smart contracts that are secure.

In this thesis we analyze the Ethereum Smart Contracts from a security viewpoint. We present a study of the security vulnerabilities observed in Ethereum smart contracts and develop a novel taxonomy for the same. We then analyse the different security tools available. For this, we create a vulnerability benchmark – a set of 180 vulnerable contracts across different categories identified in the taxonomy. The results of the tools on this benchmark are analysed to help developers and end-users make an informed decision about which tool to use depending on their use-cases.

We further collect byte-codes for 1.9M smart contracts from the main Ethereum blockchain and analyse them on various parameters like duplicity, total ether balance, etc. We observe that a small fraction of contracts dominate the others on every parameter we analysed. These 2900 contracts are identified as 'Contracts of Importance' and are further analysed using the tools available to gain valuable insights into the insecurity patterns and trends in Ethereum smart contracts.

Dedicated to Baba

Acknowldgements

I would like to express my sincere gratitude to Prof. Sandeep K. Shukla for his constant support and guidance. Blockchain Technology was a new topic for me and his critical and useful suggestions during the course of this thesis were extremely beneficial.

I am thankful to my lab-mates and class-mates for listening to my doubts and providing me with constructive feedback.

I am eternally grateful to Gracious Huzur Prof. Prem Saran Satsangi Sahab for His continued guidance, grace and protection.

I am grateful to my parents CA Ameer Chand Gupta and Dr. Sharmita Gupta for their constant support, encouragement and faith throughout my life. Thanks are also due to my sister Mrs. Chetnaa Dua, my brother-in-law Mr. Sujan Dua and my nephew Master Suchal Dua for always being there for me and for cheering me up whenever I was down.

Special thanks are due to Miss Sheena Bhatnagar for her ever-growing love and support and also for putting up with me and my quirks for the last six years.

Bishwas C Gupta

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Smart Contracts are one of the most promising features of blockchain technology and have created a lot of buzz over the years. They are essentially small computer programs that exist on the blockchain. This makes them immutable as long as the blockchain's integrity is not compromised. Therefore, the end-users can be sure that the program has not been changed or manipulated. This provides the people with trust in a distributed environment where they do not have to trust the other nodes or a centralised third party.

Ethereum is the second most valuable crypto-currency, just after Bitcoin [2]. However, the most distinguishing feature of Ethereum was the introduction of smart contracts. This has fostered a new area of software development called decentralized applications (or dApps). These are applications that utilize the features of blockchains (immutability and lack of a central authority) in the back-end and combine them with user friendly front-ends for non-technical users. The development of dApps for different application areas is an active area of research with the most popular ones being decentralised identity management, land record management, e-governance systems, etc. Essentially, blockchains help in maintaining a tamper-proof log that is used to establish ownership of actions. Therefore, it has wide applications and can help establish the trust of the users in the system.

Unfortunately, the most promising feature of smart contracts - *immutability* poses unique challenges from a security and a software engineering standpoint. Traditional software development life cycle (SDLC) is an iterative process where new features are added over time. Also, security issues and bugs found are fixed by releasing patches. However, as smart contracts are immutable they cannot be changed or fixed once they are deployed on the blockchain. Since Ethereum is a public blockchain, all the data is available in the public domain for hackers and other malicious actors to exploit.

The issue of smart contract security is not an academic one. Since Ethereum is a crypto-currency backed blockchain, hackers have a monetary incentive to find buggy contracts and exploit them. Over time we have seen many attacks happen on Ethereum smart contracts. The most popular attack was the re-entrancy attack on the Decentralised Autonomous Organisation (DAO). The DAO was a high value contract with its token sale setting the record for the largest crowdfunding campaign [3]. Unfortunately, because of the re-entrancy bug, attackers were able to siphon Ether (the native crypto-currency of Ethereum) worth US$50M at the time. The attack was so massive that it caused the Ethereum community and the blockchain itself to split into two - Ethereum (which reverted the effects of the attack) and Ethereum Classic (which carried on after the attack). In 2017, hackers stole US$30M from the Parity mutisig wallet because of function default visibility and unchecked external call bugs. They would have stolen more, however white hat hackers *hacked* the remaining wallets and saved around US$150M [4].

Also, it has been observed that many smart contract variants of Ponzi schemes and pyramid schemes have come up on the Ethereum blockchain. Even though, the community is actively spreading awareness, an unsuspecting user might get trapped in such schemes and loose their hard earned money.

The main goal of this thesis is to serve as a reference for smart contract developers and smart contract end-users on the various security vulnerabilities, the tools available and the type of on-chain smart contracts they might encounter so that they can be secure on this anonymous and trust-less network.

The thesis is divided as follows – Chapter 2 introduces the concepts of blockchains using Bitcoin. Then we look at the Ethereum blockchain and Smart Contracts. Finally, we look at the related work that has been done in the past. Chapter 3 looks at the various security vulnerabilities in Ethereum Smart Contracts and introduces a new taxonomy for classifying the same. Chapter 4 introduces the various security tools available for smart contract developers and demonstrates their performance on a newly created Vulnerability Benchmark. Chapter 5 talks about our on-chain smart contract data collection, its analysis across different parameters which led us to identify the *Contracts of Importance* and the results of the security tools on these contracts. We conclude with Chapter 6 with directions for future research work.

Chapter 2

Background

2.1 Blockchain Introduction

2.1.1 Bitcoin Introduction

The first (and arguably the most popular) blockchain is Bitcoin - the brainchild of the pseudonym Satoshi Nakamoto [5]. The main motive behind Bitcoin was to serve as a decentralised and distributed global currency - without any centralised banks.

The ledger of transactions is stored as a chain (or a linked list) of blocks but the pointers connecting these blocks are hash pointers which are dependent on the block information. The block information does not store the transaction directly. Each transaction becomes the leaf node of a Merkle tree, and the root of the Merkle tree is included in the block information. Therefore, any attempt to manipulate the data would not be possible without breaking (or forking) the chain.

Also, as we are in distributed systems territory, we encounter one of the most famous problems of consensus, represented by the Byzantine General's Problem [6]. Essentially in a distributed system which is prone to Byzantine faults, how do we ensure that every node maintains the same ledger of transactions i.e. all the nodes are in consensus about the contents of the ledger? Bitcoin introduced an approach called proof of work (explained later in this section) to deal with it. However, other consensus algorithms like Proof of Stake and Proof of Burn, etc. are also used in other blockchains.

Finding the answer to the cryptographic hash puzzle (proof of work for Bitcoin) is called mining. This is because whoever finds the answer gets a block reward in bitcoin (essentially creating or mining new bitcoin).

2.1.2 Bitcoin Features

The fundamental features on which Bitcoin (and other early blockchain platforms) was built upon are –

1. **Distributed and Public** – It is a distributed peer-to-peer (P2P) network hosted over the internet. Anyone with an internet connection can join the network, view the blockchain or become a miner.

2. **Decentralized** – There is no central authority (like banks) in the network. This means that transactions cannot be reversed and there is little chance of grievance redressal.

3. **Consensus among nodes** – All the nodes agree to the state of the blockchain. There might be temporary *soft* forks along the way but eventually only the longest chain survives.

4. **Cryptographically Secure and Immutable** – The blocks are linked by hash pointers which are computationally very expensive to find. Therefore blockchain contents are more or less immutable. The deeper the block is in the chain, the more difficult and resource intensive it becomes to manipulate transactions in that block.

5. **Pseudo-anonymous** – There are no unique identifiers on the blockchain. Everyone is known by their address. Also, there is no restriction on the number of accounts a person can have. This had earlier lead to Bitcoin becoming a hub for illegal activities as it provided people with 'digital cash'.

6. **Easily Verifiable** – Since the blockchain data is publicly visible and known to all, anyone can view and verify transactions on the chain.

7. **Limited Supply** – The supply of bitcoins is limited to 21M. After this no new bitcoins can be mined and miners will stop getting the block rewards and only receive the transaction fees.

2.1.3 Consensus Algorithms

The Consensus Problem occurs when different nodes in a distributed system need to come to an agreement in the presence of malicious nodes or faulty communication channels. [7] It is a very popular problem in distributed systems and Nakamoto gave an ingenious solution to it with proof of work. However, over the past decade different algorithms to solve this problem have come up with different blockchain platforms. Here we explain some of the popular consensus algorithms in use -

Proof of Work (PoW)

This approach was introduced in Bitcoin. Essentially, the miner (the person doing the *work*) has to find a value (also known as the nonce) which after being concatenated with the block contents and taking the hash gives a value which starts with a fixed number of zeroes (also known as the difficulty). The solution to finding this value is brute-force which is computationally very expensive. Therefore if anyone has to change the blockchain contents, he/she has to calculate the nonce value for that block and all the blocks after it up to the current block. This is highly infeasible.

Since the work requires high computational power, the nodes with higher computational power become the miners and control the blockchain. However if a miner (or a group of miners) have more than 51% of the hash power of the network, they essentially control what goes into the blockchain and the integrity is compromised. Also, the engergy consumption of PoW blockchains is extremely high.

Proof of Stake (PoS)

The concept of proof of stake is very similar to share holder voting in any company. Who ever owns the most stake (or the most coins) gets the privilege to mine the next block. Some blockchains also define stake in terms of the amount a time a person holds a coin (also referred to as the coin-age). Ethereum's Casper Protocol is a Proof of Stake Protocol.

A 51% attack is generally considered more expensive in case of a proof of stake chain. However, many people are opposed to it ideologically as the it gives the 'rich' control over the blockchain. Also, there is a problem of 'nothing-at-stake' where a malicious miner looses nothing by betting on two different forks of the chain.

Delegated Proof of Stake (dPoS)

In delegated proof of stake, the stake-holders elect a delegate by means of a weighted election (the weights are proportional to their stake in the network). This delegate verifies the transactions, makes a block and receives a block for that.

This system is usually considered to be one of the fasted consensus algorithm that can scale up-to millions of transactions per second and is used by the EOS blockchain.

Delegated Byzantine Fault Tolerance (dBFT)

This consensus mechanism was made popular by the cryptocurrency NEO. This system is very similar to a democratic system. Anyone who holds the cryptocurrency becomes a citizen. To become a delegate the node needs to satisfy certain requirements like a good internet connection, specific equipment, etc. The citizens vote for delegates. One of the delegates is randomly chosen as the speaker. The speaker then proposes a new block which is verified by the other delegates. If 66% of the delegates accept the block, it is included in the blockchain. Otherwise another delegate is chosen as the speaker and the process is repeated.

There are also other consensus algorithms like Proof of Importance, Proof of Burn, Proof of Activity, etc. It is an active area of research and new variants keep coming up with new blockchains.

2.1.4 Blockchain Evolution over time

Blockchain 1.0

It all started with Bitcoin, and quickly spread to other crypto-currencies as well. All the first generation blockchains were created in an attempt to create a decentralised digital payment system and for this different crypto-currencies were made on models similar to Bitcoin.

Blockchain 2.0

This generation started with Vitalik Buterin and his vision for Ethereum to run smart contracts which are essentially small programs that are present on the blockchain. Since they are on the blockchain, they become tamper-resistant.

Blockchain 3.0

This generation further extended the ideas of the previous generation with the introduction of decentralised applications (or dApps). These are full fledged applications with a front-end and back-end, however the back-end of these applications usually resides on a blockchain. This generation also introduced permissioned blockchains which place restrictions on who can join the network, who can mine the new blocks, etc. One of the most prominent blockchain in this generation is IBM Hyperledger.

2.1.5 Types of Blockchains

There are three main types of blockchains –

Public Blockchain

In a public blockchain, anyone can join the network, transact, and become a miner. Also, anyone can see and parse the contents of such blockchains. Bitcoin and Ethereum are popular examples of this category.

Private Blockchain

It is an invite-only blockchain. It is usually hosted on private networks by enterprises who do not wish anyone to view or modify the data on the blockchain. Unlike public blockchains, it is usually centralised however the data is still cryptographically secured from the company's view point. Multichain is an example of a private blockchain

Federated or Consortium Blockchain

It is mainly used by banks. In this the consensus is controlled by a pre-determined set of nodes. The right to read the blockchain may be restricted or open. Corda is an example in this category.

2.1.6 Blockchain Applications

Blockchain systems provide attractive properties like immutability, consensus, trust in a trust-less world, decentralisation, etc. These coupled with the fact the developers can write small programs called Smart Contracts that reside on the chain has made way for a whole new area of software development and research called decentralized applications (or dApps). They are applications that leverage the properties of a blockchain system to give solutions for real world problems.
Finding new areas for blockchains and distributed ledger technology is an active area of research. Money transactions is an obvious application for all the cryptocurrency backed blockchain platforms. Blockchains have been used for e-governance, digital identity management, land record registry, etc. Supply chain and proof of provenance is also an active area of research.

2.2 Ethereum Primer

2.2.1 Introduction to Ethereum

Ethereum is also a cryptocurrency backed blockchain like Bitcoin. It uses similar techniques like proof of work (it will eventually move to a proof of stake based consensus algorithm called Casper), hash pointers (Ethereum uses KECCAK-256), etc. However, the main difference between Ethereum and Bitcoin is that unlike Bitcoin which is just a distributed ledger of transactions, Ethereum can also run small computer programs which allow developers to develop decentralized applications (or dApps). Also, unlike Bitcoin whose founder(s) are unknown, Ethereum is the vision of Vitalik Buterin, who wrote the white paper [8]. It is maintained by the Ethereum Foundation.
Unlike Bitcoin, Ethereum has two kinds of addresses [9] –

1. **Externally Owned Accounts (EOAs)** – these accounts are owned through public-private key pairs.

2. **Contract Accounts** – these are special accounts which are controlled by the smart contract deployed on them. They can be triggered only by an EOA.

Like in Bitcoin, the users have to pay a small transaction fees for each transaction they want to be entered on the blockchain. This is paid in Ethereum's native currency called Ether.

2.2.2 Smart Contracts

Smart Contracts are essentially small programs that exist on the blockchain and are executed by the Ethereum Virtual Machine (EVM). They do not need any centralised trusted authority like banks since all the functionality required is implemented in the smart contract logic, and since the code itself resides on the blockchain we can be sure that it has not been tampered with. This property of being immutable is crucial in financial applications like escrow and other payments. Also, it allows developers to develop other smart applications by utilizing the power of blockchain technology.
However, the concept of smart contracts is not new. It was introduced by Nick Szabo [10] in 1997.

2.2.3 EVM

EVM stands for Ethereum Virtual Machine which serves a similar purpose that Java Virtual Machine (JVM) does for Java by providing a layer of abstraction between the code and the machine. This also makes the code portable across machines. It also gives the developers an option to code in their smart contract language of choice, as finally all the programs written in different languages are translated by their respective compilers to EVM byte-code.
The Ethereum Yellow Paper [11] explains the intricate workings of the Ethereum Virtual Machine in great detail. The EVM has 140 opcodes which allow it to be Turing complete. Each opcode takes 1 byte of storage space. The EVM also uses a 256 bit register stack which holds 1024 items. It also has a contract memory for complicated operations but it is non persistent. For storing data indefinitely, storage is used. Reading from storage is free, but writing to storage is extremely expensive.

Smart Contract Programming

The most popular programming language for Ethereum Smart Contracts is Solidity. It is a language similar to Javascript and C++, making it easy for existing software developers to write solidity code. Other languages, though not as popular are Vyper and Bamboo. Before Solidity was released, languages like Serpent and Mutan were used which have since been deprecated. [12]
The compiler (solidity's compiler is called `solc`) converts the source code to EVM bytecode. This code is called the contract creation code. This is like a constructor to put the contract bytecode

on the blockchain and can be executed by the EVM only once to put the run-time bytecode on the chain. The run-time bytecode is the code that is executed by the EVM on every call the contract. The run-time bytecode also contains a swarm hash of the metadata file. This file can contain information like functions, compiler version, etc. However, this is still an experimental feature and not many have uploaded the metadata to the Swarm network[12]. Figure 2.1 explains this process graphically.

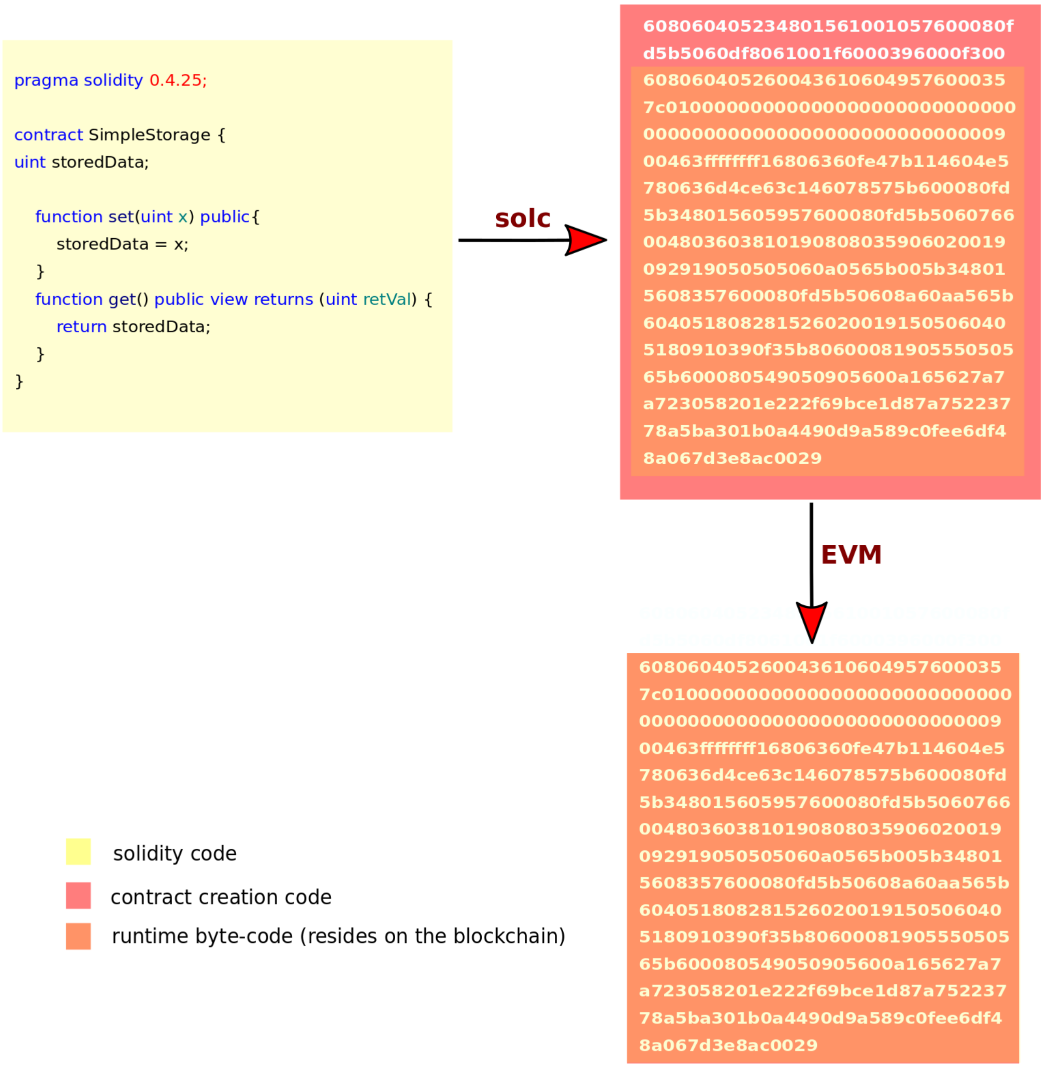

Figure 2.1: How solidity code is put on the blockchain

Ether and Gas

Ether is the native cryptocurrency of the Ethereum network.

