University of Camerino

School of Science and Technology

Master Degree in Computer Science (LM-18)

Ethereum Smart Contracts Optimization

Advisor
Prof.ssa Barbara Re

Candidate
Margherita Renieri

Co-Advisors
Prof. Fausto Marcantoni
Prof. Andrea Morichetta

ACADEMIC YEAR 2019/2020

“ ...Molti anni fa, ho letto una frase
che diceva che le persone non falliscono perché
mirano troppo in alto e sbagliano, ma perché
mirano troppo in basso e fanno centro. ”
Matteo Bussola, da *L'invenzione di noi due*

Contents

Abstract **7**

1 Introduction **8**
1.1 Motivation . . . 8
1.2 Aim of the research . . . 9
1.3 Structure of the thesis . . . 10

2 Background **11**
2.1 Blockchain . . . 11
2.1.1 Blockchain 1.0 . . . 15
2.1.2 Blockchain 2.0 . . . 16
2.1.3 Blockchain 3.0 . . . 20
2.1.4 Blockchain 4.0 . . . 20
2.2 Smart contracts . . . 21
2.2.1 History . . . 21
2.2.2 Origin . . . 23
2.2.3 Explanation . . . 23
2.3 Ethereum . . . 26
2.3.1 Solidity . . . 32
2.3.2 EVM . . . 34

3 Vulnerabilities **37**
3.1 Relevant Surveys . . . 37
3.2 Ethereum Scenario . . . 40
3.2.1 Different Types of Forks . . . 40
3.2.2 Ethereum Roadmap . . . 41
3.2.3 Main Attacks . . . 43
3.3 Security Analysis Methods . . . 53
3.3.1 Static Analysis . . . 53
3.3.2 Dynamic Analysis . . . 56
3.4 Static Analysis Tool engine . . . 56
3.4.1 Tools . . . 61
3.4.2 The efficient of smart contract analysis tools . . . 66

4 Optimization Detectors **69**

4.1 Slither . . . 69
4.1.1 SlithIR . . . 74
4.1.2 SSA . . . 76
4.2 Gas-costly patterns . . . 77
4.2.1 External Transactions . . . 78
4.2.2 Useless Code Related Patterns . . . 79
4.2.3 Loop Related Patterns . . . 79
4.2.4 Saving Space . . . 79
4.2.5 Operations . . . 80
4.2.6 Additional . . . 81
4.3 Testing process . . . 82
4.4 Case study . . . 86

5 Conclusion and further direction **93**

Appendix A **95**

Bibliography **103**

List of Figures

2.1 Blockchain Evolution. . . . 11
2.2 Distributed Ledger Architecture. . . . 12
2.3 Traditional sharing document. . . . 13
2.4 Shared Centralized System. . . . 13
2.5 Blockchain Architecture. . . . 14
2.6 Merkle Tree Structure. . . . 14
2.7 Bitcoin's Ledger. . . . 15
2.8 Traditional contract compares to smart one. . . . 16
2.9 Block structure in blockchain system. . . . 19
2.10 Dapps Architecture. . . . 20
2.11 Blockchain related to Industry. . . . 21
2.12 Smart Contract System. . . . 24
2.13 Ethereum Network. . . . 26
2.14 Architecture of the Ethereum blockchain. . . . 26
2.15 Ethereum Accounts. . . . 27
2.16 Relation between the two classification of actions. . . . 28
2.17 Transaction Structure. . . . 29
2.18 Block Header Structure. . . . 30
2.19 Block synchronization process between nodes. . . . 32
2.20 Ethereum Virtual Machine Architecture. . . . 36

3.1 A classification of Ethereum vulnerabilities and their treatments. . . . 38
3.2 Soft fork mechanism. . . . 40
3.3 Hard fork mechanism. . . . 41
3.4 Ethereum improvements table. . . . 41
3.5 Ethereum's first transaction. . . . 42
3.6 DAO attack explanation. . . . 44
3.7 ERC-20 signature replay atteck via `transferProxy()`. . . . 49
3.8 MyEtherWallet team confirmation on Twitter. . . . 52
3.9 Example of a Directed Graph. . . . 55
3.10 Example of a subgraph. . . . 55
3.11 Smart Contract System. . . . 57
3.12 Oyente Architecture. . . . 61
3.13 Vandal Pipeline. . . . 62

3.14 Securify Architecture. . . . 63
3.15 Osiris Architecture. . . . 64
3.16 MadMax analysis pipeline. . . . 65
3.17 GASOL components. . . . 66
3.18 SolidiFI Workflow. . . . 67
3.19 False negative for each tool. . . . 68
3.20 False positive by each tool. . . . 68

4.1 Smart Contract System. . . . 69
4.2 Smart Contract System. . . . 70
4.3 Screen of the ERC-20's conformance on *StandardToken* contract in *unica.sol* file. . . . 73
4.4 ERC Logo. . . . 73
4.5 Values v_1 and v_2 mergetd into a unique v_3. . . . 76
4.6 Control flow graph of an IF and a WHILE statement with instruction in SSA form. . . . 76
4.7 First step of Slither test. . . . 82
4.8 Contract Summary of Staker contract. . . . 86
4.9 Human Readable Summary of Staker contract. . . . 88
4.10 Gas consumed before optimization. . . . 89
4.11 Gas consumed after optimization. . . . 91
4.12 ERC-20's conformance on Staker contract. . . . 91
4.13 Staker Inheritance Graph. . . . 92

List of Tables

2.1 Setting for each kind of transaction. 28
2.2 Table of the most used opcodes. 35

3.1 Categorization of attacks. 39

4.1 List of all implemented detectors 71
4.2 List of all Informational and Optimizational detectors 72
4.3 Selected Solidity contracts. 85

5.1 List of ten contracts tested with Slither. 96
5.2 List of ten contracts tested with Slither. 97
5.3 List of ten contracts tested with Slither. 98
5.4 List of ten contracts tested with Slither. 99
5.5 List of ten contracts tested with Slither. 100
5.6 Optimization test 101
5.7 Optimization test 102

Abstract

The Ethereum blockchain-based platform was launched in 2015. Its network is composed of nodes that execute smart contracts. Several vulnerabilities of different importance are discovered owing to the scripting natures of the Solidity language and the immutability features of blockchain.

The research takes into account the analysis of Ethereum blockchain smart contracts with a focus on gas optimization.

The excess of the maximum amount of gas generates improper programming problems with unbounded operations.

The objective of the thesis is to detect specific costly patterns, give possible solutions in order to optimize and improve smart contract performances. To achieve this goal, it was necessary to find the perfect methodology that allows detection.

The exploration of different security analysis methods used in the improvement of vulnerability detection frameworks, enables the selection of static methodology and, subsequently, the choice of a specific tool, Slither.

Slither was tested in order to understand all its functionalities. The focus was on the automated detection of vulnerabilities. A dataset of contracts taken from the public repository Etherscan was examined.

Thanks to Slither's interactive structure, the implementation of own targeted space-saving gas detectors was tested on the set of smart contracts.

The outcomes gave the general scenario of how important optimization is in smart contracts.

Introduction

Blockchain technology is an emerging technology that enables new forms of decentralized architectures.

From a data management point of view, a blockchain is a distributed database, which logs an evolving list of transaction records by organizing them into a hierarchical chain of blocks. From a security perspective, the blockchain is created and maintained using a peer-to-peer network. It offers an innovative methodology to store information, execute transactions, and perform functions combining private key cryptography, peer-to-peer networking, and the consensus protocol technologies in order to distribute digital information in a secure way.

The evolution from the first generation of blockchain to the second one has allowed the potential of this technology. Indeed, since 2015, thanks to the introduction of Smart Contracts by Ethereum, it has been possible to run programs on the blockchain. Smart contracts are programs that run on blockchains: their code and state is stored on the ledger, and they can send and receive coins.

A *focused security analysis of smart contracts* is thus crucial for the trust of the society in blockchain technologies and their widespread deployment. Unfortunately, this task is quite challenging for several reasons.

First of all, Ethereum smart contracts are developed in an ad-hoc language, Solidity, which is similar to JavaScript but features *specific transaction-oriented mechanisms*. Indeed, any action that requires modification of the blockchain costs gas, which corresponds to a fraction of the currency used by that given blockchain, and therefore to real money.

Secondly, smart contracts are uploaded on the blockchain in the form of Ethereum Virtual Machine bytecode, a stack-based low-level code featuring dynamic code creation and invocation and, in general, very little static information, which makes it extremely *difficult to analyze*.

1.1 Motivation

With the increasing popularity of Ethereum technologies in recent years, it has become the widely adopted platform which enables the transfer of currency. During the improvement of the research, it was evident that the Ethereum system has to face the *inadequate management* of the vulnerabilities and solution approach.

Moreover,there is a *lack of deep* understanding on the *rigorously* defined properties and analysis methodologies that are sufficient for analyzing the desired properties of blockchain-based systems. This undefined management related to this field makes it a challenging area of interest. As it is widely spread, several vulnerabilities turned up due to the user's freedom and a lack of protocols of security.

As far as smart contracts are concerned, they offer a particularly unique combination of security challenges. Due to the persistent nature of the blockchain, once initialized, the contract code cannot be updated.

Furthermore, contracts are relatively *difficult to test*, especially since their runtimes enable them to interact with external services and other contracts. In addition, they can be invoked repeatedly by transactions from a large number of users. Third, since coins on a blockchain have a crucial value, attackers are highly incentivized to find and exploit bugs in contracts that process or hold them directly for profit.