Gas is another feature of Ethereum that separates it from Bitcoin. Since different smart contracts require varying amounts of computational power and time, it would be unfair to the miners to base the transaction fees just on the length of the transaction or have a constant transaction fees. Gas

is a unit introduced by Ethereum that measures the computational work done. Each operation has an associated gas cost. However, gas is different from Ether as the value of the latter is market dependent but that does not change the 'computational power' required to execute the contract. Therefore, every transaction mentions a gas price which is the price a person is willing to pay in ether per unit of gas. The combination of these two give the transaction fees in ether.
The concept of gas introduces two new scenarios –

1. Transaction running out of gas – if the gas costs go beyond the value allotted, the transaction is marked as failed and is included in the blockchain. This also prevents miners getting stuck in infinite loop computations.

2. Gas price too low/high – if the gas price is too low, no miner will pick up the transaction. On the other hand, if it is too high then it will become very expensive for the sender of the transaction.

2.3 Existing Related Work

- Atzei et. al. [1] conducted the first survey of attacks on Ethereum smart contracts and also gave the first taxonomy of Ethereum smart contract vulnerabilities. They also look at some of the popular vulnerable contracts like the DAO, Rubixi, GovernMental and King of the Ether throne.

- Dika [13] in his master's thesis, extended the taxonomy given by Atzei et. al. [1]. He also tested the effectiveness of three security tools on a data-set of 23 vulnerable and 21 safe contracts. It is observed that the data-set and the number of tools used for the study is quite less. Also, the taxonomy needs hierarchy for better analysis.

- Mense et. al. [14] look at the security analysis tools available for Ethereum smart contracts and cross reference them to the extended taxonomy given by Dika [13] to identify the vulnerabilities captured by each tool. However, the tool's effectiveness in catching those vulnerabilities is not studied.

- Buterin [15] in his post outlines the various vulnerable smart contracts with an elementary categorization. He also emphasises the need to experiment with various tools and standardization wherever possible to mitigate bugs in smart contracts.

- Angelo et. al. [16] surveyed the various tools available to Ethereum smart contract developers. They do a very broad categorization of tools - those which are publicly available and those which are not publicly available.

- Antonopoulos et. al. [17] in their book on Ethereum have dedicated a chapter on smart contract security. They cover the various vulnerabilities encountered by smart contract developers and give real world examples and preventative techniques. It is a good reference for smart contract developers.

Chapter 3

Security Vulnerabilities in Ethereum Smart Contracts

3.1 Study of Ethereum Security Vulnerabilities

We divide the security vulnerabilities in Ethereum into two broad categories - Blockchain 1.0 and Blockchain 2.0 vulnerabilities.

Blockchain 1.0 vulnerabilities includes security vulnerabilities that are present in most blockchain based systems and that Ethereum shares with it's predecessors like Bitcoin.

Blockchain 2.0 vulnerabilities include vulnerabilities introduced in the system because of the presence of smart contracts.

Our work is concerned with the Blockchain 2.0 (Smart Contract) Vulnerabilities. However, the Blockchain 1.0 vulnerabilities are introduced for completeness.

3.1.1 Blockchain 1.0 Vulnerabilities

51% Attack

In proof of work, the miners try finding the nonce value to solve the given cryptographic puzzle. However, if miner(s) get control of more than 51% of the compute power in the network then they essentially control what goes into the blockchain - compromising its integrity [18].

Double Spending

Double spending occurs when the attacker uses the same cryptocurrency more than once. This is done by leveraging race conditions, forks in the chain, or 51% attacks. A variant of this attack is the Finney attack [19]. Such attacks have been shown on Bitcoin [20].

Selfish Mining

In selfish mining, a malicious miner does not publish the block immediately after solving the proof of work puzzle. Instead, reveals it only to its pool members which then work on the next block, while the other network continues working for essentially nothing [21]. This was first demonstrated on Bitcoin [22] [23], and recently on Ethereum as well [24].

Eclipse Attack

In eclipse attacks, the victim's incoming and outgoing connections are taken over by attacker (using a botnet or otherwise). This gives a filtered view of the blockchain to the victim. Several other attacks like selfish mining and double spending can be launched at the victim. Such attacks have been demonstrated on Bitcoin [25].

BGP Hijacking Attack

The border gateway protocol (BGP) is used for handling routing information over the internet. BGP hijacking is a common attack. However in the context of public blockchains, it can be used to create unwanted delays in the network and cause monetary losses to the miners. Such attacks have been demonstrated on Bitcoin [26].

Private key security

Private keys are used to control the addresses in Ethereum. It is a security problem in itself to store these keys securely. As blockchain is a decentralised system, there is no way to report a stolen private key and prevent its misuse.

3.1.2 Blockchain 2.0 Vulnerabilities

Re-entrancy

A re-entrancy condition is when a malicious party can call a vulnerable function of the contract again before the previous call is completed: once or multiple times. This type of function is especially problematic in case of payable functions, as a vulnerable contract might be emptied by calling the payable function repeatedly. The `call()` function is especially vulnerable as it triggers code execution without setting a gas limit.

To avoid re-entrancy bugs, it is recommended to use `transfer()` and `send()` as they limit the code execution to 2300 gas [27]. Also, it is advised to always do the required work (i.e. change the balances, etc.) before the external call.

The DAO Attack is the most famous re-entrancy attack which lead to a loss of US$50 Million [28] and resulted in the chain being forked into two - Ethereum and Ethereum Classic.

In the example shown below from [28], the vulnerability arises from the fact the the user balance is not set to 0 until the very end of the function, allowing a malicious user to withdraw funds again and again.

```
mapping (address => uint) private balances;

function withdraw() public {
        uint amt = balances[msg.sender];
        require(msg.sender.call.value(amt)());
        balances[msg.sender] = 0;
}
```

Transaction Order Dependence (Front Running)

The order in which the transactions are picked up by miners might not be the same as the order in which they arrive. This creates a problem for contracts that rely on the state of the storage variables. Gas sent is usually important as it plays an important role in determining which transactions are picked first. A malicious transaction might be picked first, causing the original transaction to fail. This kind of race-condition vulnerability is referred to as transaction order dependence.

Overflows and Underflows

Solidity can handle up to 256 bit numbers, and therefore increasing (or decreasing) a number over (or below) the maximum (or minimum) value can result in overflows (or underflows). It is recommended to use OpenZeppelin's `SafeMath` library to mitigate such attacks.

Timestamp Dependence

A lot of applications have a requirement to implement a notion of time in their applications. The most common method of implementing this is using the `block.timestamp` either directly or indirectly. However, a malicious miner with a significant computational power can manipulate the timestamp to get an output in his/her favour.

Forcing Ether to a Contract

Usually, when you send ether to a contract, it's fallback function is executed. However, if the transfer of ether happens as a result of a `selfdestruct()` call, then the fallback is not called. Therefore, a contract's balance should never be used in an `if` condition as it can be manipulated by a malicious user.

In the example below from [28], we observe that the payable function always reverts, therefore 'something bad' should not happen. However, it is possible to force ether to a contract as shown before, invalidating the balance check.

```
contract Vulnerable {
        function () payable {
                revert();
        }

        function somethingBad() {
                require(this.balance > 0);
                // Do something bad
        }
}
```

Bad Randomness

Online games and lotteries are common dApp use cases. For these applications, a common choice for the seed of the random number generator is the hash or timestamp of some block that appears in the future. This is considered secure as the future is unpredictable. However, a malicious attacker can bias the seed in his favour. Such attacks on Bitcoin have already been demonstrated.

Short Address Attack

It is an input validation bug that was discovered by the Golem Team [29]. This allows the attacker to abuse the `transfer` function. The EVM automatically pads zeroes if the length of the address is less than the required length. This makes certain addresses with trailing zeroes vulnerabilities if proper sanity check is not done.

Unprotected Ether Withdrawal

Due to missing or inadequate access control mechanisms, it might be the case that anyone is able to withdraw Ether from the contract which is highly undesirable.

Authorization through `tx.origin`

This can be interpreted as a type of a phishing attack. In solidity, `tx.origin` and `msg.sender` are separate. The account calling a contract is defined by `msg.sender`. `tx.origin` is the original sender of the transaction, which might lead to a string of other calls. However, if `tx.origin` is used for authorization, and the actual owner is conned to call a malicious contract which in turn calls the victim contract, then the authorization fails.

In the example given below [30], the `owner` is set to the contract creator. However, if the contract creator is conned into calling a malicious contract which in turn calls the `sendTo` function, the authorization passes and the attacker is able to siphon funds from the contract.

```
contract Vulnerable{
    address owner;
    function Vulnerable() public{
        owner = msg.sender;
    }
    function sendTo(address receiver, uint amount) public{
        require(tx.origin == owner);
        receiver.transfer(amount);
    }
}
```

Unprotected `selfdestruct`

`selfdestruct` kills a contract on the blockchain and send the contract balance to the specified address. The opcode `SELFDESTRUCT` is one of the few operations that costs negative gas as it frees up space on the blockchain. This construct is important because contracts may need to be killed if they are no longer required or if some bug is discovered. However, if this construct is put without proper protection mechanisms in place then anyone can kill the contract.

Function and Variable Visibility

Solidity has four visibility specifiers for functions and variables. However, being declared `public` is the most tricky from a security standpoint. If an important function like a payable function or a constructor with a wrong name is declared as public, then it can cause great monetary losses. Variable visibility does not have such drastic consequences as public variables get a public getter function.

Denial of Service

A denial of service attack from a smart contract's perspective happens when a smart contract becomes inaccessible to its users. Common reasons include failure of external calls or gas costly programming patterns.

Call to the Unknown

Ethereum Smart Contracts can make calls to other smart contracts. If the addresses of these smart contracts may be user provided then a malicious actor can utilize improper authentication to call a malicious contract. If the address is hard-coded, then it does not give the flexibility to update the contract to be called over time.

Another issue is a special method called `delegatecall`. This makes the dynamically loaded code run in the caller's context. Therefore, if a `delegatecall` is made to a malicious contract, they can change storage values and potentially drain all funds from the contract.

Exception Handling

Like in any object oriented programming language, exceptions may arise due to many reasons. These must be properly handled at the programmer level. Also, lower level calls do not throw an exception. They simple return a false value which needs to be checked and the exception should be handled manually.

Vulnerabilities introduced by Ethereum updates

Early in 2019, it was discovered by Chain Security that Ethereum's Constantinople update had an effect that might have made some smart contracts vulnerable to a re-entrancy bug. Such updates cannot be predicted by smart contract developers and the Ethereum Foundation ultimately delayed rolling out the update [31].

3.2 New Taxonomy for Ethereum Smart Contract Vulnerabilities

3.2.1 Existing Taxonomies and the need for a new Taxonomy

The first taxonomy for smart contract vulnerabilities was given by Atzei et. al. [1]. They divided the vulnerabilities into three broad categories based on their source. This included Solidity, EVM and blockchain. The vulnerabilities discussed did not give a holistic picture as it did not even contain common vulnerabilities like access control, function visibility, and transaction order dependence. These vulnerabilities have been proven to be quite disastrous as shown in the infamous Parity bug. Also, it was felt that only one level of hierarchy was less for proper analysis.

Dika [13] in his Master's Thesis addressed many of the shortcomings of the Atzei taxonomy. More vulnerability categories were added and an associated severity level was also given for each vulnerability. However, the single level hierarchy was carried forward from the previous work. Also, we noticed that some vulnerability classes do not pose an immediate security risk. For example, use of `tx.origin` was labelled as a vulnerability. However, just using `tx.origin` does not cause a security breach. The problem occurs when it is used for authorization. Similarly, `blockhash` may cause a security vulnerability if used as a source of randomness. However just using it in the code does not make a contract vulnerable.

Because of these issues in the existing work, we felt that there was a need for an improved taxon-

Solidity	Call to the unknown Gas-less send Exception disorders Type casts Re-entrancy Keeping secrets
EVM	Immutable bugs Ether lost in transfer Stack size limit
Blockchain	Unpredictable state Generating Randomness Time Constraints

Table 3.1: Taxonomy of vulnerabilities as given by Atezi et. al. [1]

omy that was more hierarchical - for better analysis and understanding. Also, issues of improper vulnerability naming and incomplete vulnerability listing also needed refinement.

3.2.2 A New Taxonomy of Ethereum Smart Contract Vulnerabilities

Based on our extensive research and study of Ethereum smart contract vulnerabilities, we have come up with a new and unified vulnerability taxonomy as shown in Table 3.3.
With this new taxonomy, we try to overcome the problems in the existing literature. We try to cover almost all the security vulnerabilities that have been reported. Since, these are usually reported under different names, a security analyst would find that he/she is able to put any *existing* vulnerability he/she encounters under one of the many categories we have created. Also, unlike previous works, we have tried to eliminate any redundancies and/or incorrect categorizations.
The taxonomy is hierarchical and therefore analysis using this taxonomy would give the security researcher better insights into the root security issues in smart contracts.
The severity level is colour coded with red being high, orange being medium and green being low. The severity level has been decided taking into consideration our research, the existing literature [13] and the OWASP Risk Rating Methodology [32].

OWASP Risk Rating Methodology

IMPACT	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Note	Low	Medium
		Low	Medium	High
		LIKELIHOOD		

Table 3.2: OWASP - Determining Severity Levels

We use the OWASP Risk Rating Methodology helps to determine the severity level of any vulnerability. The risk is defined as follows -

```
Risk = Impact x Likelihood
```

Using this equation, the severity level is determined by using the Table 3.2. In this work we only utilize the severity levels from Low to High.

<table>
<tr><td rowspan="27">Solidity</td><td colspan="2">Re-entrancy</td></tr>
<tr><td rowspan="8">Access Control</td><td>Protection Issues</td></tr>
<tr><td>Authorization through <code>tx.origin</code></td></tr>
<tr><td>Unprotected Ether Withdrawal</td></tr>
<tr><td>Unprotected <code>selfdestruct</code></td></tr>
<tr><td>Unexpected Ether</td></tr>
<tr><td>Visibility Issues</td></tr>
<tr><td>Function Visibility</td></tr>
<tr><td>Variable Visibility</td></tr>
<tr><td rowspan="2">Arithmetic Issues</td><td>Integer Overflow & Underflow</td></tr>
<tr><td>Floating Point & Precision</td></tr>
<tr><td rowspan="12">Solidity Programming Issues</td><td>Uninitialized Storage Pointers</td></tr>
<tr><td>Variable Shadowing</td></tr>
<tr><td>Keeping Secrets</td></tr>
<tr><td>Type Casts</td></tr>
<tr><td>Lack of Proper Signature Verification</td></tr>
<tr><td>Write to Arbitrary Storage Location</td></tr>
<tr><td>Incorrect Inheritance Order</td></tr>
<tr><td>Typographical Errors</td></tr>
<tr><td>Use of Assembly</td></tr>
<tr><td>Use of Deprecated Functions/Constructions</td></tr>
<tr><td>Floating or No Pragma</td></tr>
<tr><td>Outdated Compiler Version</td></tr>
<tr><td rowspan="5">Exception Handling</td><td>Unchecked Call</td></tr>
<tr><td>Gasless Send</td></tr>
<tr><td>Call Stack Limit</td></tr>
<tr><td>Assert Violation</td></tr>
<tr><td>Requirement Violation</td></tr>
<tr><td></td><td rowspan="2">Call to the Unknown</td><td>Dangerous Delegate Call</td></tr>
<tr><td></td><td>External Contract Referencing</td></tr>
<tr><td></td><td rowspan="2">Denial of Service</td><td>DoS with block gas limit</td></tr>
<tr><td></td><td>DoS with failed call</td></tr>
<tr><td rowspan="3">EVM</td><td colspan="2">Short Address Attack</td></tr>
<tr><td colspan="2">Immutable bugs</td></tr>
<tr><td colspan="2">Stack size limit</td></tr>
<tr><td rowspan="5">Blockchain</td><td colspan="2">Bad Randomness</td></tr>
<tr><td colspan="2">Untrustworthy Data Feeds</td></tr>
<tr><td colspan="2">Transaction Order Dependence</td></tr>
<tr><td colspan="2">Timestamp Dependence</td></tr>
<tr><td colspan="2">Unpredictable state (Dynamic Libraries)</td></tr>
</table>

Table 3.3: A New Taxonomy of Ethereum Smart Contract Vulnerabilities

Chapter 4

Analysis of Security Tools for Ethereum Smart Contracts

4.1 Classification of Tools Available for Ethereum Smart Contracts

There are many different tools available for Ethereum Smart Contracts. These tools have been gathered from research publications and through I nternet searches. In this section, we have classified the various tools available into different categories, so that the end users can easily find which tool to use for their particular application.
Even though our work is primarily concerned with Security Tools, the other tools are included for the reader's convenience.