As a consequence, the issue of smart contract security is crucial. Additional problems derive from the **gas mechanism** which is typical and unique in Ethereum blockchain through which the smart contracts' execution is managed. The required gas is proportional to the amount of computational power required to perform the operation so it serves as an incentive system both for miners, to spend hardware and electricity costs to validate transactions, and against attackers, who would spend money to perform an attack. Gas corresponds to real money, and those transactions can fail because of an insufficient amount of gas. Furthermore, if on the one hand it is not easy to make an accurate estimation of the gas necessary for the execution of a specific smart contract, on the other hand the setting of a gas limit could be seen as a security setting, because without it if there is a bug all the gas will be wasted.

In addition, security attacks targeting smart contracts have sharply risen and there has been an increase in **financial losses** and in the **erosion of trust**. As several research highlights, smart contracts may contain bugs which can be exploited by malicious attackers for financial gains. So it is clear that it is fundamental to enable developers to find out security vulnerabilities in these contracts before their deployment. Moreover, the *immutable* and *irreversible nature of Ethereum transactions* increases the possibility of losses which cannot be recovered. There have been many bugs in smart contracts that have been detected in the recent past.

The identification and the analysis of the discovered vulnerabilities give the possibility to overcome security problems that are linked to them and have an impact on the whole system. Different typology of approaches has been developed to overcome these issues and each of them is related to a specific security analysis method. The most common categorization of methodologies splits them into static and dynamic analysis and formal verification methods.

A static analysis framework should provide a balance of several properties like a correct level of *abstraction* that enables an *accurate semantics* which captures common usage patterns, a *robustness* that allows to successfully analyze real-world code, *performance* and *accuracy*: it should allow for the development of detectors that find most potential issues while maintaining a low false positive rate.

1.2 Aim of the research

The research has followed several stages that aim to improve the knowledge of the theoretical and practical approaches related to **the need to make smart contracts much secure and optimized**. The next step has concerned the analysis of the most common security analysis techniques. Once the static method has been analyzed and picked, the Slither framework has been selected in order to test real-world codes chosen from the Etherscan public repository. The focus of the research was to **detect gas-costly patterns** and **provide advice** in order to *improve their efficiency*. It is difficult

to optimize smart contracts in terms of gas-efficiency, because it requires deep understanding of different characteristics such as EVM's instructions, gas consumption for different operations, and data locations accessed by operations. The research outlines a series of gas-costly patterns which are focused on saving gas.

One of the main issues of gas mechanisms is related to the fact that the *estimation in advance of the execution costs is difficul*t, with the risk of making the execution of the associated transaction fail. But the need to have gas saving techniques is therefore clear, indeed all operations used to manipulate memories cost gas so they have to be reconsidered. The most expensive ones are those affecting the Storage.

During the testing process, from the whole list of detectors, those related to the optimization field are selected and their results are analyzed. In addition, an *ad-hoc detector* was improved to detect used storage variables used in the line code of the targeted contract. The results of the testing phase allows to have an idea of all the possible changes that could be made in terms of efficiency.

1.3 Structure of the thesis

The thesis is structured in four chapters. In Chapter 2, we introduce the basic notion of **blockchain** and we take into consideration the roadmap that its evolution pursued. Then, we focus our attention on the *second generation* of blockchain technologies and we present the development of the concept of agreement and the introduction of **smart contracts** in the **Ethereum system** which is analyzed in a specific section.

In Chapter 3, we give an explanation of what a **vulnerability** is in a blockchain scenario. Looking in detail, we illustrate how Ethereum covers these pitfalls and then we analyze the most common *technical methods* used to implement security analysis on smart contracts and then, we introduce several **tools** that implemented those methodologies.

Chapter 4 is the core of the thesis. We present **Slither**, the framework picked to test a dataset of contracts whose resulting tables have done are in Appendix 5. Then, we highlight the **patterns** which are an area of interest for our research. Once we provide information about its scopes, we illustrate a case that enables us to show **Slither functionalities** and our contribution in order to detect an optimization problem which concerns the **unused storage variables** recognition.

Finally in Chapter 5, we sum up the goals achieded from the thesis and define possible **further improvements** that could be done in order to proceed with the work. Indeed, we introduce upgrades that take into account *Slither engine*, its *theoretical analysis methodology*, its *detectors selection* and a *different testing phase*.

Background

This first chapter is a presentation of the most important theoretical concepts of the thesis. It gives a detailed presentation of the environment which the research belongs to.

In Section 2.1, we illustrate the concept of **blockchain technology**, its *emergence* and its *evolution*. First of all, we introduce Bitcoin as an example of *Blockchain 1.0* following this, we show the *Blockchain 2.0* that deals with registering and transferring of smart contracts. Then, we take into account extension of Blockchain related to several domains like governance and education which is known as *Blockchain 3.0*. At the end, we provide information about *Blockchain 4.0* that is linked to the Industrial sector like finance in the form of digital assets, remittance and online payments or healthcare industry, and technology in IoT.

In Section 2.2, we introduce the concept of **contract**. Starting from its *traditional meaning*, we explain *its development* throughout the ages, then we focus our attention on smart contracts notion from the *revolutionary article* written by Nick Szabo and we provide *technical explaination* of those smart agreements.

In Section 2.3, we analyze **Ethereum**, the distributed ledger system taken into account in our research. Then, we breifly introduce the Ethereum environment and its engine. Moreover, we treat the *memory usage mechanism* used by EVM.

2.1 Blockchain

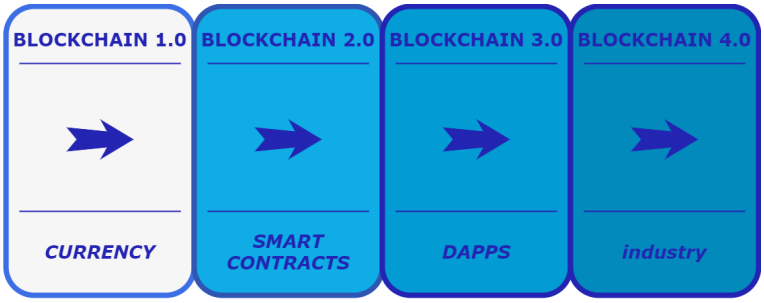

Figure 2.1: Blockchain Evolution.

Since the beginning of the new century there has been a slight increase in the usage of data and this has produced changes in different technological areas.

This evolution caused the introduction of new models and new architectures that are designed to integrate data and customer demands for providing better decisions and enrich the user experience.

Even if the amount of data has become increasingly large, the need to maintain

security standards is still the same, indeed it is fundamental in a distributed environment to ensure trust and transparency on processes over the blockchain.

In 2008, the article *Bitcoin: A Peer-to-Peer Electronic Cash System* [29], signed by Satoshi Nakamoto, introduced the concept of blockchain as a Bitcoin Cryptocurrency but nowaday it is used as a trusted platform that allows the exchange of any kind of services and transactions over the distributed network. It is evident that blockchain has revolutionized the digital economy by providing new dimensions to security and efficiency of systems. It represents an efficient solution in decentralizing transactions and providing consensus and trust among participating peers from several authoritative domains.

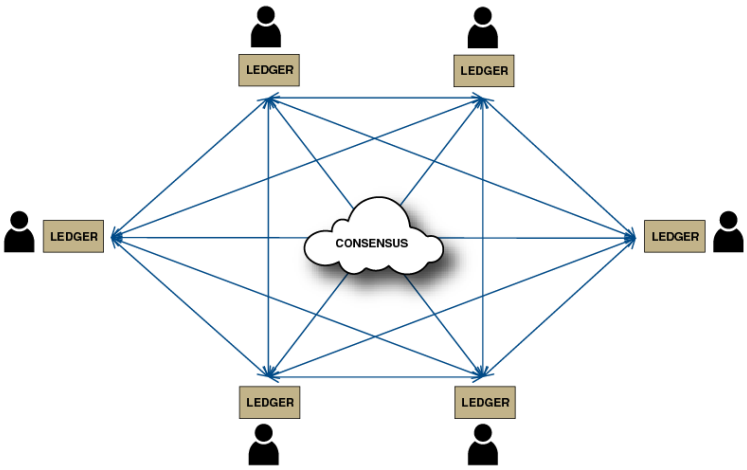

Figure 2.2: Distributed Ledger Architecture.

As the Figure 2.2 illustrates, blockchain is a shared and immutable ledger that keeps transactions in a distributed peer-to-peer network. The peers are nodes called *miners* and each one maintains a consistent copy of the ledger. Each Transaction is collected into blocks and it is hashed-chained with the previous block in the blockchain data structure.

Its decentralized characteristic has enabled service providers to register, confirm transactions via contracts and transfer credits without the presence of third trust parties. Every time miners collect their choice of new transactions in a new block. The transactions are hashed, verified and then mined into blocks that are added to the chain by miners using a consensus mechanism. This protocol uses a probabilistic algorithm for electing the miner who will publish the following block in the chain. Nodes have to perform a procedure called *proof-of-work* that is composed by the resolution of a computationally demanding cryptographic puzzle. The correctness of it is done by all others miners, in positive case they update their local copy of the chain adding the new block. The mined block is added to the longest valid chain and this rule ensures the immutability of the block, indeed any change to the block will also change the hash value and will provoke the invalidation of blocks. Thus, the valid chain contains a history of transactions as log that can be verified and created at any moment in the network.

In **traditional systems**, like the one represented in the Figure 2.3 where two parties have to share documents, both the users must have the updated shared data. In order to make some changes a user has to *download* the copy of the data over the network, make the changes and once the new copy is uploaded to the network, it is forwarded to the other user. This mechanism needs the use of multiple copies of the same data as

Figure 2.3: Traditional sharing document.

redundancy and makes the location of the most recent copy hard to find. Furthermore, a user has to wait for the other one to make changes and then only the exchange can proceed with modifications. The Figure 2.4 shows the evolution of previous scenario.

Figure 2.4: Shared Centralized System.

The advanced structure imposes the presence of a *cloud server* that resolves the issue of the recently updated version and waiting time.

In the **centralized shared system**, both parties which want to edit the document can share the same space simultaneously.

The main drawbacks of this approach are that all data is on a central server so it may soffer of *bottleneck* situations and operations may be performed based on network connectivity. In addition, cloud based systems faced many challenges like the insufficient bandwidth with high amount of jitter caused by the low Quality-of-Service and the heterogeneity of networks. Moreover, this scenario may have several security attacks because it is difficult to maintain *confidentiality*, *integrity* and *availability* so central servers are more subject to Denial of Service attacks.

For these reasons, centralized shared databases are then replaced with **distributed databases**, represented in Figure 2.5, as they are more robust in event of network failures.

They main negative aspect is that they operate over synchronous and asynchronous networks, hence packet delivery may experience *congestion*, failure, or message may wait in queue. Secondly, this system may overlook *consistency* using weak enforcement of properties over scalability normally in case of replication systems. In addition, a byzantine attack may occur where all nodes could not be working as intended and it causes the achievement of an incorrect consensus. Also, in situations with highly scalable distributed databases, multi document transactions in parallel are not allowed becasue of efficiency issues.

In the blockchain environment, the nodes continuously check other nodes integrity using a consensus protocol to agree on a common state of the chain. The chains are cryptographically auditable indeed they rely on Merkle root value and order-execute architecture. The blockchain network orders the transactions first using a consensus protocol and then sequentially executes them in the listed order in all peer nodes.

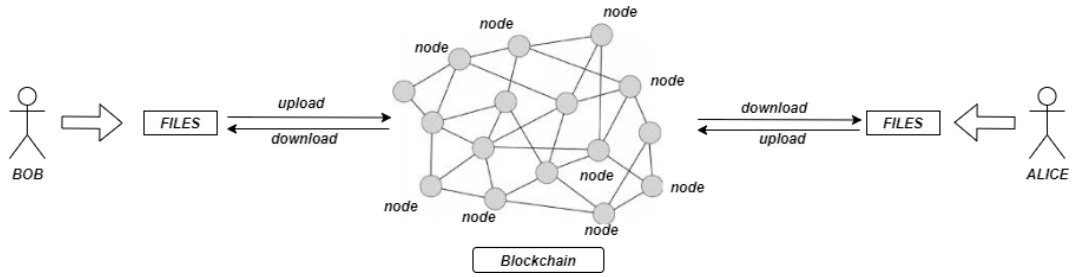

Figure 2.5: Blockchain Architecture.

As shown in Figure 2.5, `Alice` and `Bob` are the users involved in the transaction. Once the document is updated, it is added by computing its hash value which could be digitally signed using users' private/public key pairs and added to the chain.

The validation of the transactions is done by miners which add a block to a chain. The hashing process of the data blocks is responsible for the logical chaining. This mechanism is based on the fact that any block B_i stores the hash of its previous block B_{i-1}.

The hash in any i-th block is computed as $H_i = f(input_i, ID_i, \textit{Timestamp}, H_{i-1})$ where $input_i$ is the input document, ID_i is the digital identifier associated with the document, *Timestamp* is the current timestamp value and H_i and H_{i-1} are the hashes of current and previous blocks respectively.

All the blocks link to create a trace back to the genesis block, allowing consensus in a blockchain network, as Figure 2.6 illustrates. All hashes are computed and used in order to form the hash at the next higher level of the chain. This is the concept of the **Merkle tree**, a type of binary tree, that appeared for the first time in 1979 and took its name from the computer scientist Ralph Merkle who wrote a paper titled "*A Certified Digital Signature*"[28] where he explained a new method of creating proof.

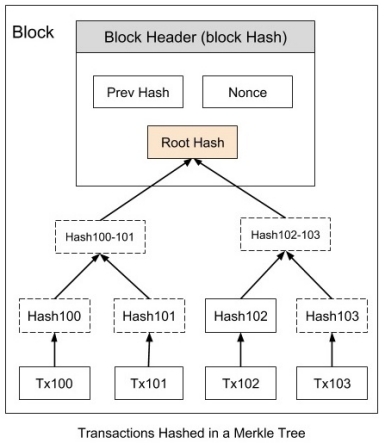

Figure 2.6: Merkle Tree Structure.

The Merkle tree is a *multi-level data structure* that allows a large amount of information to be verified for accuracy in an efficient and quick way. It is based on the concept that it is sufficient to present only a small number of nodes in a Merkle tree to give proof of the validity of a branch. A Merkle tree is composed of a*set of nodes* with a large number of leaf nodes at the bottom of the tree containing the underlying data, a set of intermediate nodes where each node is the hash of its two children, and finally a single root node, formed from the hash of its two children, that id the root of the tree. If a malicious user tries to swap in a fake transaction into the bottom of a Merkle tree, this change will produce a change in all nodes above it, from the previous one to the root of the tree. Moreover the hash of the block will be changed and the protocol to register it as a completely different block.

To add a block, a miner must solve a **cryptographic puzzle** by discovering starting bytes of the block in such a way that the hash of the block is smaller than the acceptable

target hash value. Each block acts a puzzle for a miner which is termed as nonce or difficulty value. Once the nonce is solved by a miner, the block gets appended to the existing validated chain by appending the hash value of the chain to the block and this represents the **Proof-of-Work** in a blockchain network. In order to compensate miners for this computational effort, the winner miner of each block has a reward. In addition, whether any transaction has a higher denomination in its inputs tha in its outputs, the difference goes to the miner as a transaction fee. The copies of the new block are added to all nodes in the network in order to maintain consensus.

Sometimes, it may happen that miners can generate different blocks of the same transactions and a *fork* of the chain. Thus, forking creates a problem, so as one branch becomes longer, all the miners prefer adding blocks to that branch. This rule of longest chaining increases security in the system indeed the creation of a longer chain by the attacker requires more computational power and resources than the rest of the network in a very short span of time (hence, 51% attack).

The only possibility is that the blocks are added in a slow manner or there are many fork operations in a chain, which makes it more time consuming to elect the longest chain. The attacker can create his chain to be the longest and force the miners to add blocks to its chain making it look like a legitimate chain. However, the propagation latency of adding blocks in a network is small, hence the probability of such an attack is really low and for this reason the blockchain network is very secure in a distributed environment to achieve common consensus. Hence, the blockchain network solves the current limitations of the database system and achieves consistency in a shared and distributed platform.

2.1.1 Blockchain 1.0

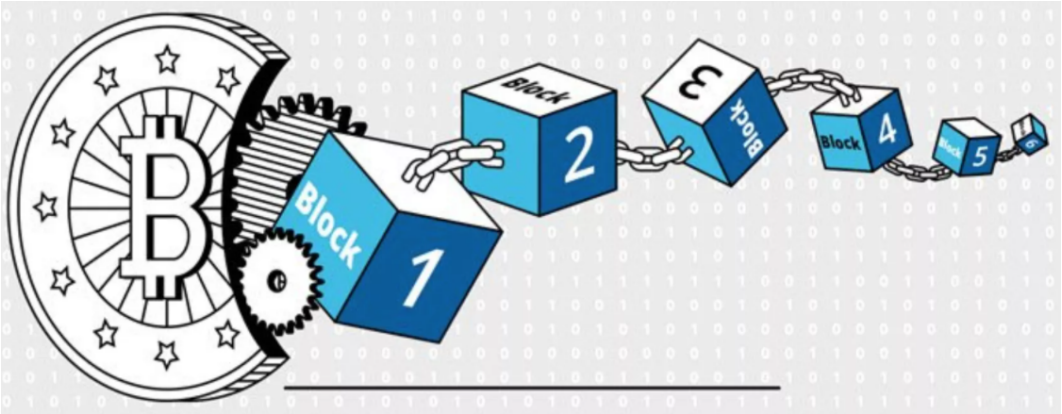

Figure 2.7: Bitcoin's Ledger.

The first generation technologies have to improve the existing monetary system and clients can send transactions relying on cryptography. It is clear that the digital currency becomes a medium to buy and sell goods and services over the anonymous network.

Cryptocurrency are digital cyrrency that entails the use of cryptography in order to achieve a secure transaction. It can be used as a secure way of exchange, indeed it uses high cryptography to ensure verifiability of asset transfer. Each coin defines an electronic signature where the private key is used for signing the transaction and the public key is used for verification.

The first blockchain, Bitcoin, was conceptualized by Satoshi Nakamoto in 2008. The Bitcoin ledger is a Finite State Transition Automaton with states as ownerships of all existing bitcoin users and transactions that are transitions between the states.

It is a decentralized anonymous system that does not need a third trusted party to validate the transactions but the same users can control their fund and do transactions. The output of any state is a transactional value if the transaction is successful so there are enough bitcoins (BTC), or error, otherwise. The state transition function allows us to reach a new state. As a ledger system, every peer of the network keeps a copy of this finite state transition system. The PoW is carried out using the hashing scheme Bitcoin based on HashCash and SHA-256 hash function.

There are a number of benefits and drawbacks in relation to the use of Bitcoin Blockchain. Firstly, the transactional costs are lower than the ones associated with other electronic payment channels. Secondly, transactions are anonymous and they provide a secure and transparent mechanism, so counterfeiting is not possible. Despite these positive aspects, there are also some negative effects. First of all, the approval of transactions are slow in comparison to other electronic channels. Another point to consider is that they have launched several fraud schemas over Bitcoin wallets like Ponzi Schemes or Bitcoin Mining Scams. It represents one of the first generation technologies that is based on the Proof-of-Work algorithm and its main aim is recording transactions.