Security Tools

These are tools which take as input either the source code or the bytecode of a contract and give outputs on the security issues present. These are the tools that we are primarily concerned in with our work.

Visualization Tools

Visualization tools help give graphical outputs like control flow graphs, dependency graphs, etc. of the given contract to help in analysis. Tools like solgraph [33] and rattle [34] fall under this category.

Disassemblers and Decompilers

A dis-assembler converts the binary code back into the high level language code while a decompiler converts the binary code to a low level language for better understanding. evm-dis [35] is a popular dis-assembler for smart contracts.

Linters

Linters are static analysis tools primarily focused on detecting poor coding practices, programming errors, etc. Ethlint[36] is a common linting tool of ethereum smart contracts.

Miscellaneous Tools

This includes tools like SolMet [37] which help give common code metrics like number of lines of code, number of functions per contract, etc. for solidity source files.

4.1.1 Methods employed by the Security Tools

Static Analysis

Static Analysis essentially means evaluating the program code without actually running it. It looks at the code structure, the decompiled outputs, and control flow graphs to identify common security issues. SmartCheck [38], Slither [39] and RemixIDE [40] are static analysis security tools for Ethereum smart contracts.

Symbolic Execution

Symbolic execution is considered to be in the middle of static and dynamic analysis. It explores possible execution paths for a program without any concrete input values. Instead of values, it uses symbols and keeps track of the symbolic state. It leverages constraint solvers to make sure that all the properties are satisfied. Mythril [41] and Oyente [42] are the popular Symbolic Execution tools for smart contract security.

Formal Verification

Formal Verification incorporates mathematical models to make sure that the code is free of errors. Bhargavan et. al [43] conducted a study of smart contracts using F*. However, the work is not available as open source to the best of our knowledge.

4.2 Creation of the Vulnerability Benchmark

4.2.1 Need for a Vulnerability Benchmark

It is observed that many security tools have come up for Ethereum smart contracts over the years. However, it is also observed that these tools are usually tested on different test-instances and in some cases even the ground truth is unknown. Therefore, as a smart contract developer or a user, it becomes difficult to actually compare the performance of different tools without a proper benchmark.

Dika [13] tried to solve this issue. However, he tested only three tools on just 23 vulnerable and 21 audited-safe contracts. A contract was called vulnerable if it had *any* vulnerability. However, it was not checked that a tool properly detected the vulnerability claimed and the results were presented as is.

4.2.2 Benchmark Creation Methodology & Statistics

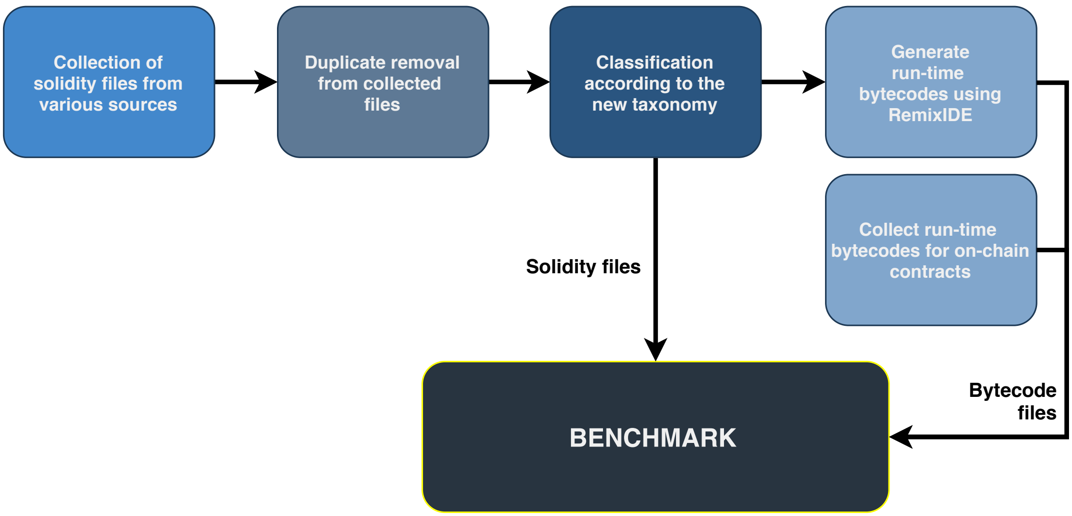

Figure 4.1: Vulnerability Benchmark Creation

To create the benchmark, we collected contracts known to be vulnerable from various sources. This included -

- Smart Contract Weakness Classification (SWC) Registry [30]
- (Not So) Smart Contracts [44]
- EVM Analyzer Benchmark Suite [45]
- Research papers, theses and books [1][13][17][46][47]
- Various blog posts, articles, etc. [15][48][49][50][51][52][53][54][55][56]

After collecting all the instances, we manually removed the duplicate contracts - this was important as we found that there was notable overlap between contracts gathered from different sources. After this, we manually checked the contracts and classified them as per the new taxonomy. Finally we compiled the smart contracts into run-time bytecode. However, as each contract required a different version of solidity, and `solc-select` [57] did not support such a large range of compiler versions, we leveraged Remix IDE [40] to manually generate the run-time byte-codes for each contract and stored it separately. This was not done for the on-chain contracts and the run-time byte-codes for these were directly taken from the blockchain. A summary of the benchmark creation methodology is depicted in Figure 4.1.

Contract Name	Vulnerability	Address
SmartBillions	Bad Randomness	0x5acE17f87c7391E5792a7683069A8025B83bbd85
Lottery	Bad Randomness	0x80ddae5251047d6CeB29765f38FED1C0013004b7
EtherLotto	Bad Randomness	0xA11E4ed59dC94e69612f3111942626Ed513cB172
Ethraffle_v4b	Bad Randomness	0xcC88937F325d1C6B97da0AFDbb4cA542EFA70870
BlackJack	Bad Randomness	0xA65D59708838581520511d98fB8b5d1F76A96cad
LuckyDoubler	Bad Randomness	0xF767fCA8e65d03fE16D4e38810f5E5376c3372A8
EthStick	Bad Randomness, Floating Point Precision	0xbA6284cA128d72B25f1353FadD06Aa145D9095Af
TheRun	Bad Randomness, Transaction Order Dependence	0xcac337492149bDB66b088bf5914beDfBf78cCC18
Parity MultiSig Wallet	Dangerous Delegate Call, Denial of Service, Function Default Visibility	0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4
GovernMental Ponzi Scheme	Denial of Service, TimeStamp Dependence	0xF45717552f12Ef7cb65e95476F217Ea008167Ae3
Private_Bank	External Contract Referencing	0x95D34980095380851902ccd9A1Fb4C813C2cb639
StackyGame	Function Default Visibility	0x8f13a1d43408b6434dd10e161361386f3952d665
GoodFellas	Function Default Visibility	0x5E84C1A6E8b7cD42041004De5cD911d537C5C007
Rubixi	Function Default Visibility, Immutable Bugs	0xe82719202e5965Cf5D9B6673B7503a3b92DE20be
BeautyChain (BEC)	Integer Overflow	0xC5d105E63711398aF9bbff092d4B6769C82F793D
MESH	Integer Overflow	0x3AC6cb00f5a44712022a51fbace4C7497F56eE31
UGToken	Integer Overflow	0x43eE79e379e7b78D871100ed696e803E7893b644
SMT	Integer Overflow	0x55F93985431Fc9304077687a35A1BA103dC1e081
SMART	Integer Overflow	0x60be37dacb94748a12208a7ff298f6112365e31f
MTC	Integer Overflow	0x8febf7551eea6ce499f96537ae0e2075c5a7301a
GGToken	Integer Overflow	0xf20b76ed9d5467fdcdc1444455e303257d2827c7
CNYToken	Integer Overflow	0x041b3eb05560ba2670def3cc5eec2aeef8e5d14b
CNYTokenPlus	Integer Overflow	0xfbb7b2295ab9f987a9f7bd5ba6c9de8ee762deb8
DAO	Reentrancy	0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413
SpankChain	Reentrancy	0xf91546835f756DA0c10cFa0CDA95b15577b84aA7
CityMayor	Reentrancy	0x4bdDe1E9fbaeF2579dD63E2AbbF0BE445ab93F10
ICO	Transaction Order Dependance	0xd80cc3550Da18313aF09fbd35571084913Cd5246
LastIsMe	Transaction Order Dependance	0x5D9B8FA00C16BCafaE47Deed872E919C8F6535BF
FirePonzi	Typographical error	0x062524205cA7eCf27F4A851eDeC93C7aD72f427b
KingofTheEtherThrone	Unchecked External Call	0xb336a86e2feb1e87a328fcb7dd4d04de3df254d0
EtherPot	Unchecked External Call	0x539f2912831125c9B86451420Bc0D37b219587f9
OpenAddressLottery	Unitialized Storage Pointers	0x741F1923974464eFd0Aa70e77800BA5d9ed18902
CryptoRoulette	Unitialized Storage Pointers	0x8685631276cFCf17a973d92f6DC11645E5158c0c
G_GAME	Unitialized Storage Pointers	0x3CAF97B4D97276d75185aaF1DCf3A2A8755AFe27

Table 4.1: On-chain vulnerable contracts in our benchmark

It has been observed that Ethereum smart contracts have been used for creating ponzi-schemes to scam innocent people into loosing money by promising extraordinarily high returns [47]. Even

though a ponzi contract might not be a direct security vulnerability, we have included them in our study because of the high monetary impact of such contracts. Apart from ponzi schemes like GovernMental, FirePonzi and Rubixi which have already been included, we added ponzi schemes that exhibited one or more of the following properties - contracts that do not refund and contracts that allow the owner to withdraw funds. These properties are typical in ponzi schemes, however they cannot be classified as security vulnerabilities directly without knowing the context.
Table 4.2 lists the additional ponzi schemes which have been included in the benchmark because they exhibit the above properties and had not been included before in the benchmark.

Contract Name	Vulnerability	Address
DynamicPyramid	Does not Refund	0xa9e4E3b1DA2462752AeA980698c335E70E9AB26C
GreedPit	Does not Refund; Allow Owner to withdraw funds	0x446D1696a5527018453cdA3d67aa4C2cd189b9f6
NanoPyramid	Does not Refund	0xe19e5f100d6a31169b5dcA265C9285059C41D4F6
Tomeka	Does not Refund	0x24Ec083b6A022099003e3D035fed48b9a58296E5
ProtectTheCastle	Allow Owner to withdraw funds	0x7D56485e026D5D3881F778E99969D2b1F90c50aF
EthVentures	Allow Owner to withdraw funds	0x99D982E49bCB5465A6B4c1e0eC4341c912D9Ba42

Table 4.2: Additional Ponzi Schemes included in our benchmark

The final benchmark consists of 180 contracts spread over all the categories. Out of these we have 162 unique contracts. This includes 40 on-chain contracts (including six additional ponzi schemes). This is very high in comparison to the 23 vulnerable smart contracts identified by [13]. Figure 4.2 shows the distribution of the benchmark instances across different categories.

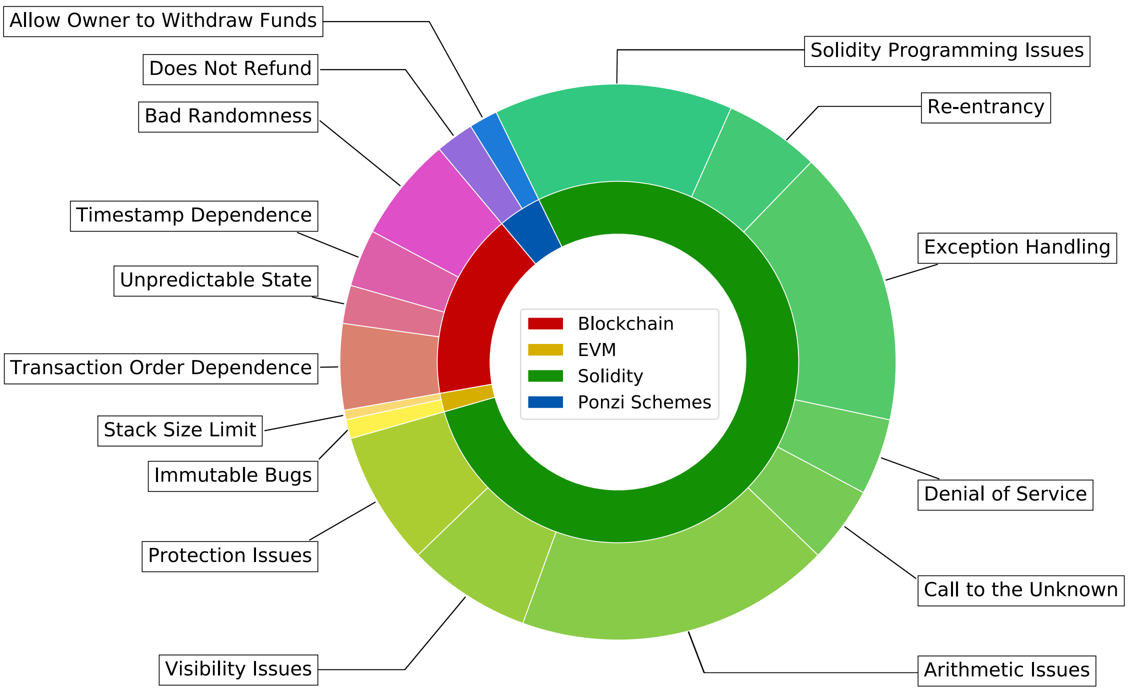

Figure 4.2: Distribution of instances in the Vulnerability Benchmark

4.3 Experimental Setup

For the purpose of the study, we select the security tools that are actively maintained, open-sourced, ready for use and cover a fairly large section of the vulnerabilities. Keeping the above constraints in mind, the following tools were selected -

Remix IDE

Remix IDE [40] is primarily an integrated development environment (IDE) for developing Solidity smart contracts. It can connect to the Ethereum network using Metamask and developers can directly deploy smart contracts from Remix. It is developed and maintained by the Ethereum Foundation.

The IDE has a security module to help developers with common security issues like re-entrancy, etc. It requires the solidity file of the contract to work. As a web interface was available, the testing using the benchmark instances was carried out manually.

SmartCheck

SmartCheck [38] is a static analysis tool for Solidity and Vyper smart contracts. It is developed by SmartDec and the University of Luxembourg. Like other static analysis tools, it does not work on byte-codes and requires the source codes to be present for analysis.

It works by transforming the source codes into an intermediate representation which is XML-based. This representation is then checked against XPath patterns to highlight potential vulnerabilities in the code. The tool is open sourced and also has a web interface hosted at [58].

Slither

Slither [39] is a static analysis tool for solidity source files written in Python 3. It is open sourced and is developed by by Trail of Bits. It works on contracts written in solidity >= 0.4 and requires the solidity files for analysis.

It leverages an intermediate representation call SlithIR for code analysis. However, it requires the correct solidity version to be installed in the system. For this, we utilize another tool by Trail of Bits called `solc-select`[57] to switch to the right compiler version which is predetermined manually.

Oyente

Oyente [42] is one of the earliest security tools for Solidity smart contracts. It was developed by security researchers at the National University of Singapore and is now being maintained by Melonport. Oyente leverages symbolic execution to find potential vulnerabilites in the smart

	Remix IDE	Smart-Check	Slither	Oyente	Securify	Mythril
Version/ Date Used	4-Mar-2019	2.0.1	0.4.0	0.2.7	17-Apr-19	0.20.4
Technique	Static Analysis	Static Analysis	Static Analysis	Symbolic Execution	Symbolic Execution	Symbolic Execution
WUI/ CLI	WUI	WUI + CLI	CLI	WUI + CLI	WUI + CLI	WUI + CLI
Works on src-file/ bytecode	src-file	src-file	src-file	src-file + bytecode	src-file + bytecode	src-file + bytecode
Developed by	Ethereum Foundation	SmartDec	Trail of Bits	NUS + Melonport	ETH Zurich	ConsenSys

Table 4.3: Summary of tools used in the study

contracts. It works with both byte-codes and solidity files.

Being one of the first tools in this area, Oyente has been extended by many researchers over the years. For example, the control flow graphs generated by Oyente are also used by EthIR [59], which is a high level analysis tool for Solidity. A web interface for the tool is also available [60].

Securify

Securify [61] has been created by researchers at ETH Zurich in collaboration with ChainSecurity for security testing of Ethereum smart contracts. It works on both solidity source files and byte-codes. It has also received funding from the Ethereum Foundation to help mitigate the security issues in smart contracts. It analyzes the contract symbolically to get semantic information and then checks against patterns to see if a particular property holds or not. A web interface is also available at [62].

Mythril

Mythril [41] is a security tool developed by ConsenSys. It uses as a combination of symbolic execution and taint analysis to identify common security issues. Recently, a new initiative called MythX was launched with a similar core as Mythril for smart contract developers to provide security as a service. However, it is still in beta testing and is not available as open source. Therefore, we use Mythril Classic for our testing purposes.

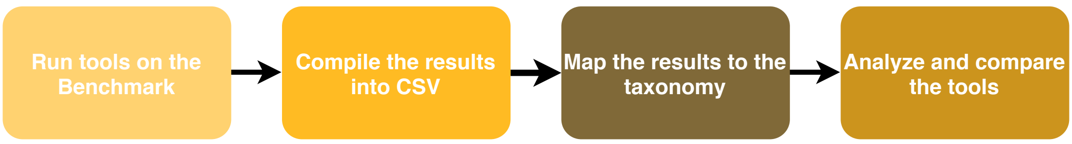

Figure 4.3: Experiment Summary

Reported by the Tool	Mapping to the new Taxonomy
Remix IDE	
Transaction origin	Authorization through `tx.origin`
Check-effects	Re-entrancy
Block timestamp usage	Timestamp Dependence
`block.blockhash` usage	Bad randomness
inline assembly	Use of Assembly
Use of `selfdestruct`	Unprotected `selfdestruct`
Low level calls/use of send	Unchecked Call
SmartCheck	
Deprecated Constructions	Use of Deprecated Functions/Constructions
Gas limit in loops	DoS with block gas limit
Upgrade to 0.5.0	Outdated Compiler Version
Pragmas version	Floating or No Pragma
Send, Unchecked call, Call without data	Unchecked Call
Using inline assembly	Use of Assembly
Incorrect Blockhash	Bad Randomness
Transfer in loop	DoS with failed call
Exact time	Timestamp dependence
Div mul	Floating Point and Precision
Visibility	Function Default Visibility
Locked money	Ponzi Scheme – Do Not Refund
Redundant fallback reject, Balance equality	Unexpected Ether
Array length manipulation	Write to Arbitrary Storage Location
Slither	
Reentrancy-eth,reentrancy-no-eth,reentrancy-benign	Re-entrancy
tx-origin	Authorization through tx.origin
timestamp	Timestamp dependence
Uninitialized-state, uninitialized-local, uninitialized-storage	Uninitialized storage pointers
suicidal	Unprotected selfdestruct
assembly	Use of Assembly
deprecated-standards	Use of Deprecated Functions or Constructions
solc-version	Outdated Compiler Version
calls-loop	Denial of Service with failed call
arbitrary-send	Unprotected Ether Withdrawal
incorrect-equality	Unexpected Ether
Unused-return, low-level-calls	Unchecked External Call
Shadowing-builtin, shadowing-local, shadowing-state	Shadowing State Variables
controlled-delegatecall	Dangerous Delegate Call
locked-ether	Ponzi scheme – Does not Return
OYENTE	
Call stack	Stack size limit
Re-entrancy	Re-entrancy
Time Dependency	Timestamp Dependence
Integer Overflow, Integer Underflow	Integer Overflow & Underflow
Money Concurrency	Transaction Order Dependence
Mythril Classic	
Integer Underflow, Integer Overflow	Integer Overflow & Underflow
Unchecked Call Return Value	Unchecked Call
Unprotected Selfdestruct	Unprotected selfdestruct
Unprotected Ether Withdrawal	Unprotected Ether Withdrawal
Use of tx.origin	Authorization through tx.origin
Exception State	Exception Handling
External Call To Fixed/User-Supplied Address	Dangerous Delegate Call
Use of callcode	Use of Deprecated Functions/Constructs
Dependence on predictable variable/environment variable	Bad Randomness
Multiple Calls in a Single Transaction	Denial of Service
Securify	
DAO, DAOConstantGas	Re-entrancy
LockedEther	Ponzi Scheme – Do not Refund
MissingInputValidation	Type Casts
RepeatedCall	Dangerous Delegate Call
TODAmount, TODReceiver	Transaction Ordering Dependence
UnhandledException	Unchecked Call
UnrestrictedEtherFlow	Unprotected Ether Withdrawal
UnrestrictedWrite	Write to arbitrary storage location

Table 4.4: Vulnerability mapping to the new taxonomy

All the experiments were carried out on a machine running Ubuntu 18.04.2 LTS on an Intel® Core™ i7-4770 CPU with 16GB DDR3 RAM. Also, the tools that worked on both solidity and bytecode files were tested on bytecode files only. The results output by each tool were then converted to the new taxonomy as shown in Table 4.4 to allow us to compare the tools uniformly.

4.4 Results and Analysis

4.4.1 Results on the Vulnerability Benchmark

Remix IDE

The performance of Remix IDE is surprisingly good. As seen in figure 4.4, it detects vulnerabilities like `tx.origin` authorization, use of assembly, unchecked call and timestamp dependence with 100% accuracy. However, we find that these vulnerabilities are caught by mere presence of certain constructs without checking whether they actually result in a vulnerability or not. For example, Timestamp Dependence flag is raised if `timestamp` is used anywhere in the code. Similarly, `tx.origin` flag is raised if `tx.origin` is used anywhere within the code without checking if any it causes any security issue or not. The `selfdestruct` module works similarly. However it could not detect the Parity Bug because it uses the older `suicide` construct. It was also observed that for solidity versions 0.3.1 and prior, the check-effects and the selfdestruct modules gave an error. This resulted in the famous DAO contract not being analysed by the tool.

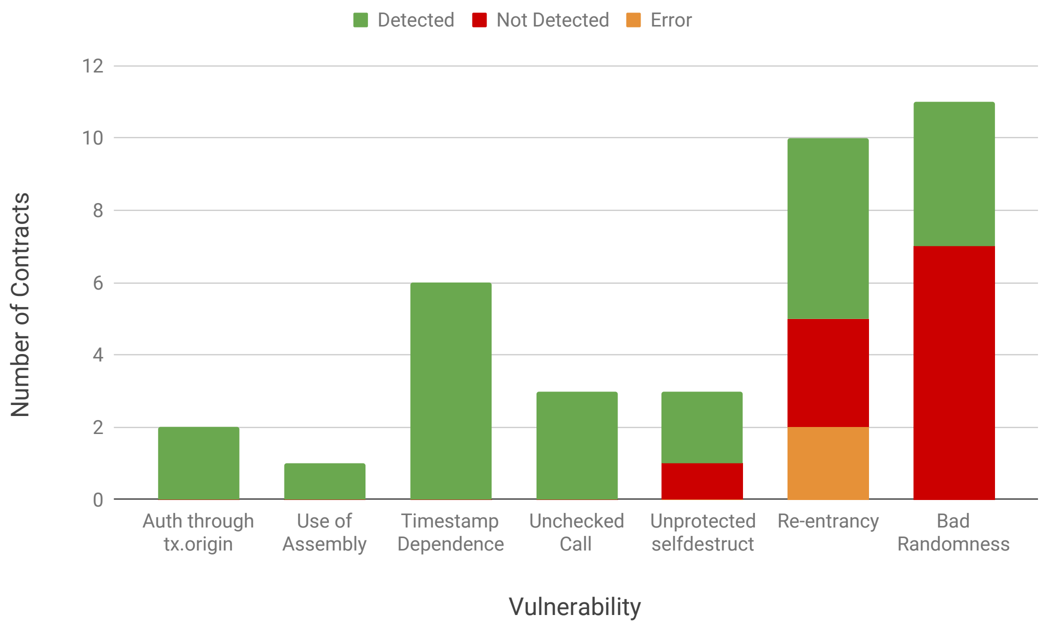

Figure 4.4: Results of Remix IDE on the Vulnerability Benchmark

SmartCheck

The performance of SmartCheck is given in figure 4.5. It has a good performance in only a few of the many categories that it can detect. The include security issues like use of deprecated functions, unchecked call, use of assembly, etc. However, the performance on other instances is not very good.

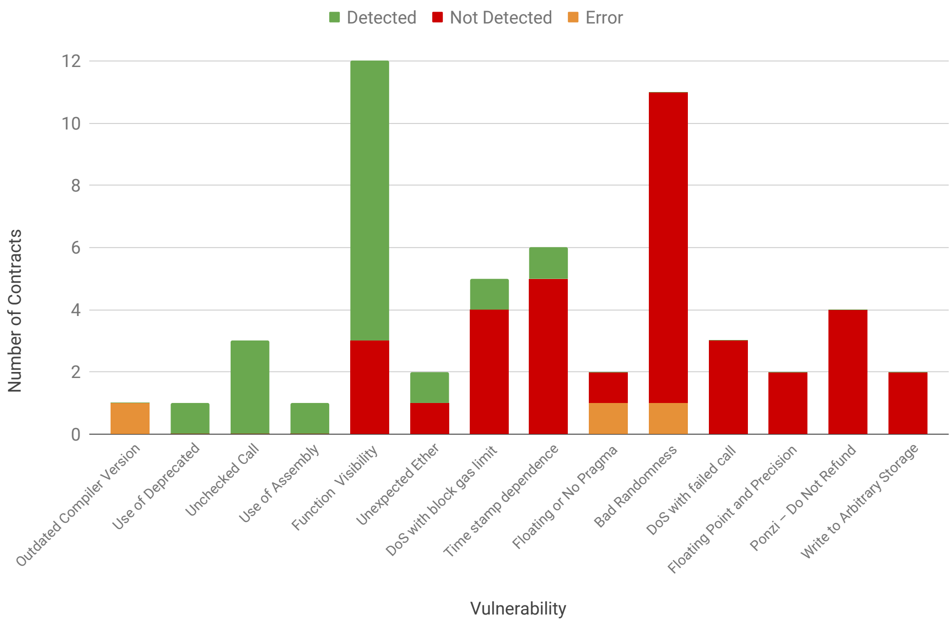

Figure 4.5: Results of SmartCheck on the Vulnerability Benchmark

Slither

Slither has a very good performance across most of the categories that it detects. There was no category that it could not detect even one instance from. The biggest drawback of slither is that does not work with older solidity versions (prior to 0.4) and requires the correct version of solidity to be present on the system. Because of this, a lot of contracts in the benchmark gave errors with slither. However, it is a very good tool for smart contract developers who are developing in newer versions of solidity.

Oyente

Being one of the earliest tools, Oyente is now showing it's age. It covers a very low number of vulnerabilities. The results are shown in figure 4.7. Average EVM code coverage for the entire benchmark set was found to be 75.98%. Also, there was not a single report of integer overflow or underflow across the complete benchmark. We believe this is some bug in the tool causing this

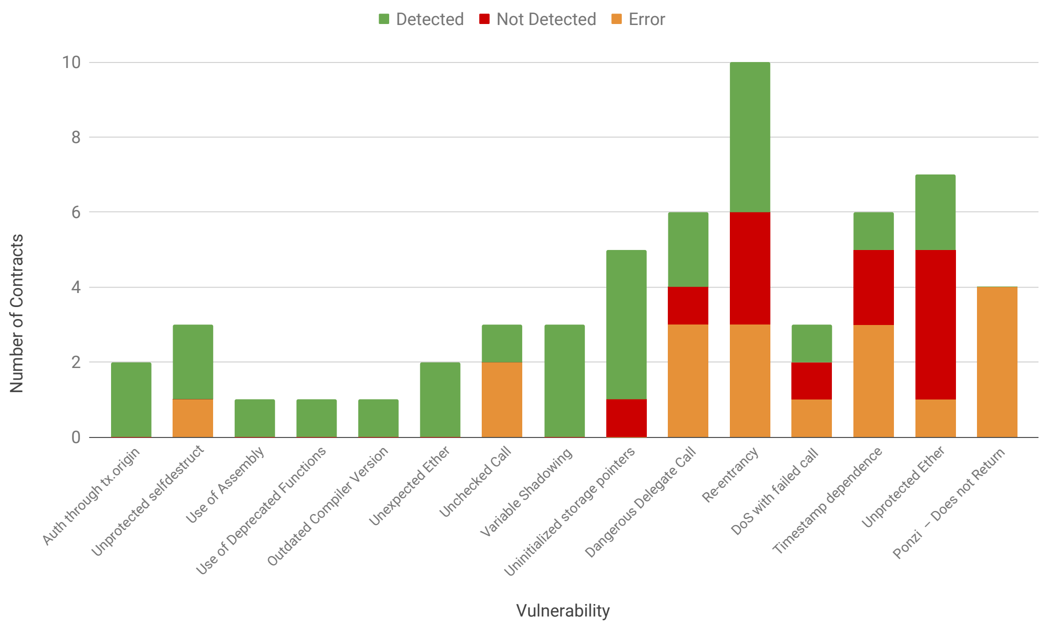

Figure 4.6: Results of Slither on the Vulnerability Benchmark

behaviour.

Figure 4.7: Results of Oyente on the Vulnerability Benchmark

Securify

Securify is the only tool that reports a contract as 'safe' from a particular vulnerability. If the contract contains a vulnerability, and Securify reports it as 'safe', we call it false negative. From figure 4.8 we can see that Securify reports a lot of false negatives. However, it has a decent performance on re-entrancy bug detection.

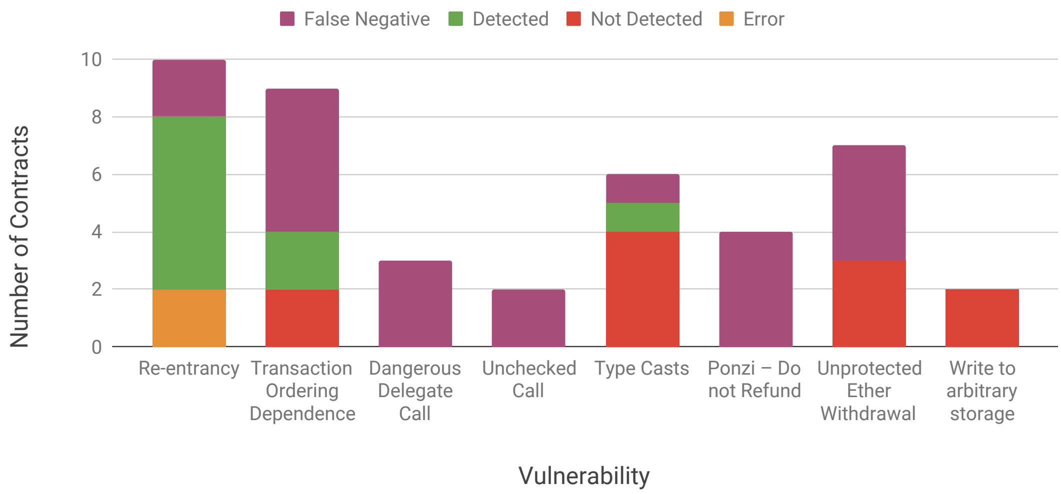

Figure 4.8: Results of Securify on the Vulnerability Benchmark

Mythril

The performance of Mythril on the benchmark is shown in figure 4.9. It is able to detect the attacks with a fair accuracy, however it encounters a lot of errors. This makes its performance inferior to some static analysis tools like Slither.

4.4.2 Analysis

Figure 4.10 depicts the time taken by each tool on the benchmark. Static Analysis tools are faster than symbolic execution tools as expected. Mythril is the slowest, while Slither is the fastest tool. Remix could not be included in this comparison as it was used manually.

Slither and Mythril gives the most errors as showin in figure 4.11. Remix and Oyente are the only tools that do not give any errors on any contract in the benchmark.

Table 4.5 shows the tool vulnerability matrix as claimed by the tools. Here, `Y` implies that the tool claims to detect a particular vulnerability. Also PS stands for Ponzi Schemes. It is clear from this table that most of the vulnerabilities have been covered by one tool or the other. Slither and Oyente cover the most vulnerabilities closely followed by Mythril. Also, the most popular vulnerabilities with the tools are re-entrancy, unchecked call and timestamp dependence.

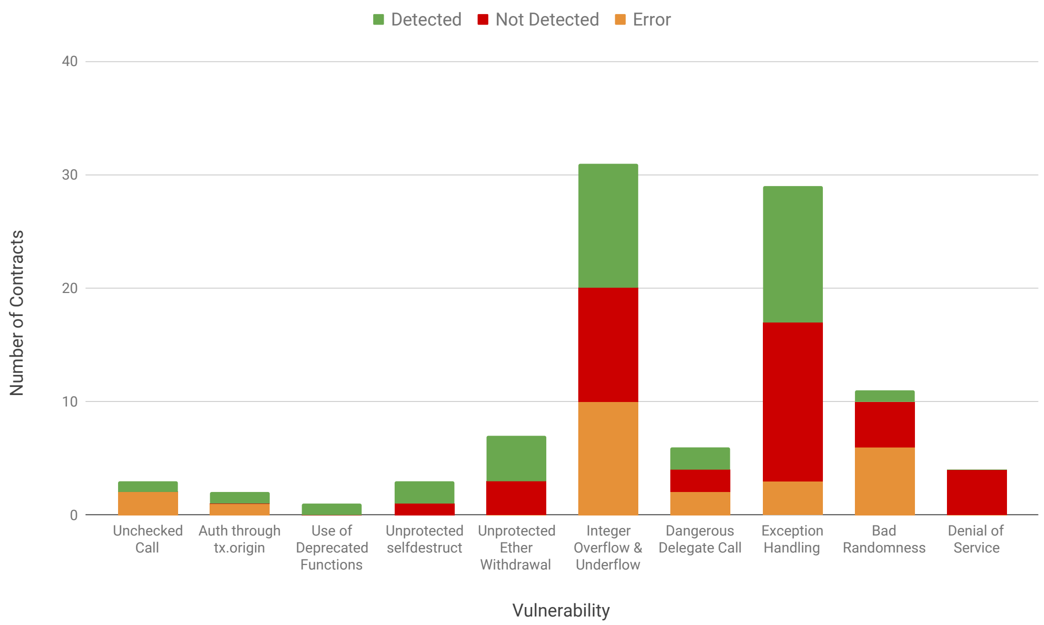

Figure 4.9: Results of Mythril on the Vulnerability Benchmark

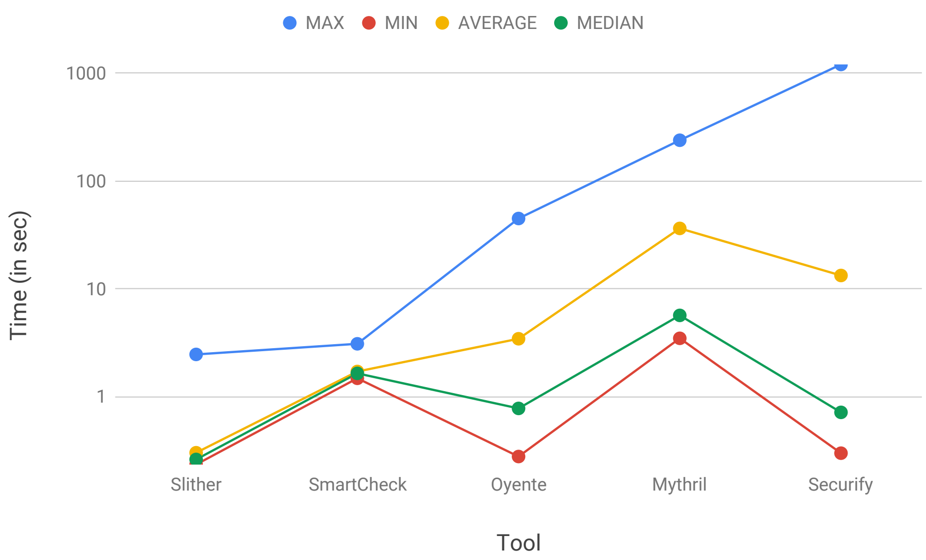

Figure 4.10: Time taken by the tools on the Vulnerability Benchmark

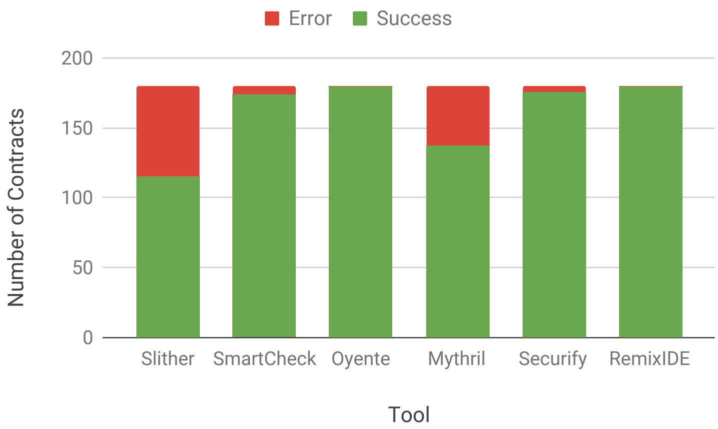

Figure 4.11: Percentage of contracts in the Vulnerability Benchmark successfully analyzed