2.1.2 Blockchain 2.0

Figure 2.8: Traditional contract compares to smart one.

In 2015 the **Ethereum** blockchain-based platform was launched and the concept of blockchain evolves into a second generation category driven by the requirement of general purpose application. This platform is not a simple cryptocurrency but it is a digital environment that can be used for decentralized projects.

The most remarkable innovation of Ethereum is the *enablement of smart contracts* that provide security to transactions verifying the execution of all conditions by all the peers and processing the transaction as the result of this execution.

In addition, smart contracts are precise and store all terms and conditions which are fully visible to all transactional involved node peers.

The Blockchain 2.0 are very popular as they can be extensively applied in many areas, still some serious challenges and limitations that make usage of smart contracts vulnerable.

2.1.2.1 Transactions

A public blockchain is a globally distributed database. Each node of the blockchain network has a copy of the digital ledger, it is enabled to read entries in the database and it checks the validity of each transaction. In order to to change something, the node has to make a **transaction** which has to be accepted by all others. If a majority of nodes say that a transaction is valid then it is written into a block.

If the transaction successfully terminates, the remaining gas is returned to the caller, otherwise all the gas allocates for the transaction is lost. If a computation consumes all the allocated gas, it terminates with an **out-of-gas exception** and the caller loses all the gas. As the blockchain is a distributed network, the change the node wants to make will be eventually broadcast throughout the network.

Once the transaction is applied to the database, no other transaction can alter it. Furthermore, a transaction is always cryptographically signed by the creator and this mechanism enables to avoid specific modifications of the database.

2.1.2.2 Consensus Mechanism

Blockchains by definition are decentralized systems and do not need a third party trusted authority. In order to guarantee the *reliability* and *consistency* of the data and transactions, they used a **decentralized consensus mechanism** that is due to a transformation of the Bizantine General Problem.

According to this scenario, there is a group of generals who command a portion of Byzantine army circle the city. Some of those prefer to attack whereas others want to retreat. The attack would fail if only part of the generals attack the city. They have to reach an agreement to decide their future action. The achievement of a consensus in distributed environment is challenging both in Bizantine scenario and in blockchain network. Indeed, in the latter, there is no central node that ensures ledgers on distributed nodes are all the same.

In the existing blockchain systems, the most common approaches to reach a consensus are:

- ***PoW***, Proof of Work
- ***PoS***, Proof of Stake
- ***PBFT***, Practical Byzantine Fault Tolerance
- ***DPoS***, Delegated Proof of Stake
- ***PoB***, Proof of Bandwith
- ***DPoS***,Proof of Authority

Ethereum as one of the most popular blockchain systems uses the *PoW* consensus algorithm and it incorporates the *PoA*, whereas other cryptocurrencies also use the PoS mechanism (i.e. PeerCoin or ShadowChash).

As far as **PoW** mechanism is concerned, the idea was conceptualized in 1993 by Cynthia Dwork and Moni Naor[17] but the term *Proof of Work* was coined in 1999 by Markus Jakobsson[22]. Bitcoin uses this algorithm to achieve consensus in its P2P network. PoW protocol has the main goal of deterring cyber-attacks such as a distributed

denial-of-service attack (DDoS) which has the purpose of exhausting the resources of a computer system by sending multiple fake requests.

Typically, in Bitcoin, each block constains `PrevHash`, `nonce` and `T`. `PrevHash` is the hash value of the last generated block, and the `T`s are the transactions of this block which are stored in a memory called *mempool*.

The procedure of *mining* allows the verification of a transaction and, once it obtains a successful result, the transaction will be added in the next block. Miners' task is to get to know the cryptographic hash value of the recorded block because it must be referenced for creating the next block. All the network miners compete to be the first to solve the value of the `nonce` by solving the Pow puzzle that is a mathematical problem. In particular, a correct `nonce` should satisfy as it is shown in the Equation that the hash value is less than a target value.

$$SHA256(PrevHash||Tx1||Tx2||...||nonce) < Target \tag{2.1.1}$$

This solution cannot be solved in other ways than through brute force so that essentially requires a huge number of attempts. The miner who finds the hash of the last recorded block and solves the mathematical puzzle, receives the reward and broadcasts it to the whole network. Then, a new block is created that has all the transactions in the mempool post verification.

Looking in detail, the miner has to solve a asymmetric cryptographic puzzle. The computational difficulty of the puzzle will gradually rise if the solution of the puzzle is easily find. This grow means that the miner nodes have to use ever-increasing computing power so other kind of consensus algorithm are improved in order to resuce this drawback.

Proof of stake mechanism uses the proof of ownership of cryptocurrency to prove the credibility of the data. In PoS-based blockchain, instead of miners, *validators* decide to stake their cryptocurrencies for the transaction verification. The larger the amount of stake and the longer the duration of the stake, the better are the chances of the staker to get transaction validation responsibility. All cryptocurrencies in this network are already created, and there is not the mining process. This change eliminates the need to solve a complex cryptographic puzzle and its resulting computational cost.

In this scenario, the *forging* is the transaction validation process where it is not necessary the involvement of the whole network. So, this mechanism improves scalability. PoS allows the *sharding* which is the implementation of other technology solution and it enables the store of horizontal portions of the network in different groups of nodes. Each node has its own perception of the network so sharding can not be implemented in conjunction with POW algorithm, and PoS is needed with separate stakers for separate shards.

2.1.2.3 Blocks

In order to overcome *double-spend attack* which happens if two transactions in the network want to empty an account. Only the first accepted transactions can be valid but in a peer-to-peer environment, the concept of *first* is not an objective term. The countermeasure of this kind of issue is the creation of a globally accepted order of the transactions. Indeed, the transactions are stored in a **block** and then they are executed and distributed among all nodes of the network. If two transactions contradict each other, the one that ends up being second will be rejected and not become part of the block.

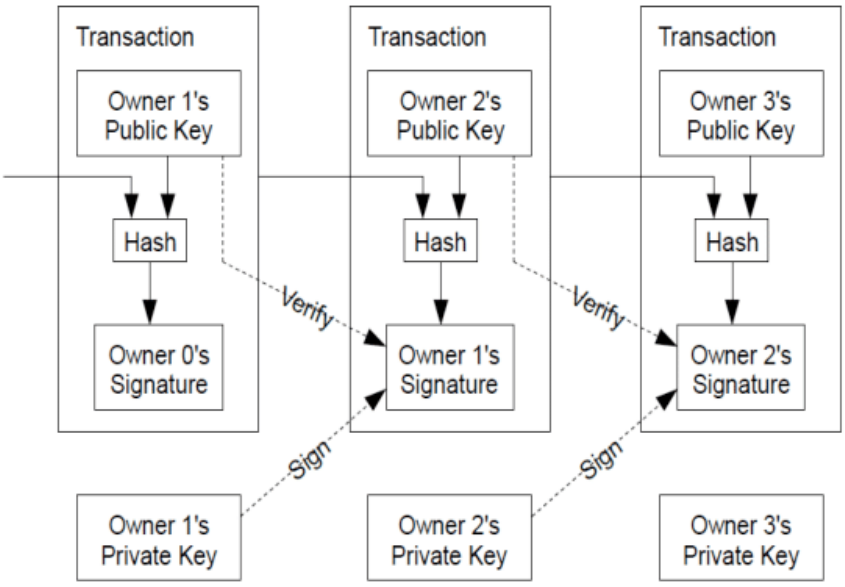

Figure 2.9: Block structure in blockchain system.

As far as the blocks are concerned, they form a linear sequence in time. Blocks are added to the chain in rather regular intervals[1]. As part of the mining process, blocks may be reverted from time to time, but only at the *tip* of the chain. The more blocks are added on top of a particular block, the less likely this block will be reverted.

In the blockchain, each full node saves the information of all blocks. The block propagation mechainisms are the basis of the consensus and trust of this network and they can be categorized in the following way:

- *Advertisement-based propagation*: this is a Bitcoin scenario. When a node `A` obtains information of a block,`A` will send an `inv` message to the connected peers. Once the node `B` gets `A`'s `inv` message, it will reproduce the message to its linked peers. If the node `B` already has information of this block, it will do nothing, otherwise, it will reply to node `A`. When a node `A` receives the reply message from node `B`, node `A` will send the information to `B`.

- *Sendhaeders propagation*: this mechanism is an improvement of the advertisement-base propagation. Node `B` will send a `sendheaders` message to `A`. Once `A` receives the information of the block, it will forward the block header to `B`. In this situation there is a speedup of the mechanism, indeed, node `A` does not transmit the `inv` messages.

- *Unsolicited push propagation*: once the block is mined, the miner will broadcast the block to the other peers. This transmission does not require `inv` and `sendheaders` messages so it increases the speed of the block propagation.

- *Relay network propagation*: in this situation, all the miners share a transaction pool. Each transaction has its own global ID, that will reduce the broadcasted block size, the network load and improve the propagation speed.

[1] In Ethereum environment, the blockchain is updated every 17 seconds.

- *Push/Advertisement hybrid propagation*: this scenario is used in Ethereum. In our generic scenario `A` has n linked peers. `A` will push the block to $\sqrt{n}$ nodes in a direct way. The other $n - \sqrt{n}$ connected peers, node `A` will advertise the block hash to them.

2.1.3 Blockchain 3.0

The ever-growing climb of smart contracts produces a large flow of micro-transactions. Even if Ethereum improved the transaction rate to 15 tps (over Bitcoin 7 tps), it is not sufficient to support today's economic trends.