		Remix	**Slither**	**SmartCheck**	**Oyente**	**Mythril**	**Securify**	**SUM**
SOLIDITY	Re-entrancy	Y	Y		Y		Y	4
	Authorization through `tx.origin`	Y	Y			Y		3
	Unprotected Ether Withdrawal		Y			Y	Y	3
	Unprotected `selfdestruct`	Y	Y			Y		3
	Unexpected Ether		Y	Y				2
	Function Visibility			Y				1
	Variable Visibility							0
	Integer Overflow & Underflow				Y	Y		2
	Floating Point & Precision			Y				1
	Uninitialized Storage Pointers		Y					1
	Variable Shadowing		Y					1
	Keeping Secrets			Y				1
	Type Casts						Y	1
	Lack of Proper Signature Verification							0
	Write to Arbitrary Storage Location			Y			Y	2
	Incorrect Inheritance Order							0
	Typographical Errors							0
	Use of Assembly	Y	Y	Y				3
	Use of Deprecated Functions		Y	Y		Y		3
	Floating or No Pragma			Y				1
	Outdated Compiler Version		Y	Y				2
	Unchecked Call	Y	Y	Y			Y	4
	Gasless Send						Y	1
	Call Stack Limit						Y	1
	Assert Violation						Y	1
	Requirement Violation						Y	1
	Dangerous Delegate Call		Y			Y	Y	3
	External Contract Referencing							0
	DoS with block gas limit			Y			Y	2
	DoS with failed call		Y	Y			Y	3
EVM	Short Address Attack							0
	Immutable bugs							0
	Stack size limit				Y			1
B/CHAIN	Bad Randomness	Y		Y		Y		3
	Untrustworthy Data Feeds							0
	Transaction Order Dependence				Y		Y	2
	Timestamp Dependence	Y	Y	Y	Y			4
	Unpredictable state							0
PS	Does not Return		Y	Y			Y	3
	Allows Owner to Withdraw Funds							0
	TOTAL	7	15	15	5	7	14	

Table 4.5: Tool-vulnerability matrix as claimed by the tools

4.5 Summary

In this chapter, we have looked at the various tools available for smart contracts. For proper comparison of the tools, we have created a new vulnerability benchmark by collecting vulnerable

		RemixIDE	Slither	SmartCheck	Oyente	Mythril	Securify
SOLIDITY	Re-entrancy	5	4		5		6
	Authorization through `tx.origin`	2	2			1	
	Unprotected Ether Withdrawal		2			4	3
	Unprotected `selfdestruct`	2	2			2	
	Unexpected Ether		2	1			
	Function Visibility			9			
	Variable Visibility						
	Integer Overflow & Underflow				-	11	
	Floating Point & Precision			0			
	Uninitialized Storage Pointers		4				
	Variable Shadowing		3				
	Keeping Secrets						
	Type Casts						1
	Lack of Proper Signature Verification						
	Write to Arbitrary Storage Location			0			0
	Incorrect Inheritance Order						
	Typographical Errors						
	Use of Assembly	1	1	1			
	Use of Deprecated Functions/Constructions		1	1		1	
	Floating or No Pragma			0			
	Outdated Compiler Version		1	0			
	Unchecked Call	2	1	3		1	0
	Gasless Send					0	
	Call Stack Limit					0	
	Assert Violation					12	
	Requirement Violation					0	
	Dangerous Delegate Call		2			2	0
	External Contract Referencing						
	DoS with block gas limit			1			0
	DoS with failed call		1	0			
EVM	Immutable bugs						
	Stack size limit				1		
B/CHAIN	Bad Randomness	4		0			
	Transaction Order Dependence				5		2
	Timestamp Dependence	6	1	1	2		
	Unpredictable state (Dynamic Libraries)						
PS	Does not Return		0	0			0
	Allows Owner to Withdraw Funds						

Table 4.6: Tool effectiveness for different vulnerabilities

smart contracts from different sources. These are then run against the different tools. To ensure uniformity, and to enable comparison of different tools we create a mapping between the outputs given by the tools and our taxonomy as given in Chapter 3. The effectiveness of the tools in detecting the vulnerabilities in the benchmark is shown in Table 4.6. The cells highlighted in green indicate that all the instances of that vulnerability present in the benchmark are successfully detected, while grey highlights the maximum vulnerabilities (though not all) accurately detected across all the tools. We observe that many vulnerabilities are not being detected by the tools. We also observe that even though the tools cover a wide spectrum of vulnerabilities, they are not very accurate in detecting them. The best tool from our study is Slither. It covers a wide range of vulnerabilities and is the only tool that detected at-least one from each category it could successfully evaluate. The only drawback is that it works on solidity versions greater than 0.4.0. Nevertheless, it is still a good tool for new smart contract developers.

Chapter 5

On-Chain Smart Contracts: Data Collection and Analysis

5.1 Data Collection

This section details the various aspects involved in collection of smart contract data from the Ethereum main-net including the problems faced, solutions employed and data statistics.

5.1.1 The challenges in Smart Contract Data Collection

Even though Ethereum is a public blockchain which means that all the data is available publicly to anyone who connects to the network, there were a few challenges that we faced -

- The Ethereum data has grown in size over the years, with its size crossing 1TB in May, 2018 [63]. To have access to all the data, we have to run a 'full' node which is both time and space consuming with full node sync times increasing drastically over the last few years.
- The next issue was finding which were the smart contract addresses in the Ethereum network. Simply brute-forcing address (16^{40} possibilities) was infeasible.

5.1.2 Smart Contract Bytecode Collection

To deal with the problem of running a full node, we leveraged INFURA API [64] (by Consensys) and leveraged it's `geth-like` methods and `Web3` to get the byte-codes of the on-chain contracts. To tackle the problem of finding the smart contract addresses, we went through all the transactions from the genesis block till block number 7.1M (mined on 20 January, 2019), found all the addresses and used the `getCode()` method provided by `geth` to find if it was a smart contract address or not.

```
//Store all the addresses from all the transactions
for blockNo 1 to 7.1M do:
    for all txn in blockNo do:
        store from_addr and to_addr in addr_file.txt
//Keep only the unique addresses
sort -o addr_file_u.txt -u addr_file.txt
For each address, store the bytecode if available
for all addr in addr_file_u.txt do:
    code = geth.getCode(addr)
    if(code != 0x0):
        store code in addr.bin.hex
```

The trade-off with this approach is that we do not get the contracts without any normal transactions recorded on the blockchain or which have been killed. Therefore, only live and interacted-with at-least once contracts are collected by our methods.

However, as the search space was huge, the network quickly became a bottleneck. Therefore, we spread out this data collection activity to Google Compute Engine instances.

In total, our scripts traversed 380M transactions over 7.1M blocks. The total number of unique addresses we found in those transactions was 44M, and out of those only 1.9M addresses were found to contain smart contracts. The byte-codes of these 1.9M smart contracts was stored.

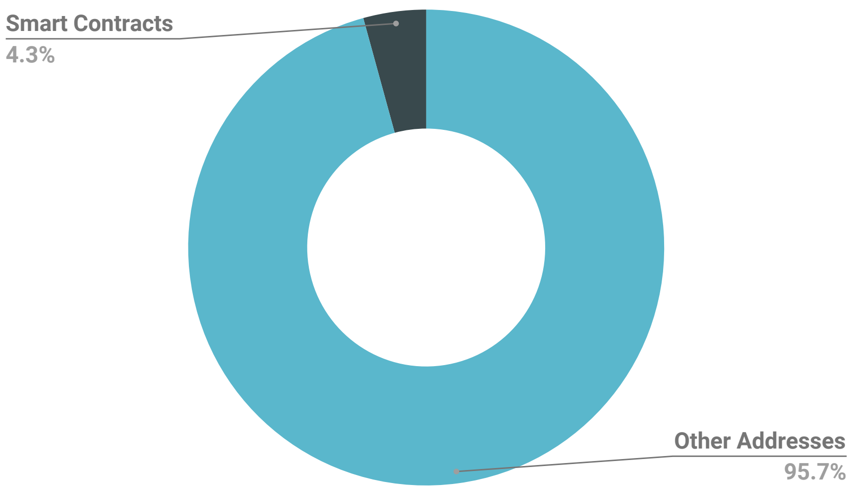

Figure 5.1: Percentage of Smart Contract Addresses

5.1.3 Source Code Collection

Etherscan has a utility called verify source code, using which smart contract developers can publish the source-code of the smart contract. The utility takes the source code, compiler version, optimization parameters, libraries, etc. and compiles the given source-code with the provided parameters. If the resulting bytecode matches that present on the chain, the source code is verified successfully and is published on Etherscan's website.

We utilized Etherscan's API to get the source codes of all the 1.9M smart contracts. Our scripts found 887K (46.4%) smart contracts with source codes available.

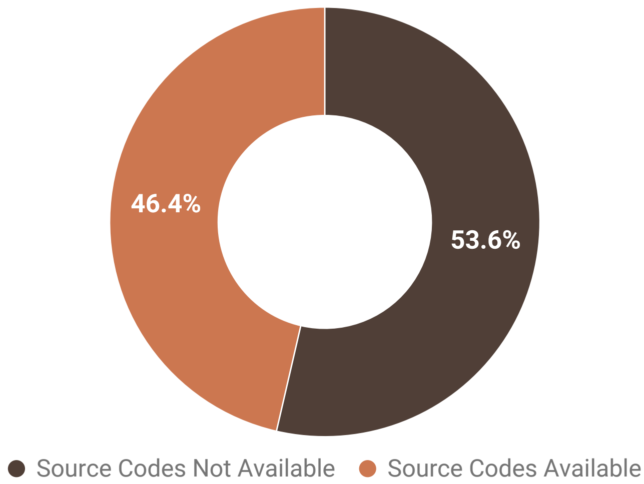

Figure 5.2: Percentage of smart contracts with verified source codes available

5.2 Analysis of the collected on-chain contracts

5.2.1 Duplicity

Code reuse is a common practice in many applications and Smart Contracts are no exception. As a matter of fact, many real world access control vulnerabilities (like Rubixi) have arisen because of improper copy-pasting i.e. not changing the constructor name when the contract name was changed, leading to anyone becoming the `owner` of the contract.

Therefore, we suspect that our on-chain data-set also has duplicates. For the purpose of our study, we define two contracts to be duplicates of each other if they have the exactly same deployed bytecode on the blockchain. To find duplicates, we use an approach similar to [65] - Take the bytecode of every contract, calculate the `MD5` hash, and only keep the contracts with a unique `MD5` hash.

In the bytecode data set, it is observed that only 103K (or 5.4%) of the 1.9M bytecodes are unique. Similar results are observed for the solidity data-set with only 42K (or 4.7%) of the 887K contracts

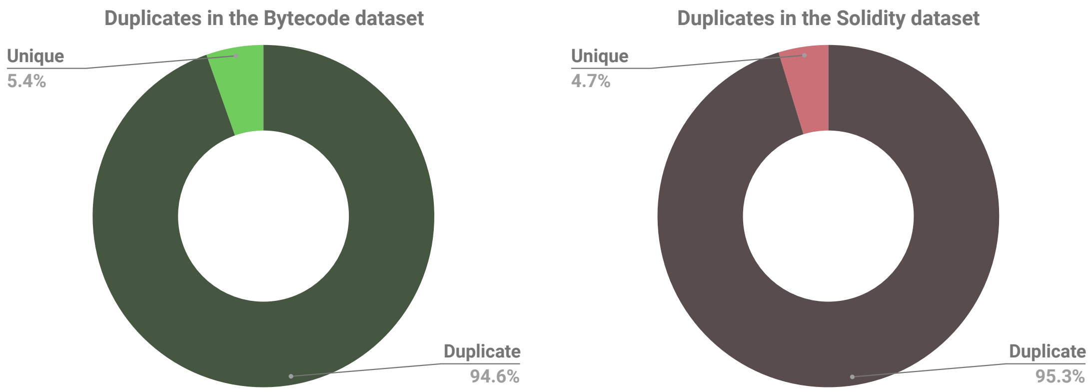

Figure 5.3: Duplicates in our dataset

having unique bytecode. This indicates a very high re-usability in smart contract development and further highlights the importance of having good security practices as if one smart contract is vulnerable, all its duplicates are vulnerable too. On the positive side, the number of contracts to be secured is drastically lower that previously imagined. Therefore, using tools that report vulnerabilities by formal methods (like Zeus [66]) can be viable option.

Table 5.1 shows the top ten most duplicated contracts in the data-set. The most duplicated contract is a User Wallet contract that has been deployed over 651K times. However, only two out of the top ten duplicated contracts have verified source codes available. This is highly suspicious as it is unlikely that a contract is being replicated so many times without it's source code being available.

Figure 5.4 plots the number of contracts (in sorted order of most duplicates) and the corresponding

Contract Bytecode Hash	src code	Contract Name/ Owner	Freq
2bf69ddcf80f6b24f2e6a8bf1454f662	Y	User Wallet	651930
fa00c5b8d83dbf920aec56d52c1df224	N	?	158186
55f0329f9e5dbac461e933c66e0e29b5	N	?	115132
dfcc91bcdc37abae7e8e9c82d57fbf6d	Y	Forwarder	99548
702edb219bba3238d55b2b38c759798b	N	?	90489
923d7eaf6e90eb272493d3ca5c5859d5	N	?	78018
7b63bae3ec81aa70d809a091240dccaa	N	?	42868
62dbffb5cce3d14500568320ab6dcd75	N	?	40456
1ae99eb3c89152c83cf788a5e7df4532	N	?	37534
125fb7c1ad488e0d0b9b034cfd12a977	N	?	28255

Table 5.1: Top 10 most duplicated contracts in the on-chain data-set

percentage of the total data-set they cover. It is observed that the top hundred contracts (0.1% of the data-set) amount to a total of 90.28% (or 1.72M occurrences) of the data-set.

We call these hundred contracts **High Occurrence Targets**.

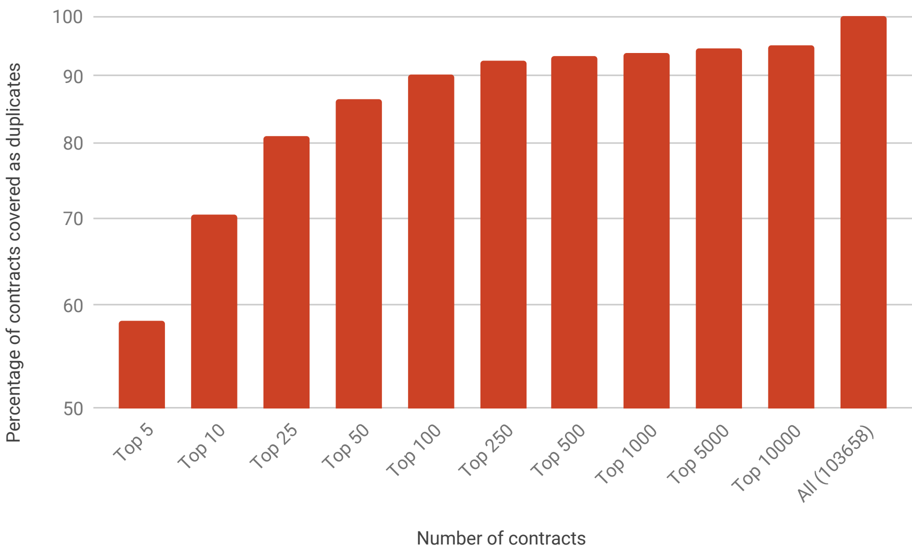

Figure 5.4: Duplicity in the smart contracts

5.2.2 Ether Balance

Every address (contract or normal) has an associated balance with it. For all the 1.9M contract addresses in our data-set, we leverage `geth`'s `getBalance()` method to find the ether balance. We find that the collected contracts contain a total of 10.88M Ether (worth roughly US$1.66B[1]).