Hence, Blockchain moves to decentralized internet, which will combine data storage, communication Networks, Smart Contracts and Open standards platforms and uses **DApp - Decentralized Applications**. Thus, an ultimate blockchain application

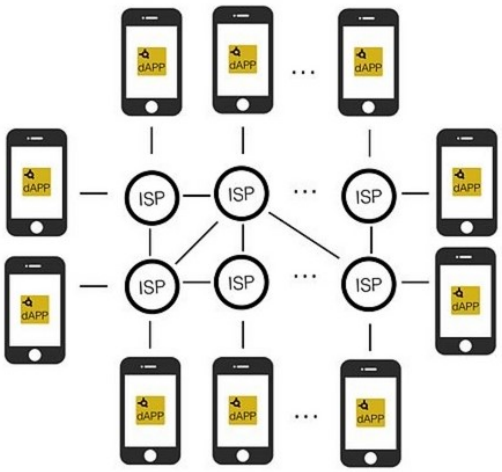

Figure 2.10: Dapps Architecture.

should be DApp hosted on a peer network as it does not require maintenance, governance ore human intervention.

As a consequence, there is the formation of Decentralized Autonomous Organizations (DAO) in which profit is shared by all members by simply recording their activities on the Chain.

Its first positive effect is that no single node controls the transaction which could have become a single point of failure. Moreover, the transactional speed is increased about 100 times in distributed environment systems.

On the other hand, one of the drawbacks is taking into account updates and bug fixes that are difficult because all copies in the network need to be updated.

Another point to consider concerns the achievement of consensus, indeed complex protocols need to be implemented to get data validation, thus restricting the network scalability.

2.1.4 Blockchain 4.0

The use of decentralized applications suddenly rises and a platform that can integrate several services and architectures by allowing cross-chain communications becomes essential.

Figure 2.11: Blockchain related to Industry.

The use of a scalable network which increases degree of trust and privacy is fundamental and for this reason the Blockchain evolved to a new level known as **Blockchian 4.0**. It allows users from different platforms to work together as an own unit, thus making an integration towards business needs and demands of Industry 4.0.. Industry 4.0 requires an enterprise resource planning platform which can provide automation and integration of different execution platforms as a coherent unit. Blockchain 4.0 enables IT systems to do business integration, operates on Cross-Blockchain business processes like allowing for autonomously placing an order via smart contract as well as ensuring safety to machines.

In conclusion, "*the use of Blockchain is not only limited to provide trust and privacy as a means of crypto currency, but be a business provider in current industry and market demands. Blockchain is the biggest disruptive technology that has hit the markets and most industries are gradually shifting to blockchain platforms.*"

2.2 Smart contracts

The term *smart contract* appeared over twenty years ago and it was strongly related to blockchain technology.

The idea of them has been offered by the cryptographer Nick Szabo. This innovation has impacted on technology from IoT to AI and the application of blockchain has fastly diffused.

2.2.1 History

The concept of **contract** has always been present in human social relationships. Since the dawn of time, the need for respecting the rules of in the various period of time has given birth to several kind of contracts.

In primitive societies, these rules consisted in the basic regulation of barter and, then they evolved in ancient legal codes of written rules. Indeed, in the *Code of Hammurabi*[2] which regulated the Babylonian society, there were several references to the correct behaviour of a dweller in exchange circumstances (rules 100-198). The first significant evidence of contracts in Wenstern societies can be found in the Greek civilization.

The Greeks were navigators and merchants and they used written rules which could

[2] 1750 BC

look like our basic contracts. The philosopher **Plato**[3] dealt with this topic in the eleventh book of ***The Laws***[4] where he highlighted that contracts were a well-understood and familiar part of contemporary civic life in Athens. He warned about what might happen if a man (probably only men could undertake contracts then) breaks a contract, and who should arbitrate between the disputing parties. He also suggested that not all contracts were valid — for instance, when someone engaged in illigal conduct, when one on of the parties was under pressure, or when the breach of the contract was due to a third party not involved in the contract.

In *ancient Roman law*, a contract did not necessarily imply an agreement - the **conventio** - but it constituted the contractual obligation which bonded two entities - the **negotium contractum** or **contractus negotii**. The contract gained several solemn forms and most of them were oral. There were two different types of formal agreements; the former were verbal like the sponsio, the stipulatio, the dotis dictio or the promissio iurata liberti and the latter were written like the nomen transcripticium, the singrafe or the chirografo.

By contrast, since the *Justinian Roman law* (and also in the modern law), the contract has been defined as an agreement between two or more parties and it has represented a legal relationship recognized by the law.

Over the centuries, it is evident that the *medieval society* was based on the personal relationship of feudal subordination between the vassal and his lord. The vassal's obligation to the lord was an exchange of complete allegiance whereas the lord offered to protect the vassal from external forces. It was an actual bilateral contract which could be broken by the default by one of the two contracting parties. This kind of subordination has persisted throughout the centuries and it caused wars that aimed to the freedom of princes.

As far as the fundamental function of the contract law is concerned, it is necessary to mention Hobbes' idea[5] outlined in the **Leviathan**.

According to this, the contract has to deter people from behaving opportunistically toward their contracting parties and "*if a covenant be made wherein neither of the parties perform presently, but trust one another, in the condition of mere nature ... upon any reasonable suspicion, it is void: but if there be a common power set over them both, with right and force sufficient to compel performance, it is not void.*"

[3] *Plato* (ca. 427-347 B.C.E.) was an Athenian philosopher who is recognized among the most important philosophers of the Western world. Plato can be plausibly credited with the invention of philosophy as we understand it today – the rational, rigorous, and systematic study of fundamental questions concerning ethics, politics, psychology, theology, epistemology, and metaphysics. He wrote primarily in dialogue form.[3]

[4] The Laws - Νόμοι- is Plato's last, longest work written in the 4th century BC. The book is a conversation on political philosophy between three elderly men. These men work to create a constitution for Magnesia, a new Cretan colony. The government of Magnesia is a mixture of democratic and authoritarian principles that aim at making all of its citizens happy and virtuous.

[5] *Thomas Hobbes* (1588-1679) is an English philosopher known for his political thought. His vision of the world is strikingly original and still relevant to contemporary politics. His main concern is the problem of social and political order: how human beings can live together in peace and avoid the danger and fear of civil conflict. His political philosophy is articulated in the masterpiece ***Leviathan or The Matter, Forme and Power of a Commonwealth Ecclesiasticall and Civil***[2]. It is written in 1588–1679 and published in 1651. It argued that civil peace and social unity could be achieved by the establishment of a commonwealth through social contract. Hobbes's ideal commonwealth is ruled by a sovereign power responsible for protecting the security of the commonwealth and granted absolute authority to ensure the common defense. In his introduction, Hobbes describes this commonwealth as an "artificial person" and as a body politic that mimics the human body. .

2.2.2 Origin

The concept of *traditional contract* implies the presence of a trusted third party that manages and validates the transaction from one to another. The role of this third party is essential and it defines a generic centralized scenario.

From the end of XX century, this contract needed to be smarter, and therefore the idea of smart contracts spread out. Smart contract technology are intended to replace the traditional contracts without loosing important properties like safety, efficiency, and reduce possible issues.

The naming *smart contracts* appeared in 1996 on the Extropy magazine, the cryptographer Nick Szabo wrote an article ***Smart Contracts: Building Blocks for Digital Markets*** where he gave his definition of this revolutionary and innovative concept. Szabo explained that a generic contract is a "*set of promises agreed to in a meeting of the minds [which] is the traditional way to formalize a relationship*".

In addition, he defined a smart contract as a " *set of promises, specified in digital form, including protocols within which the parties perform on the other promises*" without the use of artificial intelligence.

In 2016, the lawyer Josh Stark[34] gives an overview of two different ways that the term **smart contract** is commonly used:

- the former is named ***smart contract code***, is operational and involves software agents but not necessary on a shared ledger. In this scenario, the *contract* specifies that software agents are performing obligations and practicing specific rules. The agents may take control of some assets within a shared ledger. This concept of contract does not need consensus, indeed each definition is different in suble ways.

- the latter is known as ***smart legal contract*** and it is focused on the legal point of view. It analyzes how legal contracts can be expressed and implemented. It contains operational aspects, issues relating to how legal contracts are written and how the legal aspects should be interpreted.

It is clear that the term *smart contract* should cover both versions, so it is possible to give this definition: *a smart contract is an automatable and enforceable agreement. Automateble by computer, although it requires human input and control. Enforceable by legal enforcement of rights and obligations.* Indeed, this explanation contains smart contracts as legal agreements and their automated software [6].

2.2.3 Explanation

The smart contract is a legal contract between mutually distrusting participants without interference of any third party in the form of programming code and it is stored in the blockchain by a *contract-creation* transaction.

As it is clear from the Figure 2.12, all the transactions of the smart contract are processed by the blockchain when the conditions in the agreement are matched. Data in the blockchain are guaranteed to be valid in line with certain predefined rules of the system, they are public and everyone can access a copy of them. The agreement is enforced by the consensus mechanism of the blockchain. Each smart contract account holds an *amount of virtual coins* and an own *state* and *storage*.

[6] It may not necessarily be linked to a formal legal agreement.

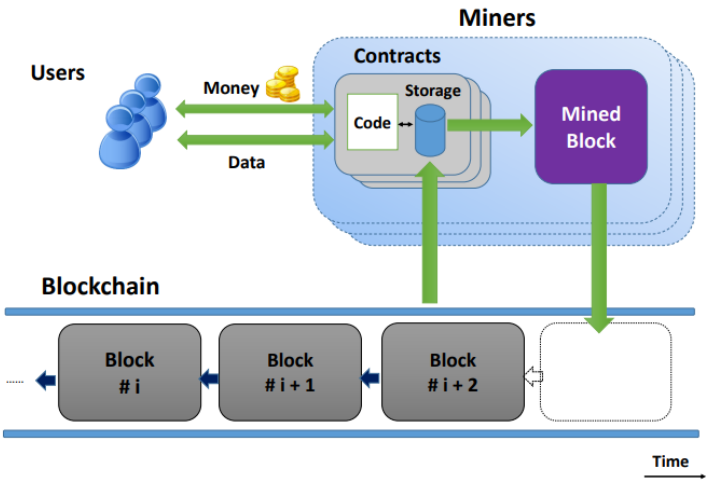

Figure 2.12: Smart Contract System.

2.2.3.1 Benefits of Smart Contracts

The introduction of smart contracts has several positive aspects.

Accuracy is one of the advantages that the implementation of smart contracts enhance. In the process of setting up a smart contract, all the information concerning the contract is expressed in a conditional format in if-then statements. The expression of all terms and conditions in a contract must be explicitly and accurate. This is a critical requirement because transaction errors may emanate from any omission. The automation in the definition of smart contracts would avoid the majority of the problems that are detected in the traditional contracts.

Another positive effect is the **clear communication and transparency**. Once the agreement is established, changes cannot be easily implemented. Each transaction is monitored and controlled by the other network peers in the blockchain. In this scenario, trasparency is provided and issues are decreased. Moreover, unlike the traditional contract where the organization would have to use the legal framework as third party, in the network are needed other nodes which ensured that each of the transaction pertaining to the contract is accurate and correct. In addition, smart contracts implementation is managed by other nodes in the blockchain network. So, once the contract is triggered, the contract self-executes. This procedure is achieves through the use of trigger events during the scripting of the contract. A trigger event may be, for example, a date, time, or an activity started by a party of the contract, such as the transfer of an amount of cryptocurrency from the customer's wallet to that of the receiver.

This means that smart contracts are more **efficient** than traditional ones and they do not require human verification, as a matter of facts all the checks are done by the nodes in the network. Each contract is a separate entity and each transaction is always validated. This mechanism produces a fast, resilent and robust way of contract execution. According to the research carried on by Marino and Juel[27] in 2016, smart contracts used one of the highest security measures. The peers of the network are non-trusting and this ensure that each trnasaction is carried out in the best way.

In addition, all the nodes have a **uniform view** of the status of the transactions. Moreover, the whole technology is implemented through cryptography techniques which provide an high encryption of data and enable the use of private and public keys. The

security of the smart contracts is enhanced by the validation mechanism, indeed, before any node commits a transaction, the transaction must first be validated by all the peers across the blockchain network. Another advantage that is strictly related with the security concept is the cost reduction. The implementation of smart contracts through a distributed ledger technology removes the rule of a middleman and cuts the overall organizational costs.

2.2.3.2 Limitation of Smart Contracts

Apart from several positive aspects involved in the implementation of smart contracts, they have some limitations and disadvantages which limit their application in certain scenario.

The main drawback is the **difficult modification** after the setting up phase. In traditional agreements, change of terms and conditions are often used and they keep changing throughout the contracts life. Furthermore, the conventional contracts allow their annulment, embedment, and modification as explained in the research provided by Huckle et al [21].

By contrast, in smart scenario, the **alteration** of contracts terms may cause practical problems, compromise security apparatus, and require further strengthening of the transaction controls. These countermeasures may prevent to initiate invalid transaction or transactions that are aimed at unauthorised manipulation of records. Due to the complexity of the smart contract and blockchain technology, it may be necessary to ensure that the correct permission is granted to the right node, and that all the nodes can monitor the amendment of the contract.

Moreover,blockchain technology involves the sharing of smart contracts across all the node of the network and all transactions are stored on a distributed ledger using encoded permissions in each of the nodes. This technology provides **anonymity**, indeed all the participiants are anonymous and secured.

However, there is not any security guarantee during the contract excution, because the nodes are anonymous in their operations, the ledger is public, and the transactions are visible. Buterin supports that, despite anonymity of the nodes, the maintenance of a public ledger in the distributed environment causes a loss of privacy[11].

Besides, smart contracts need to develop a protocol that can ensure the validation of transactions without reading the contents of the transaction. As a matter of fact, transaction may be anonymous whereas the contents are not and can be read and accessed by each peer of the network. Kim and Laskowski show that the main characteristics of a legally enforceable contract are: "*the acceptance by the other party or parties, a promise, consideration, and legal capacity mutuality [25]*". All these elements are not applicable to smart agreements.

2.2.3.3 Technical Explanation

The initial step to develop smart contracts is its compilation and it is done with the help of Solidity compiler, **solc**, that creates two main objects:

- **A**pplication **B**inary **I**nterface - ABI - definition,
- Contract bytecode.

The former allows to invoke functions the contract and is composed by declarations of external and public funnctions with parameters and return types. The ABI creates a new *instance* for the contract that is used when any caller calls the contract function. The latter is a **low-level rapresentation** of the smart contracts, indeed it is an assemply language made up of multiple opcodes. Each opcode perform a specific action on the Ethereum blockchain.

Once the ABI definition generates a new instance for the contract that generates a new transaction, it is possibloe to go on with the mining phase. After this, the contract is accesible at an address firmed by Ethereum. when the contract is deployed in Ethereum Virtual Machine, the contract and its functions can be invoked byt the use of the created address.

2.3 Ethereum

Figure 2.13: Ethereum Network.

Ethereum is a decentralized virtual machine, which runs ***smart contracts*** upon request of users. Contracts are sets of functions and they are written in EVM bytecode.

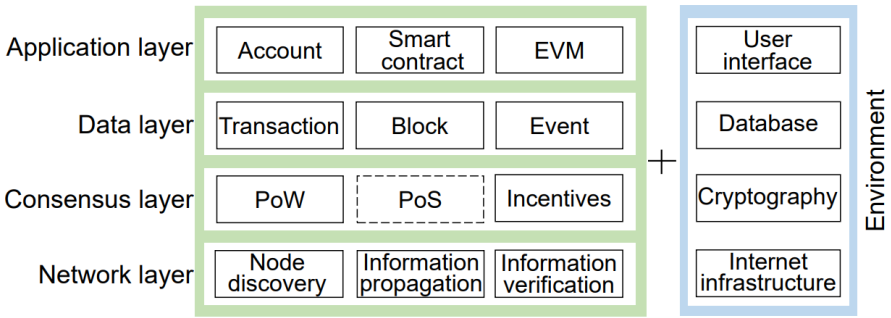

Figure 2.14: Architecture of the Ethereum blockchain.

As Figure 2.14 shows, Ethereum blockchain has a 4-layer architecture and it operates across them.

First of all, there is the **application layer** where clients execute smart contracts which are, usually, written in Solidity, the most mature high-level smart contract language and are associated with Ethereum accounts.

Ethereum supports two kind of accounts that Figure 2.15 takes into consideration:

- externally owned accounts, also known as **EOA**;
- contract accounts.

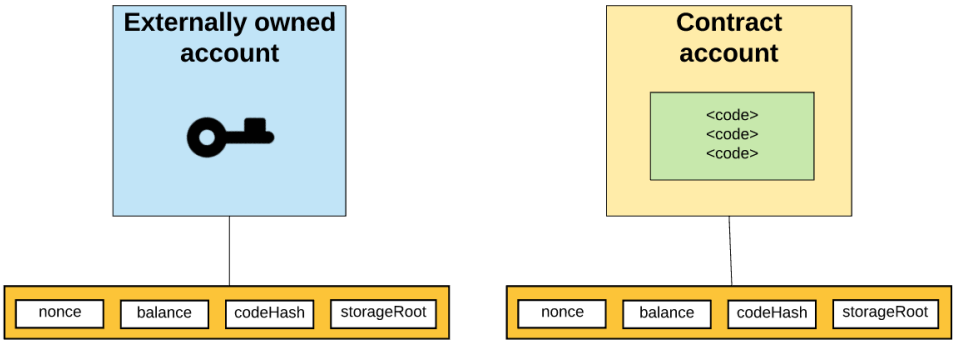

Figure 2.15: Ethereum Accounts.

On the one hand, the former stores the user's funds and it is associated with a public key. The EOA represent participants, including callers (who call functions of smart contracts), deployers (who deploy smart contracts on Ethereum), and miners (whose nodes work to do contribution to the ledger).

On the other hand, the latter is a type of programs, a piece of executable bytecode which defines the logic of interest. Both the accounts has a dynamic state with a common structure that is defined by:

- the `nonce` tracks the number of transactions that are initiated of the owner of the EOA or it represents the number of contracts created by the contract account;
- the `balance` shows the amount of owned Wei (i.e. 10-18 Ether);
- the `storageRoot` is the hash of the data structure trie that contains variables associated to a piece of bytecode;
- the `codeHash` represents the hash value of the contract account's blockchain.

Second, the **data layer** defines the data structure of the blockchain. The execution of the transaction allows to update the states of the accounts involved and also the state of the whole blockchain.

A sender constructs a transaction, digitally signs it and submits it to a client. A transaction is a single cryptographically-signed instruction and they can be classified from two dimensions. According to the *data in the transaction* first category, transactions can be classified into:

- **Ether transfers** involves the exchange of Ether from one user account to another;

- **contract executions** represents that an account calls a function of a smart contract with some data as the input and some Ether as reward for executing the contract.

By contrast, the second group is focused on the *transaction initiator* and it illustrates that a transaction can be divided into:

- **external transactions** that are intiated by user accounts
- **internal transactions** which are mad by smart contract accounts.