However, 93% (or 1.77M) of the contracts had zero balance. The most valuable contract is

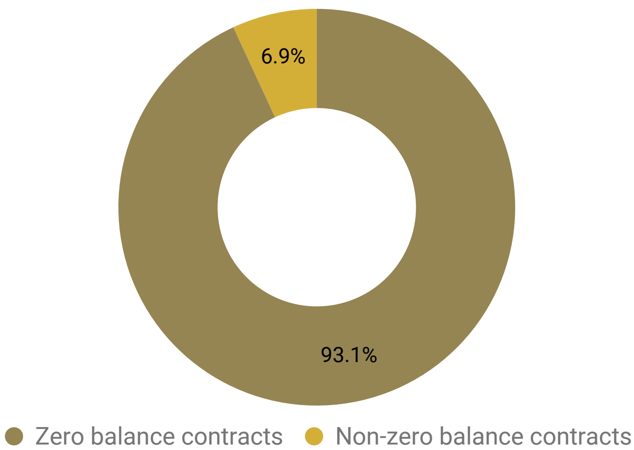

Figure 5.5: Percentage of zero-balance contracts

Wrapped Ether (WETH9) with roughly 2.4M Ether (worth roughly around US$367M). This contract essentially wraps your Ether into wETH that can be further used to trade with other ERC-20 compliant alt-coins.[67] This single contract holds nearly 22% of all the ether in contracts.

[1]as per exchange rate on 26 April, 2019

Table 5.2 shows the most valuable contracts in our dataset after considering duplicates. We

Contract Bytecode Hash	src code	Contract Name/ Owner	ETH Balance
0e10248e905a92ffd070393ea0a2890c	Y	WETH9	2383501
e6bf83c2201a1b2070a529a4f4814488	Y	Wallet	1430029
d62d28a4c3fad57a5158297ce6efed9e	N	Gemini's Cold Wallet	895999
108810ec1a9185e325c05b24f1a1c21e	N	Ethereum Foundation	645173
f5c40e048aca031a2ea4f32ba04646e1	Y	MultiSigWalletWithDailyLimit	600341
c7010bf53217a9fe2fc6ae82cf19907d	Y	MultiSigWalletWithDailyLimit	568083
4143e0500473d8164fae03423612e9e4	Y	Wallet	515035
c5fa4304659be1268f19e04c1a4add89	Y	MultiSigWallet	395432
cdb19b39b2dc549cd112a1a9ca3197ac	Y	MultiSigWallet	368023
1f086f1ded7ac3799011e0d6526bc436	N	?	292524

Table 5.2: Top 10 most valuable contracts in the on-chain data-set

see that contracts like MultiSigWalletWithDailyLimit and MultiSigWallet appear more than once. This is because even though they are similar contracts, they have been compiled using different `solc` versions, and therefore generate unique bytecode. Also, as expected we observe that wallet contracts store the most amount of Ether.

Figure 5.6 plots the number of contracts (sorted in non increasing order of ether balance) and the

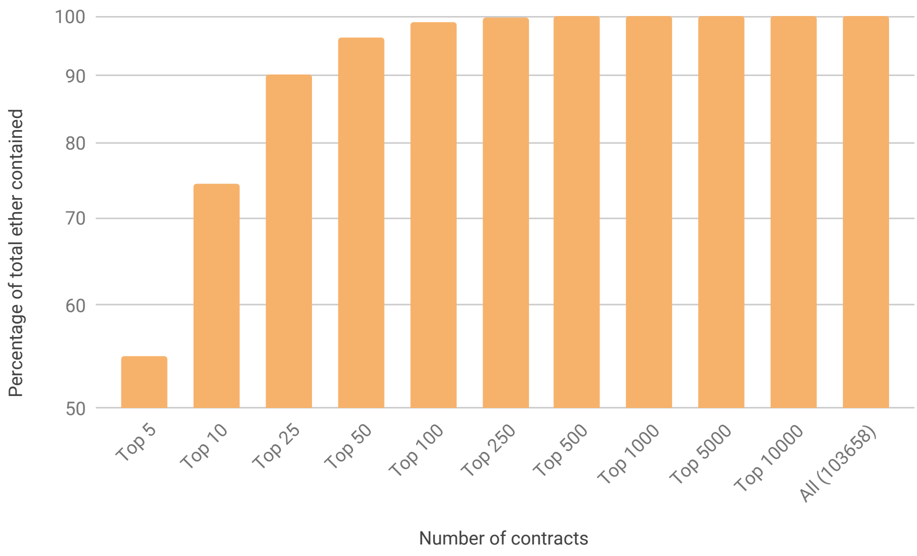

Figure 5.6: Distribution of Ether in Smart Contracts

corresponding percentage of the total ether-value they cover. It is observed that the top hundred contracts (0.1% of the data-set) amount to a total of 98.86% of the total ether-value in smart contracts (or 10.7M ETH). This ether is worth around US$1.6B.

We call these hundred contracts **High Value Targets**.

5.2.3 Number of Transactions

Another metric we looked into when analysing the on-chain data-set is the interactions other addresses and contracts on the network have with that smart contract. For that we looked at the number of transactions each contract was involved in. More the number of transactions for a contract, more it's interaction with others in the network and more it's security impact as well. As before, we also take care of duplicates in our analysis and only consider the total across a particular duplicate group.

We observe that smart contracts in our data-set were present at either the sending or the receiving end in 175M transactions (which is roughly 46% of all the transactions we went through). Therefore almost 1 in 2 transactions involves a smart contract. This further highlights the importance of having secure contracts.

It is also worth noting that 434K contracts (22.7%) had only one recorded transaction on the blockchain (most likely the contract creation transaction).

The single contract with the most number of transactions is EtherDelta with 5.2M transactions,

Contract Bytecode Hash	src code	Contract Name/ Owner	Number of Txns
2bf69ddcf80f6b24f2e6a8bf1454f662	Y	UserWallet	5833642
91f778605c0976e25dc93fc4a591bb96	Y	EtherDelta	5272000
de379d9a60e5f52e45c99874eb61bc70	Y	IDEX Exchange	3966000
f779bf5757253c974724c46b1e9f441e	Y	KittyCore (CryptoKitties)	3141000
f7e3b0272ca30480eb26b89d198f4c84	Y	DSToken (EOS)	2955507
ee302cfe0db1c206484facb2c9543751	Y	TronToken (Tronix)	1990664
748f8df24378a8d1dd82d44e598c46f4	Y	HumanStandardToken	1904722
c24bf798c5a50007d907a5d397cc46b0	N	Poloneix Exchange	1766954
fa00c5b8d83dbf920aec56d52c1df224	N	?	1681369
ca200836bce3f6fcb8bc9bbd2b34b6c9	Y	Controller	1578593

Table 5.3: Top 10 most interacted-with contracts in the on-chain data-set

while the contract group (after considering duplicates) with the most number of transactions is UserWallet with 5.8M transactions.

Figure 5.7 plots the number of contracts (in non-increasing order of number of transactions) and the corresponding percentage of the total transactions done by smart contracts that they cover. It is observed that the top 2500 contracts (2.5% of the data-set) amount to a total of 90.37% of the total smart contract transactions (or 158M transactions).

We call these 2500 contracts **High Interaction Targets**.

5.2.4 Value of Transactions

For each of the 1.9M contracts in our data-set, we also calculated the total value of transactions (in ETH) that it was involved in (on either side of the transaction). This gave us an idea of which

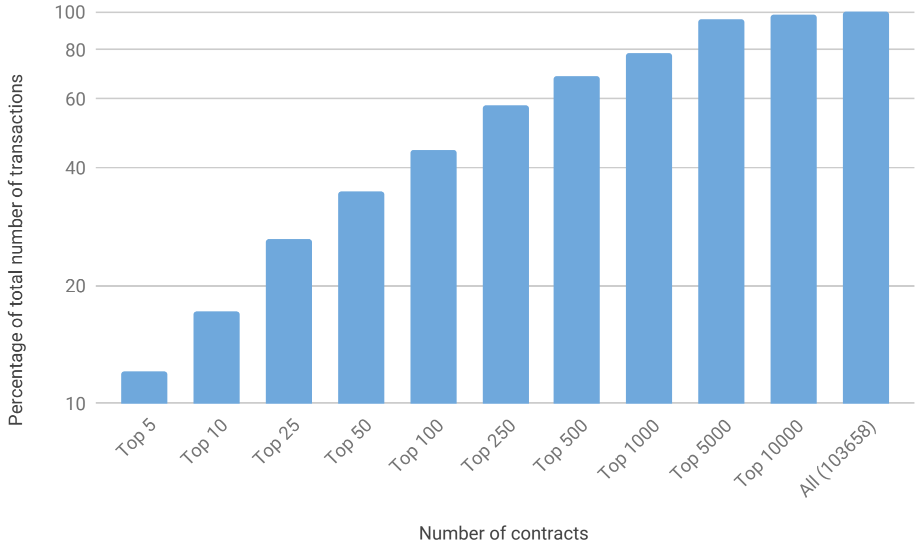

Figure 5.7: Distribution of Number of Transactions in Smart Contracts

contracts were involved in moving the crypto-currency across the network.

Smart Contracts in our data-set have been involved in transactions valuing 484M ETH worth around US$73B. Interestingly, 877K contracts (45.9%) have not been involved in any transaction involving ether.

Contract Bytecode Hash	src code	Contract Name/ Owner	Value of Txns (in ETH)
9473b7b4e2820ec802fc3d3814f90916	Y	ReplaySafeSplit	85246694
ea63abcce55531452cf1f6fad33757cd	N	Bitfinex Exchange	50031528
c24bf798c5a50007d907a5d397cc46b0	N	Poloniex Exchange	42613873
b58e896a767147b204a8f9203c850c77	N	Kraken Exchange	41267671
70d285d1aa9cb3b94ce39ef82a59bf6d	N	Gemini Exchange	24808753
dc5b7eea9e5e2308d7efbfc6c33f4d96	N	?	21104195
e6bf83c2201a1b2070a529a4f4814488	Y	Wallet	18243092
aec1a8e9808dd257802994ae9bc737d4	Y	ReplaySafeSplit	15547424
709738b34faa8702cbd4568ff5c2e382	Y	DAO	11983614
108810ec1a9185e325c05b24f1a1c21e	N	Ethereum Foundation	11911040

Table 5.4: Top 10 most ether-moving contracts in the on-chain data-set

Figure 5.8 plots the number of contracts (in non-increasing order of the total ether moved) and the corresponding percentage of the total ether moved done by smart contracts. It is observed that the top 100 contracts (0.1% of the data-set) have moved a total of 459.4M ETH (94.89% of the total ether moved by smart contracts). This ether is valued at US$69.89B.

We call these hundred contracts **High Ether Moving Targets**.

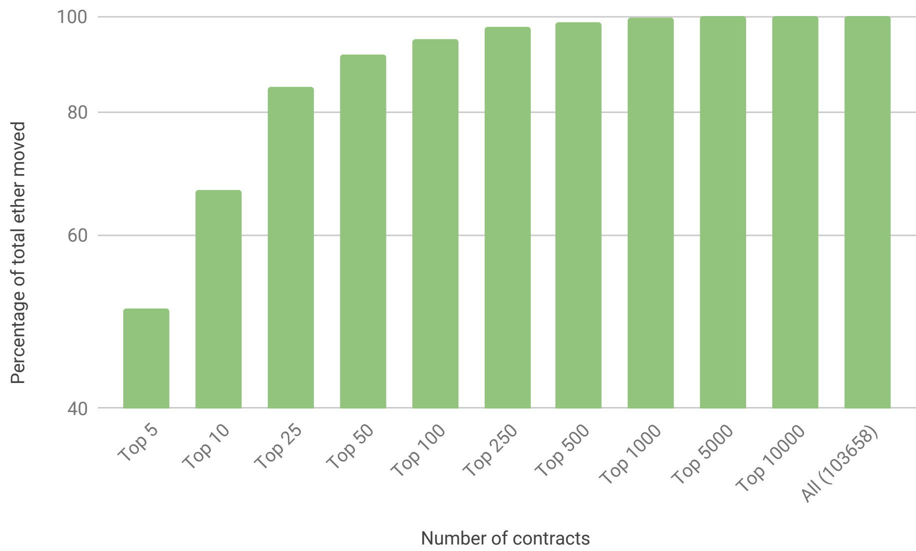

Figure 5.8: Distribution of the total Ether moved by Smart Contract over all it's transactions

5.2.5 Contract Creation Analysis

Next we move our analysis to the contract creation transaction for every contract in our data-set. For all the 1.9M contracts we find the transaction which created the contract. However, this leaves us with many contracts without any creation information. After further investigation we realized that many contracts in our data-set have been created by 'internal' transactions and are therefore not present on the blockchain. Therefore, to get their information we leverage Etherscan's API to get the internal transaction information as well. Finally, we were able to get the contract creation information of all the 1.9M contracts.

Figure 5.9 shows the distribution of the contract deployment mechanisms in our data-set. Surprisingly 60.6% of our data-set has been deployed by contract internal transactions.

Interestingly, we also observed that for many addresses, the contract creation transaction was not the first transaction(or internal transaction) for that address. This happened 760 times for contracts deployed using normal transactions and 6 times for contracts deployed using internal transactions. This is because the Ethereum Virtual Machine has no way to check the validity of a particular address and ether may be sent to an address which is not yet claimed by any individual or smart contract. Common reasons for such anomalies seem to be pre-funding of smart contracts and mistakes by the developers.

Figure 5.10 shows the deployment of smart contracts over-time. We see that the trend closely resembles the price graph of crypto-currencies like Bitcoin and Ethereum with a surge near the end of 2017 and interest slowing down after that.

We observed that the average gas used for contract deployment is 318K gas. Also, the oldest

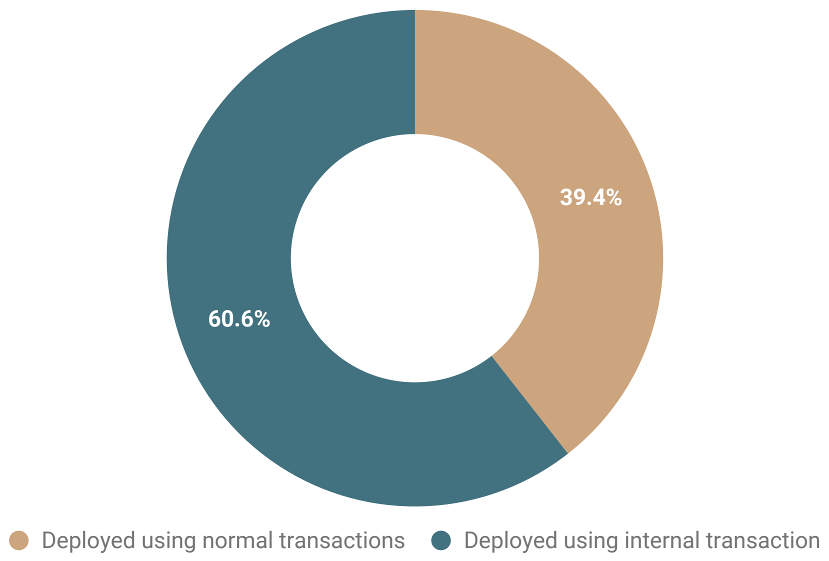

Figure 5.9: Distribution of deployment means for the smart contracts in the data-set

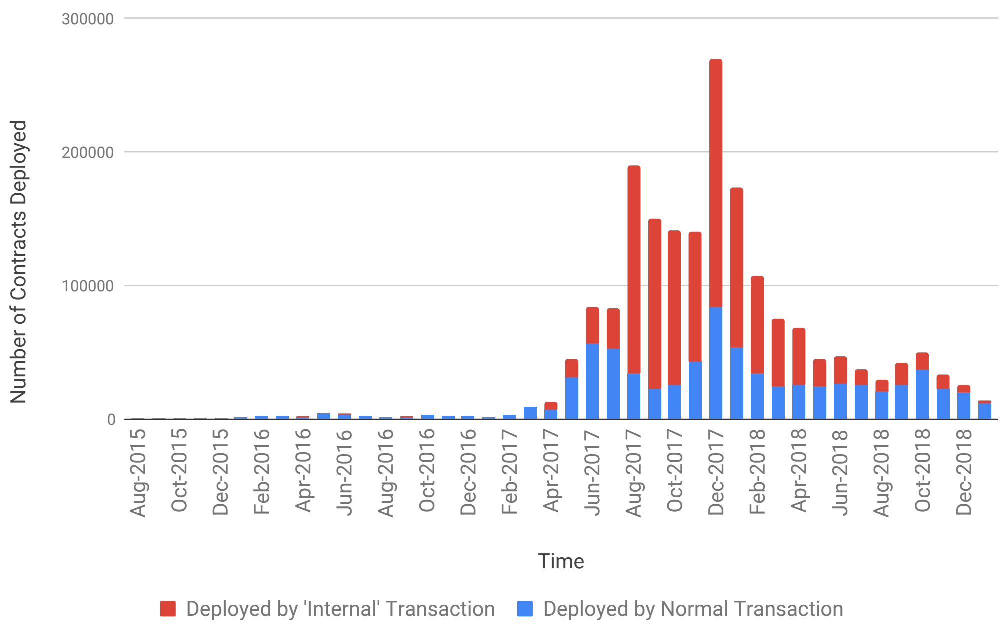

Figure 5.10: Smart Contract deployments per month

contract in our data-set is `0x6516298e1C94769432Ef6d5F450579094e8c21fA` which was deployed on 7^{th} August, 2015.