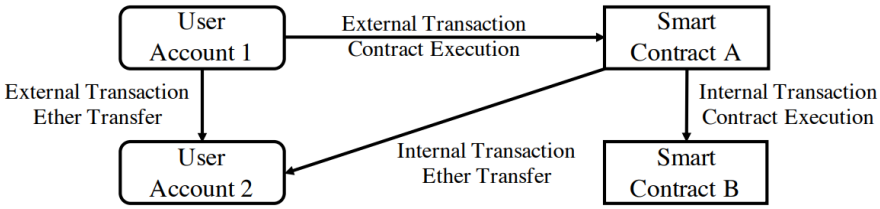

Figure 2.16: Relation between the two classification of actions.

Figure 2.16 provides information about the relationship between the two categories. If the target account of a transaction is a user account, the transaction belong to Ether transfer, whereas if the target account is a contract account, the transaction belong to contract execution.

Transaction	**Setting**
`money-transfer` transaction	The specific val*value* is transferred from the sender's EOA to the recipient's EOA contract account
`contract-creation` transaction	The *input* is a piece of bytecode, a new contract account is created and is associated with the bytecode
`contract-invocation` transaction	The recipient is a callee contract and *input* uniquely identifies the callee function, the bytecode associated to the callee contract account is loaded into the EVM.

Table 2.1: Setting for each kind of transaction.

In general, a transaction contains common field:

- **nonce**: this value is the number of transactions sent by the sender;
- **gasPrice**: it represents the number of Wei to be paid per unit of gas and it is the cost of the execution of the transaction;
- **gasLimit**: it is the upper bound of gas used in executing the transaction. This is paid up-front, before the beginning of the computation and it cannot be modified;
- **to**: it is a 160-bit field that stores the message's call recipient;
- **value**: the scalar constitues the number of Wei transferred to the message call's receiver or it is a field for the newly created account;
- **v, r, s, $\mathbf{T}_w$, $\mathbf{T}_r$, $\mathbf{T}_s$**: it is the signature of the transaction that establishes the sender.

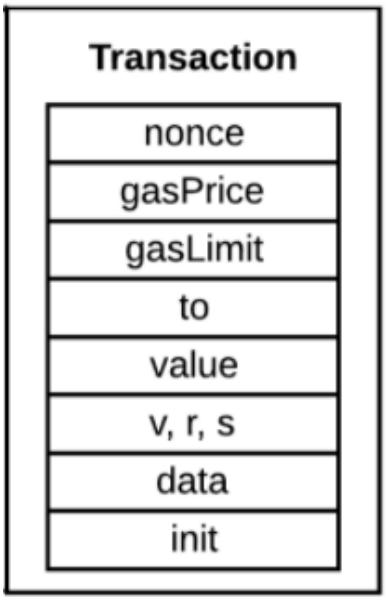

Figure 2.17: Transaction Structure.

In addition, the contract creation transaction holds the **init** field, an unlimited size byte array that specifies the EVM-code for the account initialization mechanism.

By contrast, the message call transaction contains the **data** value, an input data of message call. Following this,the client validates the received transaction and broadcasts it to the network. Once the miner clients have received the transaction, they will update their pool. Subsequently, the miner executes a set of transactions and creates a new block and then the whole state of the blockchain is updated.

Looking in detail, a block consists of the *block header* an the *block body*, as shown in the Figure 2.18. In particular, the block header is unique, and each such block is identified by its block header hash individually. The header of the block stores several important data such as:

- **parentHash**: it is a 32-byte field that contains the hash of the previous block header. It contains a pointer to the previous block and it constitutes an important field for the blockchain, indeed its change would affect the previous block and consequently the entire blockchain;
- **stateRoot**: it is the root node of the whole state;
- **transactionsroot**: it shows the the root node of transaction hash;
- **receiptsRoot**: it represents the root node of the receipt hash;
- **difficulty**: it is a 4-byte value that illustrates how difficult is to get the hash;

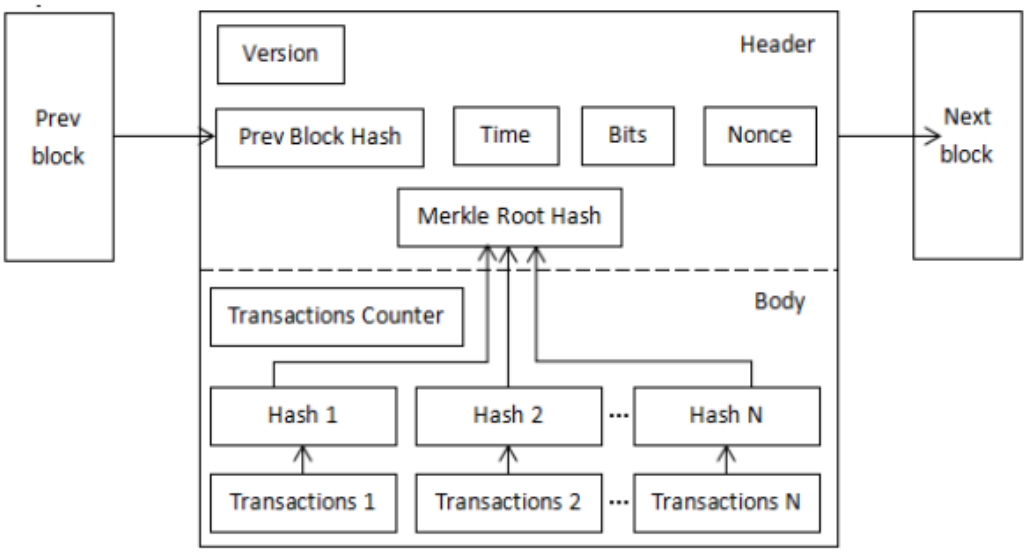

Figure 2.18: Block Header Structure.

- **timestamp**: it is a 4-byte field that give information about the time when mining ends[7];
- **number**: it is the counting value of the block which is sequentially incremented (0 is a genesis block.);
- **gasLimit**: this value represents the maximum amount of gas allowed in a block. It defines how many transactions can be kept on the sum of gas;
- **gasUsed**: this field contains the amount of all the gas used by all transaction of the block;
- **nonce**: it is 4-byte file that is a one-time code randomly chosen to prevent replay attacks. This is a Proof-of-work nonce that is used as a meaningless number when mining;
- **extraData**: this is an optional and free field to deposit extra data.

The *block body* is composed of a transaction counter and transactions. The maximum amount of transactions that a block can contain depends on the size of the block and of each single transaction.

The miner has to solve a **PoW** by finding a random nonce such that the hash value of the block is smaller than a threshold value. This puzzle solving method prevents cheating from several perspectives, such as leveraging powerful computing resources, forming colluding partners, altering the proof of the correct puzzle answer. Moreover, miners have to find the correct hash value to show the proof of work, but each node on the network can easily confirm that the hash value is correct. By combining the account nonce with the proof of work `nonce`, Ethereum can speed up the time required

[7]The timestamp begins when the miner started the hashing the header. This value must be strictly greater than the median time of the previous 11 blocks. Full nodes will not be accepting blocks with headers that are more than 2 hours in the future according to their clock

to mine a block significantly compared to Bitcoin, without substantially weakening the resilience of blockchain against malicious manipulation. Upon creating a block, the miner broadcasts to the network this value and,once the block is validated, it is added to the blockchain.

The next layer is the **consensus** one that assures a consistent state of the blockchain. Ethereum **proof of work** is designed in a similar way as that of Bitcoin. A new block can be accepted by the network after being validated through *mining*. Miners may mine any unverified blocks on the network by solving a puzzle and complete with one another until a winner gets the successful results.

If a miner first finds a hash that matches the current target, it broadcasts the block across the network to each node. Once the block passed the verification, each node adds this block to their own copy of the ledger.

If another miner finds the hash faster, the rest of the miners will stop working on the current block and start the mining process for the next block.

Mining mechanism is simultaneously repeated by multiple miners. When two miners mine the next block at the same time, the network will decide which one will ben the main chain. When two blocks `X` and `Y` are mined at the same time. Miners would accept the first block that was broadcasted to them. Thus, some miners accept `X` and others `Y`. The block that is accepted by the major part of the network (52% or more) will be the winner. Miners who accepted block `X` will continue to mine the next block on the top of the block`X`, whereas those accepted block `Y` will continue to mine he next block on the top of `Y`.

If the next block is found and added on the top of block `X` faster, then the miners working on top of block `Y` will turn to the `X` chain, which is the main chain. The block `X` will be the winner and the block `Y` will become an orphaned block.

Ethereum uses a variant of the **GHOST consensus protocol** which ensures that any attempt to tamper with the transactions and selects the heaviest branch as the main chain that has the highest block difficulty. This blockchain rewards the regular blocks (with 2 ETH) and also the uncle blocks (that receives 1/32 of the reward given to the regular blocks). Like the PoW in Bitcoin, a system-defined timer controls the hardness of the hash puzzle to ensure that a block can be validated in approximately every 12-15 seconds.

However, in order to overcome the negative aspects of the Proof-of-Work mechanism (i.e. power consumption as it is pointed out above in 2.1.2.2 section), Ethereum is gradually shifting to more efficient Proof-of-Stake procedure which enables to mine blocks according to the amount of money owned by the stakers.

The **network layer** is the last one and it is characterized by the Ethereum structure Peer-to-Peer network composed by nodes that stored a copy of the whole blockchain.

In the synchronization process on the Ethereum blockchain, node `A` can request block synchronization from node `B` following four stages as it is illustrates in Figure 2.19:

1. Node `A` transmits a `GetBlockHeaders` message where it requests the header of the previous block from the node `B`. Node `B` will reply to `A` with a `BlockHeaders` message containing the block header needed by `A`.

2. In order to find ancestor from node `B`, node `A` calls for `MaxHeaderFetch`[8].

[8]The default value of `MaxHeaderFetch` is 256 and the number of block headers send from `B` to `A` should be less then this value.