Next, we look at the addresses and the contracts which are involved in deploying these contracts -

- Out of the 753K contracts deployed using normal transactions, we find that they have been deployed by only **57,600 accounts** on the blockchain. The actual number may be far less than this as there is no restriction on the number of accounts an individual can own. The top ten contract creating addresses are listed in Table 5.5. We also observe that these top ten accounts don't create many new contracts, with most of them being exact duplicates.
- Out of the 1.16M contracts deployed using internal transactions, it is observed that these contracts have been deployed by only 9228 contracts. When we consider the duplicate creator-contracts as one, this number further reduces to **2420 contracts**

Address	Total Contracts Deployed	Unique Contracts Deployed
0xb42b20ddbeabdc2a288be7ff847ff94fb48d2579	158189	4
0x9862d074e33003726fa05c74f0142995f33a3250	78018	1
0x42da8a05cb7ed9a43572b5ba1b8f82a0a6e263dc	57921	10
0x2e05a304d3040f1399c8c20d2a9f659ae7521058	40456	1
0x17bc58b788808dab201a9a90817ff3c168bf3d61	37534	1
0x866f649cd9280d3dfa282372a3f5828839944959	15272	1
0xe35f12181a2748285358b63cff25887410d0804b	14174	10
0xa2635b3d63b4e31976419865e1a81553bb347be3	13191	11
0x174443351e21d47ed9ab51517a301107d92ede64	13105	1
0x0536806df512d6cdde913cf95c9886f65b1d3462	12098	1

Table 5.5: Top ten accounts creating the most smart contracts

Contract Bytecode Hash	Total Contracts Deployed	Unique Contracts Deployed
b071cebdaac85e6cfa20b9de78314386	651930	1
23eed1c018699f2aa9217e487851262b	115132	1
07459966443977122e639cbf7804c446	99547	1
1409c3e3b055b70674d7446f3d30df36	90489	1
366dd689499a9ec1538b94d3c57bbcb0	62313	2
1093df22ad83b4efffcb608adcff6569	11369	1
11f7b022128ec8ceaa93375856a2613c	7904	1
05b915a788451c8a5f4d715914d8ca5c	7654	1
2d382304429d30b1706f4089d19dd265	6693	1
1a5383732e90b3fdbf70d13dd90b2684	6144	1

Table 5.6: Top ten contracts creating the most smart contracts

Table 5.5 and Table 5.6 give some insights into where so many duplicate contracts are coming from. It was expected that similar contracts create similar child contracts, however we also observe that there are very few addresses (both account and contract addresses) which are responsible for the bulk of contract creation on the blockchain.

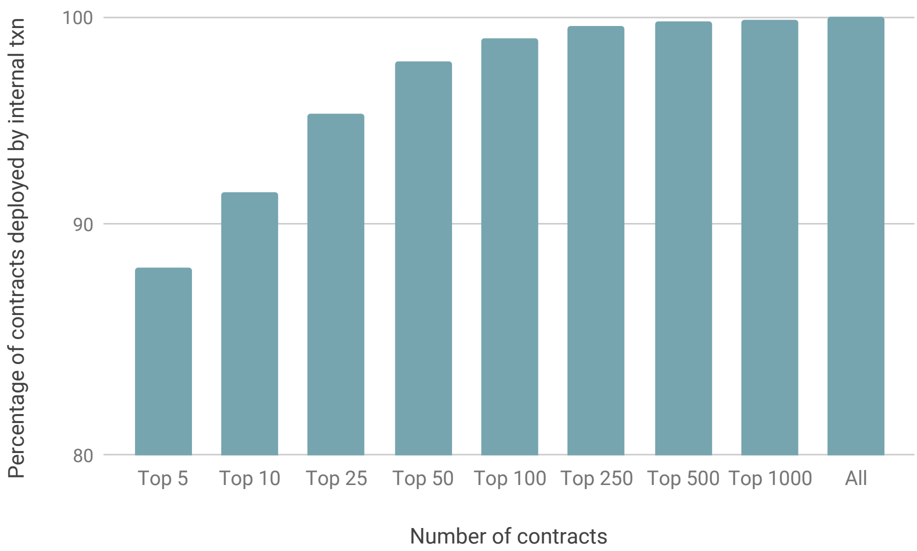

Figure 5.11: Distribution of Number of Contracts deployed by Internal Transactions

Figure 5.11 plots the number of contracts (in non-increasing order of the number of contracts deployed) and the corresponding percentage of the total number of contracts deployed. It is observed that the top 100 contracts have deployed 1.14M contracts (98.93% of the total contracts deployed by internal transactions).

We call these hundred contracts **High Origin Targets**.

5.2.6 Summary

We have analysed the on-chain smart contracts across various different parameters and we observe that a very small number of contracts are the most 'important' for each category. Finally we collect:

- 100 High Ether Moving Targets,
- 100 High Occurrence Targets,
- 100 High Origin Targets,
- 100 High Value Targets and,

- 2500 High Interaction Targets

These 2900 (2715 unique) smart contracts are called **'Contracts of Importance'**. Solidity files are available for 2053 (70%) of these contracts.

Figure 5.12 shows the intersection across the various categories. The high origin and high value contracts are the most independent, with there being significant overlap across the other categories. Surprisingly, we observe that two contracts are present across all the five categories.

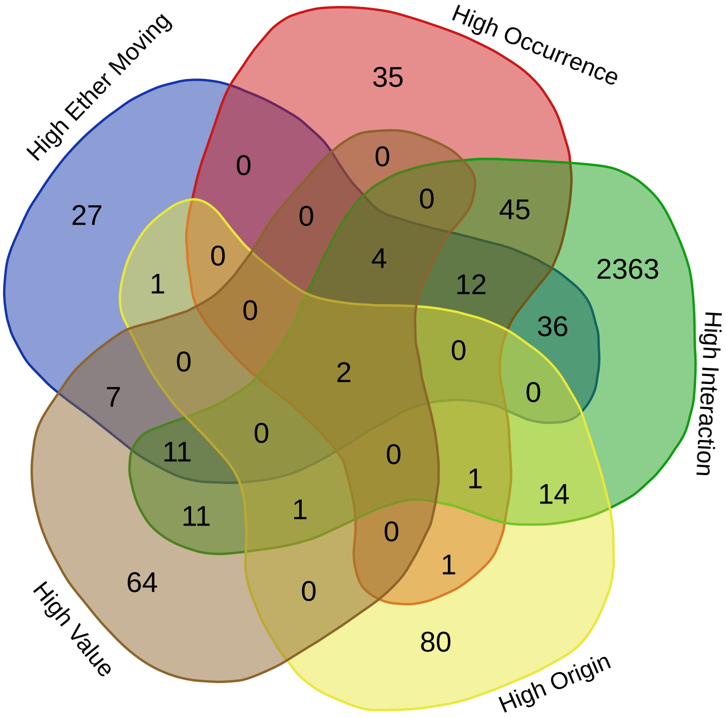

Figure 5.12: Intersection of Contracts of Importance

5.3 Security Analysis of On-Chain Contracts

5.3.1 Experiments with Different Tools

Static Analysis Tools

- **SmartCheck**

  Figure 5.13 shows the percentage of contracts reported as vulnerable by SmartCheck across the different categories it uses. Visibility issues, floating pragmas, old solidity versions, deprecated constructions are the most common vulnerabilities that are reported across all the five smart contract categories. High origin contracts are more vulnerable to locked money, inline assembly usage, unchecked call and `tx.origin` than the other contracts.

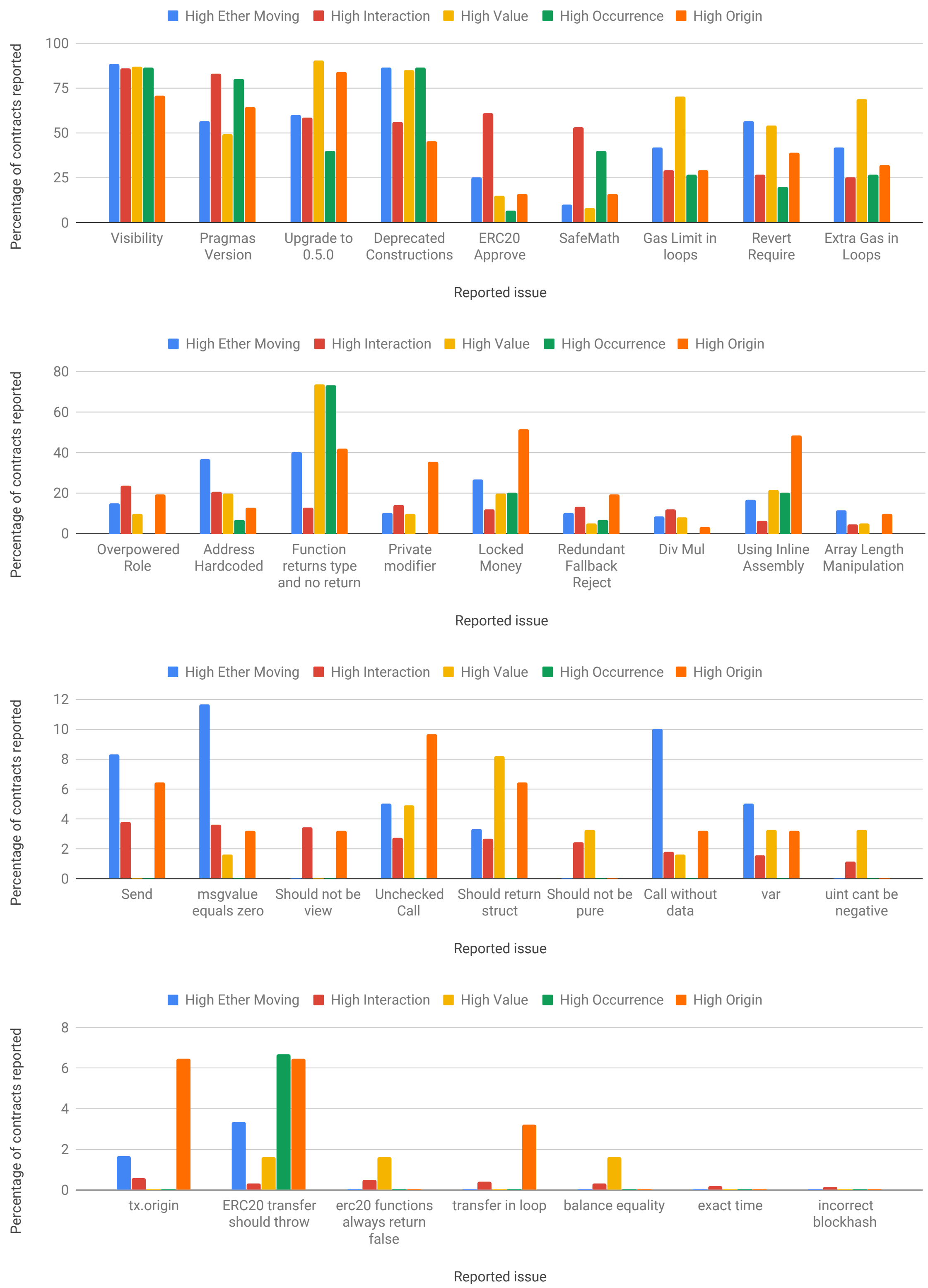

Figure 5.13: Results of SmartCheck on the Contracts of Importance

- **SolMet**

  Figure 5.14 shows some of the important results of SolMet on our Contracts of Importance. SLOC denotes the Source lines of Code, LLOC is the Logical lines of Code and CLOC is the comments line of code. Across all the categories, we observe relatively smaller files (< 400 LLOC). Also, we observe good commenting practices (nearly 1 in 3 logical lines have a comment). Therefore, readability of contracts whose source code is available should not be an issue.

  We also observe, that per solidity file, the high origin contracts have the maximum number of functions and contracts. This is expected as they have to contain the code of the contracts they create as well. The use of libraries is low with high origin and high interaction contracts averaging at nearly one library per contract file.

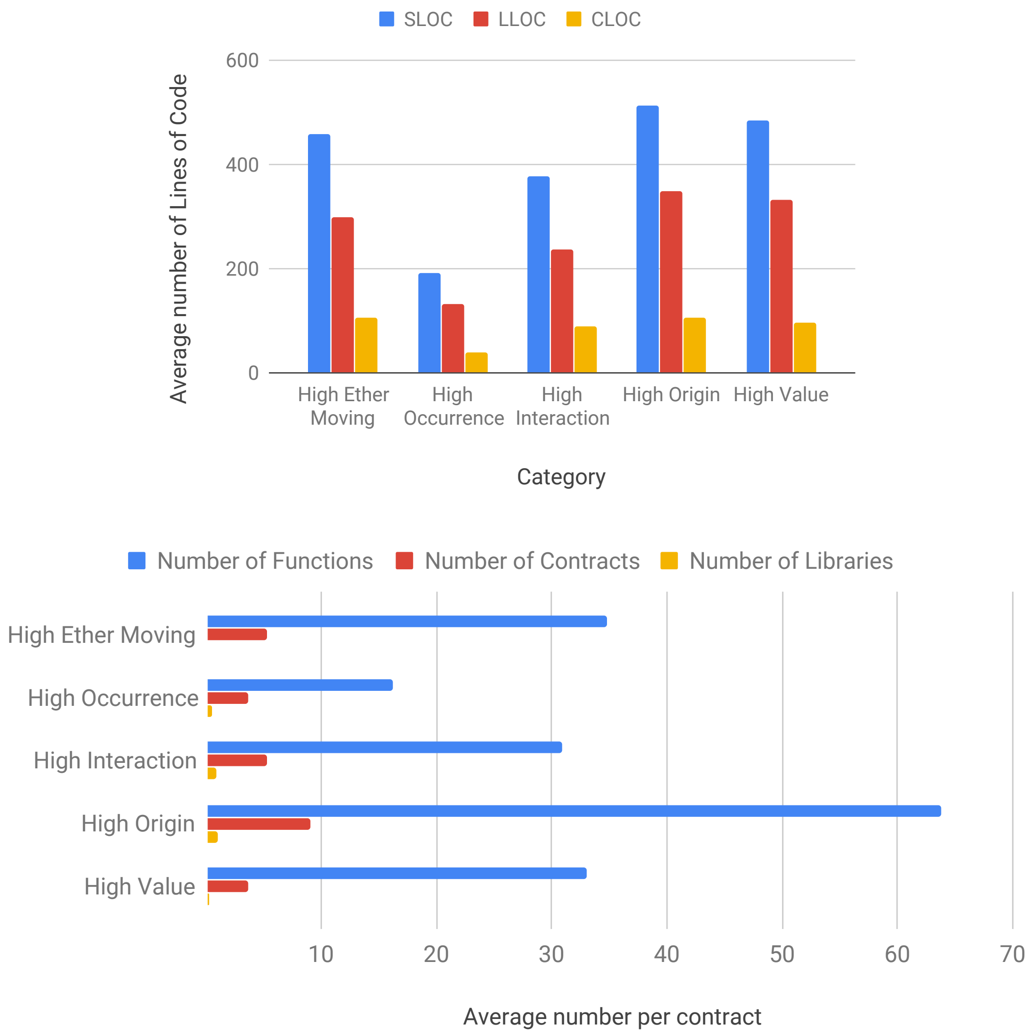

Figure 5.14: Results of SolMet on the Contracts of Importance

Symbolic Execution Tools

- **Oyente**

  Oyente is one of the oldest security tools for Ethereum smart contracts. The average EVM code coverage with Oyente was reported to be 67.81%. As shown in figure 5.15, money concurrency is the biggest issue in the contracts of importance, followed by time dependency and re-entrancy. Also, Oyente did not detect any overflows and underflows, like on the benchmark.

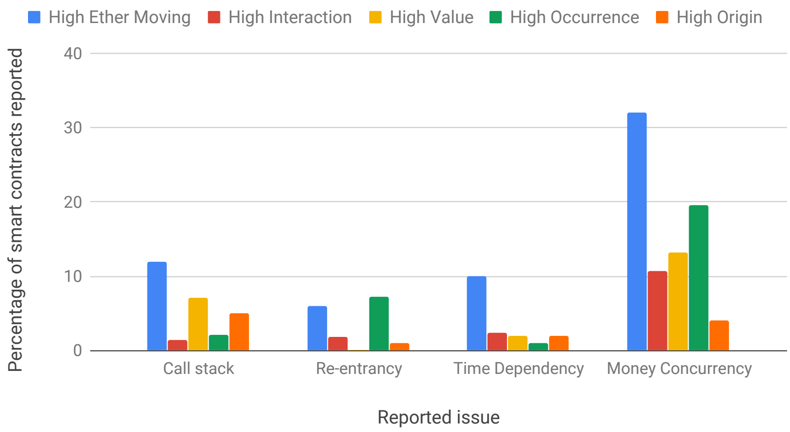

Figure 5.15: Results of Oyente on the Contracts of Importance

- **Securify**

  Securify is the only tool which marks the contract as safe too. The results of Securify on the contracts of importance as a whole are shown in in figure 5.16. DAO (re-entrancy) is the most prominent vulnerability reported, followed by missing input validation and repeated call. For markers like unrestricted Ether flow and unrestricted write, Securify did not give an output for more that 93% of the contracts and are therefore not shown in the graph.

- **Mythril**

  The results on Mythril show that multiple calls in a single transaction (which might lead to a denial of service attack) and dependence on predictable environment variable (bad sources of randomness) are the major vulnerabilities. However, critical issues like unprotected `selfdestruct`, unprotected Ether withdrawal, use of `tx.origin` also appear quite frequently. Also, since the tool was getting stuck on some contracts (for more than the day), the analysis was done by setting the `max-depth` parameter to 10.

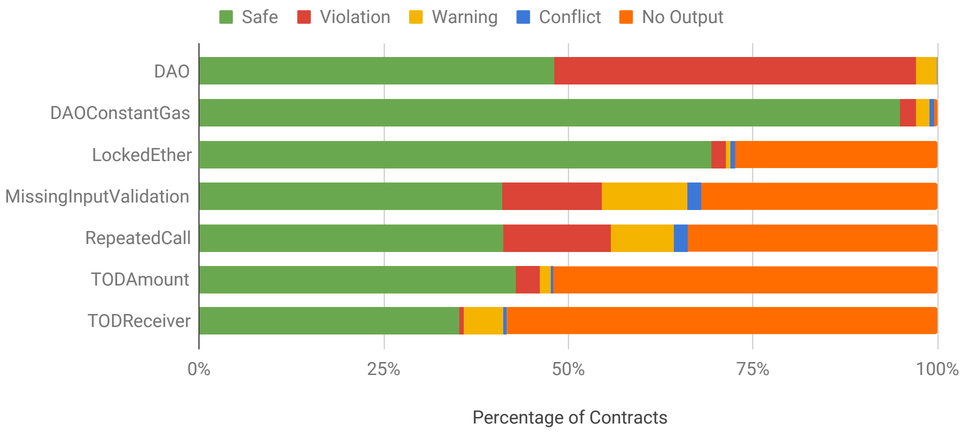

Figure 5.16: Results of Securify on the Contracts of Importance

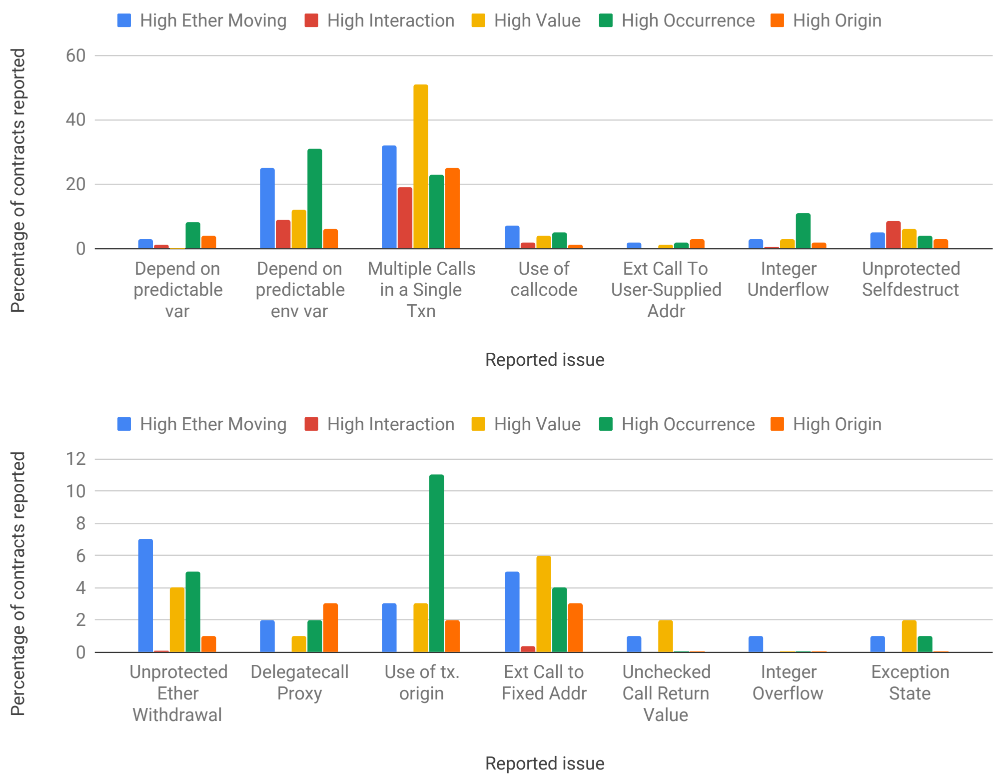

Figure 5.17: Results of Mythril on the Contracts of Importance

5.3.2 Analysis

The experiments for this section were carried out on Google Compute Engine `n1-highmem-8` instances with 8 vCPUs and 52GB RAM.

Figure 5.18 demonstrates the time taken by various tools on the *'Contracts of Importance'*. As expected, static analysis tools work much faster that symbolic execution tools. We also observe a larger gap between the maximum, minimum, average and median times for these tools.

Figure 5.19 shows that all tools (except Securify and Oyente on some instances) are able to analyse the *Contracts of Importance* successfully.

We also observe the following -

- Smart contracts are generally not too long. The use of libraries is less. However, one smart contract file usually contains more than one contract (roughly five on average). Therefore, tools should be cognizant of this fact when analyzing.
- Many of the on-chain contracts have become old (using outdated compiler versions or deprecated constructions). Also poor coding practices like costly loops, hard-coded addresses, using inline assembly frequently occur in the on-chain contracts
- There is a good chance that vulnerabilities like re-entrancy (DAO), transaction order dependence, bad randomness, unprotected `selfdestruct` might still exist in these contracts.

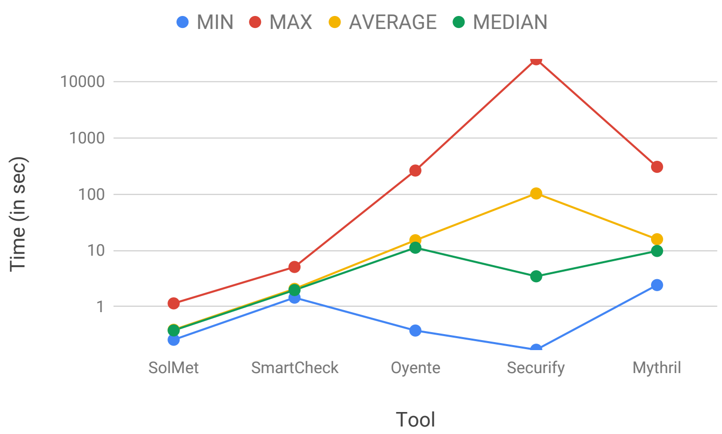

Figure 5.18: Time taken by the tools on the Contracts of Importance

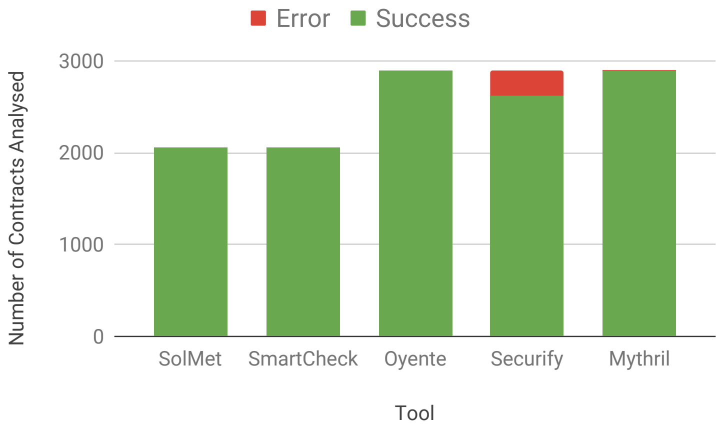

Figure 5.19: Percentage of Contracts of Importance successfully analyzed by the tools

5.4 Summary

In this section we have shown how the on-chain smart contracts were collected. We then analyse the smart contracts across different categories - duplicity, high ether balance, high number of transactions, etc. Surprisingly, we observe that only a small fraction of the contracts dominate each category that we analysed. This points to the smart contract space being not so *decentralized*. We observe that even though there are no banks, the exchanges and wallet contracts take their place. Therefore it becomes even more crucial to check these contracts and make sure that they are secure. For further analysis, we identified *Contracts of Importance* - a collection of the most important contracts from each of the categories that we studied.

We further study these contracts of importance using the different tools. We observe that most of the contracts are older and the biggest issue seems to be improper coding practices. The security tools also identified vulnerabilities in these contracts. However, if any of the warnings is true, it can be quite disastrous as these contracts dominate the blockchain. Any security flaw in these contracts will likely become a big issue. We also observe that the size of the contracts (in lines of code) is quite small as compared to normal large pieces of software. Therefore, using more intensive security auditing techniques and practices should not be a big issue.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this work we look at Ethereum smart contracts from a security viewpoint. We start by studying the various security vulnerabilities and developed a new taxonomy to help security researchers find the root causes for the security issues. We then look at the various security tools available for researchers and end users and test their effectiveness at detecting various security issues. For this, we develop a benchmark of insecure smart contracts collected from various sources. We observe that this is a ripe area for further research as there exists a lot of area that is still left uncovered by the present tools.

We then move on to the contracts that are present on the Ethereum main-net. We wrote scripts that collected byte-codes for 1.9M smart contracts and solidity files for 887K contracts (from Etherscan). After this we analyse the collected contracts across different parameters like duplicity, ether balance and number of transactions. We also analyse how the smart contracts on the blockchain are being created. Across all the categories we observe that a very small fraction of the smart contracts dominate the others. This led us the create *'Contracts of Importance'* - a collection of 2715 smart contracts that are relevant across the different categories analysed. We further analyse these contracts using the various security and analysis tools to identify the current trends in contracts that are being deployed on the chain.

6.2 Future Work

Since this is a ripe area for further research, there are many directions this work can be extended to -

- **Replicate study for other blockchains** - This study was conducted only for the Ethereum blockchain. However, many other blockchains with similar capabilities have been developed over time and similar studies can be carried out for them as well.

- **Improving and Maintaining the Taxonomy and Benchmark** - New security vulnerabilities keep popping up and therefore the taxonomy and the benchmark need to be updated periodically so that they stay relevant. Help from the open-source community can also be taken for this cause.

- **Formal verification of relevant on-chain smart contracts** - Since we realise that only a small number of contracts are actually 'important' - formal verification of these contracts becomes a feasible option.

- **Try exploiting the reported vulnerabilities** - We observe that the tools have reported different vulnerabilities for the on-chain contracts. It would be interesting to see how many of these can actually be exploited.

Bibliography

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, 2017.

[2] CoinMarketCap. Top 100 Cryptocurrencies by Market Capitalization. `https://coinmarketcap.com/` [Online; Accessed on: 13 May, 2019].

[3] Wikipedia contributors. The dao (organization) — Wikipedia, the free encyclopedia, 2019. `https://en.wikipedia.org/w/index.php?title=The_DAO_(organization)&oldid=895983593` [Online; accessed 13-May-2019].

[4] Haseeb Qureshi. A hacker stole $31M of Ether — how it happened, and what it means for Ethereum. `https://medium.freecodecamp.org/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce` [Online; Accessed on: 13 May, 2019].

[5] Satoshi Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[6] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[7] LM Bach, B Mihaljevic, and M Zagar. Comparative analysis of blockchain consensus algorithms. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1545–1550. IEEE, 2018.

[8] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 2014.

[9] Ethereum Community. Ethereum Homestead Documentation. `http://ethdocs.org/en/latest/index.html` [Online; Accessed on: 10 May, 2019].

[10] Nick Szabo. The idea of smart contracts. *Nick Szabo's Papers and Concise Tutorials*, 6, 1997.

[11] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.

[12] Luit Hollander. The Ethereum Virtual Machine – How does it work? `https://medium.com/mycrypto/the-ethereum-virtual-machine-how-does-it-work-9abac2b7c9e` [Online; Accessed on: 10 May, 2019].

[13] Ardit Dika. Ethereum smart contracts: Security vulnerabilities and security tools. Master's thesis, NTNU, 2017.

[14] Alexander Mense and Markus Flatscher. Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*, iiWAS2018, pages 375–380, New York, NY, USA, 2018. ACM.

[15] Vitalik Buterin. Thinking About Smart Contract Security. `https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/` [Online; Accessed on: 2 May, 2019].

[16] Monika Di Angelo and Sophia Antipolis. A survey of tools for analyzing ethereum smart contracts. 2019.

[17] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'Reilly Media, 2018.

[18] Iuon-Chang Lin and Tzu-Chun Liao. A survey of blockchain security issues and challenges. *IJ Network Security*, 19(5):653–659, 2017.

[19] Hal Finney. Best practice for fast transaction acceptance - how high is the risk? https://bitcointalk.org/index.php?topic=3441.msg48384#msg48384 [Online; Accessed on: 24 April, 2019].

[20] Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in bitcoin. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 906–917. ACM, 2012.

[21] Ghassan O Karame and Elli Androulaki. *Bitcoin and Blockchain Security.* Artech House, 2016.

[22] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 515–532. Springer, 2016.

[23] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. *Communications of the ACM*, 61(7):95–102, 2018.

[24] Jianyu Niu and Chen Feng. Selfish Mining in Ethereum. *arXiv e-prints*, Jan 2019.

[25] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 129–144, 2015.

[26] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 2017.

[27] nick256. Smart Contract Security: Part 1 Reentrancy Attacks. https://hackernoon.com/smart-contract-security-part-1-reentrancy-attacks-ddb3b2429302 [Online; Accessed on: 24 April, 2019].

[28] ConsenSys. Ethereum Smart Contract Best Practices – Known Attacks. https://consensys.github.io/smart-contract-best-practices/known_attacks/ [Online; Accessed on: 24 April, 2019].

[29] Paweł Bylica. How to Find $10M Just by Reading the Blockchain. https://medium.com/golem-project/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95 [Online; Accessed on: 10 May, 2019].

[30] Consensys. Smart Contract Weakness Classification and Test Cases. https://smartcontractsecurity.github.io/SWC-registry/ [Online; Accessed on: 2 May, 2019].

[31] Hudson Jameson. Security Alert: Ethereum Constantinople Postponement. https://blog.ethereum.org/2019/01/15/security-alert-ethereum-constantinople-postponement/ [Online; Accessed on: 10 May, 2019].

[32] Tony UcedaVelez. OWASP Risk Rating Methodology. https://www.owasp.org/index.php?title=OWASP_Risk_Rating_Methodology&oldid=247702 [Online; Accessed on: 25 April, 2019].

[33] Raine Revere. solgraph. https://github.com/raineorshine/solgraph [Online; Accessed on: 16 May, 2019].

[34] Crytic. rattle. https://github.com/crytic/rattle [Online; Accessed on: 16 May, 2019].

[35] Nick Johnson. evmdis. https://github.com/arachnid/evmdis [Online; Accessed on: 16 May, 2019].

[36] Raghav Dua. EthLint. `https://github.com/duaraghav8/Ethlint` [Online; Accessed on: 16 May, 2019].

[37] Peter Hegedus. Towards analyzing the complexity landscape of solidity based ethereum smart contracts. *Technologies*, 7(1):6, 2019.

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

[39] Crytic. Slither, the Solidity source analyzer. `https://github.com/crytic/slither` [Online; Accessed on: 2 May, 2019].

[40] Ethereum Foundation. Remix - Solidity IDE. `https://remix.ethereum.org/` [Online; Accessed on: 2 May, 2019].

[41] ConsenSys. Mythril Classic. `https://github.com/ConsenSys/mythril-classic` [Online; Accessed on: 16 May, 2019].

[42] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269. ACM, 2016.

[43] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016.

[44] Crytic. (Not So) Smart Contracts. `https://github.com/crytic/not-so-smart-contracts` [Online; Accessed on: 2 May, 2019].

[45] Consensys. EVM Analyzer Benchmark Suite. `https://github.com/ConsenSys/evm-analyzer-benchmark-suite` [Online; Accessed on: 2 May, 2019].

[46] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. Attacks — A Survey of Attacks on Ethereum Smart Contracts. `http://blockchain.unica.it/projects/ethereum-survey/attacks.html` [Online; Accessed on: 2 May, 2019].

[47] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting ponzi schemes on ethereum: identification, analysis, and impact. *arXiv preprint arXiv:1703.03779*, 2017.

[48] NCC Group. DASP — TOP 10. `https://dasp.co/index.html` [Online; Accessed on: 2 May, 2019].

[49] Adrian Manning. Solidity Security: Comprehensive list of known attack vectors and common anti-patterns. `https://blog.sigmaprime.io/solidity-security.html` [Online; Accessed on: 2 May, 2019].

[50] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts. `https://blog.peckshield.com/2018/04/25/proxyOverflow/` [Online; Accessed on: 2 May, 2019].

[51] Arseny Reutov. Predicting Random Numbers in Ethereum Smart Contracts. `https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620` [Online; Accessed on: 2 May, 2019].

[52] Parity Technologies. Parity: Security Alert. `https://www.parity.io/security-alert-2/` [Online; Accessed on: 2 May, 2019].

[53] CityMayor. How Someone Tried to Exploit a Flaw in Our Smart Contract and Steal All of Its Ether. `https://blog.citymayor.co/posts/how-someone-tried-to-exploit-a-flaw-in-our-smart-contract-and-steal-all-of-its-ether/` [Online; Accessed on: 2 May, 2019].

[54] Michael Yuan. Building a safer crypto token. https://medium.com/cybermiles/building-a-safer-crypto-token-27c96a7e78fd [Online; Accessed on: 2 May, 2019].

[55] Samuel Falkon. The Story of the DAO — Its History and Consequences. https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee [Online; Accessed on: 2 May, 2019].

[56] KingoftheEther. Post-Mortem Investigation (Feb 2016). https://www.kingoftheether.com/postmortem.html [Online; Accessed on: 2 May, 2019].

[57] Crytic. solc-select. https://github.com/crytic/solc-select [Online; Accessed on: 2 May, 2019].

[58] SmartDec. SmartCheck. https://tool.smartdec.net/ [Online; Accessed on: 2 May, 2019].

[59] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. Ethir: A framework for high-level analysis of ethereum bytecode. In *International Symposium on Automated Technology for Verification and Analysis*, pages 513–520. Springer, 2018.

[60] Melonport. Oyente. https://oyente.melonport.com [Online; Accessed on: 2 May, 2019].

[61] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82. ACM, 2018.

[62] ChainSecurity. Securify Scanner for Ethereum Smart Contracts. https://securify.chainsecurity.com [Online; Accessed on: 16 May, 2019].

[63] StopAndDecrypt. The Ethereum-blockchain size has exceeded 1TB, and yes, it's an issue. https://hackernoon.com/the-ethereum-blockchain-size-has-exceeded-1tb-and-yes-its-an-issue-2b650b5f4f62 [Online; Accessed on: 29 April, 2019].

[64] Consensys AG. Infura - Scalable Blockchain Infrastructure. https://infura.io/ [Online; Accessed on: 29 April, 2019].

[65] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1317–1333, 2018.

[66] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS*, pages 18–21, 2018.

[67] Radar-Relay-Inc. wETH — ERC20 tradable version of ETH. https://weth.io/ [Online; Accessed on: 26 April, 2019].

[68] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, and Huaimin Wang. Blockchain challenges and opportunities: A survey. *Work Pap.–2016*, 2016.

[69] Harry Halpin and Marta Piekarska. Introduction to security and privacy on the blockchain. In *2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 1–3. IEEE, 2017.

[70] Ghassan O Karame and Elli Androulaki. Security of transactions in bitcoin. In *Bitcoin and Blockchain Security*, pages 59–83. Artech House, 2016.

[71] Tanya Bahrynovska. History of Ethereum Security Vulnerabilities, Hacks and Their Fixes. https://applicature.com/blog/blockchain-technology/history-of-ethereum-security-vulnerabilities-hacks-and-their-fixes [Online; Accessed on: 25 April, 2019].

[72] Wikipedia contributors. Blockchain – Wikipedia, the free encyclopedia, 2019. https://en.wikipedia.org/w/index.php?title=Blockchain&oldid=895749643 [Online; Accessed 8-May-2019].