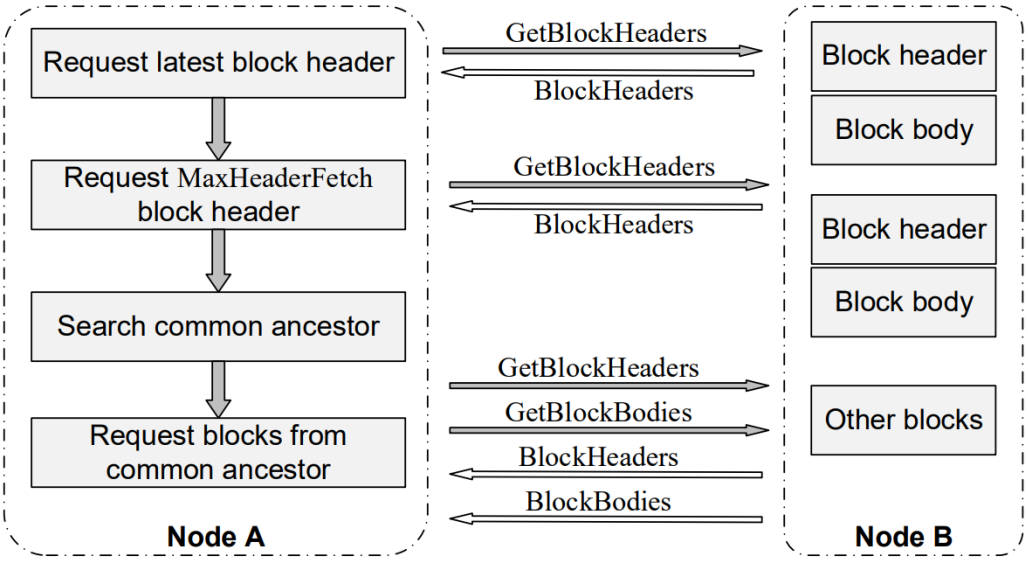

Figure 2.19: Block synchronization process between nodes.

3. If `A` finds a common ancestor, `A` will sychronize with the ancestor by the request `MaxHeaderFetch`.

Looking in detail, to execute a smart contract, a sender has to send a transaction to the contract and pay a fee that is derived by the computational cost of the contract, measured in units of gas. A ***gas-limit*** for contract execution is set in order to avoid system failure and out-of-gas exceptions are verified once this limit has been reached. In addition, smart contracts can call other accounts on the Ethereum network and this allows both to call a function in another contract and to send Ether, the currency in Ethereum, to an account.

Due to the rise of its popularity, this technology has revealed several vulnerabilities of different importance. Despite blockchain technology offering many cryptographic advantages such as digital signature and hashing, Ethereum platform suffers from critical cybersecurity threats indeed smart contracts are irreversible and immutable, once detected bugs can not be easily patched. Moreover, the correctness of execution alone is not sufficient to make smart contracts secure.

2.3.1 Solidity

Solidity is a high-level contract-oriented language similar to JavaScript and C languages. It enables to develop contracts and compile to EVM bytecode. There are other three languages - Serpent, Lisp-Like Language and Mulan - in the Ethereum protocol and they have the same level of abstraction but Solidity is the most common one. **Ethereum smart contract account** is composed by a executable code and a state consisiting of *private storage* and *amount of virtual coins* - Ether - (i.e. the contract *balance*).

Peers transfer Ethers using transactions, moreover they are able to invoke contracts using contract-invoking transactions. A the lowest level the code of an Ethereum smart contract is a stack-based bytecode language run by an **Ethereum Virtual Machine**

(EVM) in each node. Developers define contracts using high-level programming languages. The most commun one is Solidity, a JavaScript-like language that is compiled into EVM bytecode.

Once a smart contract is created at a specific address `X`, it is possible to invoke it by sending a contract-invoking transaction to the address `X`. A typical invoking transaction to the generic address `X` is composed by the payment for the execution and the input data for the invocation.

```
pragma solidity >=0.4.16 <0.8.0;
  contract Storage {
    uint storedData;
        function set(uint x) public {
            storedData = x;
        }
        function get() public view returns (uint) {
            return storedData;
        }
    }
```

The `Storage` is a simple example of smart contract structure. The first line specifies that the source code is written for Solidity version `0.4.26` of a newer version but version `0.8.0` is not included. In thsi way, it is ensure that the contract is not compatible with a new compiler that behave differently. `pragma` is a common instruction for compilers which specifies how the source coed has to be treated.

In general, a Solidity contract is a set of code - its *functions* and data - its *state* that are in a address of the Ethereum blockchain. `uint storedData;` is the statement of the `storedData` variablr of type `uint` which stands for *u*nsigned *i*nteger of *256* bits.

In the example, the contract defines the getter and setter functions in order to enable the modification and the recovery of the variable. that rewards anyone who solve a solution and submits the solution to the smart contract. This contract allows anyone to store a single number that is accessible by anyone that could call `set` with a different value and overwrite the value number. The `Coin` contract enables the creation of a new coin.

```
pragma solidity >0.5.99 <0.8.0;
contract Coin {
    address public minter;
    mapping (address => uint) public balances;
    event Sent(address from, address to, uint amount);
    constructor() {
        minter = msg.sender;
    }
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        require(amount < 1e60);
        balances[receiver] += amount;
    }
    function send(address receiver, uint amount) public {
        require(amount <= balances[msg.sender], "Insufficient balance.")
            ;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }
}
```

The `address public minter;` declares a state variable of type *address* which is a 160-

bits value which is suitable for storing addresses of contracts. The `public` keyword automatically produce a function that enables the get access the value of the state variable from outside of the contract[9].

In the next line, `mapping (address => uint) public balances;` creates a public complex datatype which maps addresses to unsigned integers[10].

The line `e event Sent(address from, address to, uint amount);` declares the *event* emitted in the last line of the `send` function. Like for web application users, Ethereum client can listen for these event. The `constructor` is a special function executed during the creation of the contract and cannot be called later. It permanently stores the address of the user creating the contract. The `msg` variable (together with `tx` and `block`) is a special global variable that contains properties which enable the access to the blockchain. `msg.sender` is the address where the current (external) function call came from. The contract is composed by `mint` and `send` functions. The `mint` function transfer an amount of newly created coins to another address. The `require` function call defines conditions that reverts all changes if not met.

For instance, `require(msg.sender == minter);` confirms that only the creator of the contract can call `mint`, whereas `require(amount < 1e60);` ensures that there are no overflow errors in the future.

Moreover, the `send` function can be used by anyone to send coins to anyone else. If the sender does not have enough coins to send, the `require` call fails and provides the sender with an appropriate error message string.

2.3.2 EVM

The **E**thereum **V**irtual **M**achine is a runtime environment for Ethereum smart contracts. It is a *quasi-Turing complete machine* and the *quasi* qualification comes from the fact that the computation is bounded through the **gas** setting which restricts the total amount of computation done. All smart contracts are sets of bytecode instructions that sequentially executed. Furthermore, the bytecode enables jumps allowing a Turing complete behaviour.

In Solidity, once a contract is compiled, it is converted into a sequence of *opcodes* that are operation codes and they are identified by abbreviations. Table 2.2 shows the most common opcodes used and their fourth column reports the predetermined amount of gas assigned to each of them. This value establishes the computational effort required to perform that operation. The EVM executes bytecodes that are similar to opcodes but they are represented by hexadecimal numbers. 21,000 gas is the minimum quantity of gas that affects the state of the EVM, indeed, this amount is required in order to transfer Ethers through accounts. According to this, execturing a function requires 21,00 gas and the gas needed to preform the certain operation. As far as the read operations are concerned, they are simple views of the state of the EVm so they do not have an execution fee.

In a general scenario, before preforming an operation, the user sets a `gasLimit` that is the maximum amount of gas that he is willing to pay and two scenari can occur. In the former the quantity of gas overcomes the `gasLimit`, so the operation will be aborted, the modification will be rolled-back and the spent gas will be lost. By contract, in the

[9] without the keyword, other contracts cannot access to the variable.

[10] Mappings can be seen as hash tables which are virtually initialised such that every possible key exists from the start and is mapped to a value whose byte-representation is all zeros.

latter the user has set a `gasLimit` higher than the required, the operation will end, and only the used gas will be taken.

The gas mechanism is an *incentive system* for miners, to spent hardware and power costs to validate transactions, and it is a prevention for attackers. Furthermore, Turing complete characteristic of the EVM ensures a limit on the amount of `gasLimit` to set of computable functions.

Opcode	***Name***	***Description***	***Gas***
`0x00`	**STOP**	*Halts execution*	0
`0x01`	**ADD**	*Addition operation*	3
`0x02`	**MUL**	*Multiplication operation*	5
`0x03`	**SUB**	*Subtraction operation*	3
`0x04`	**DIV**	*Integer division operation*	5
`0x08`	**ADDMOD**	*Modulo addition operation*	8
`0x09`	**MULMOD**	*Modulo multiplication operation*	8
`0x10`	**LT**	*Comparison operations*	3
`0x11`	**GT**		
`0x12`	**SLT**		
`0x13`	**SGT**		
`0x14`	**EQ**		
`0x16`	**AND**	*Bitwise operations*	3
`0x17`	**OR**		
`0x18`	**XOR**		
`0x31`	**BALANCE**	*Get balance of the given account*	700
`0x50`	**POP**	*Remove word from stack*	2
`0x51`	**MLOAD**	*Load word to memory*	3*
`0x52`	**MSTORE**	*Save word to memory*	3*
`0x54`	**SLOAD**	*Load word from storage*	800
`0x55`	**SSTORE**	*Save word to storage*	20,000**
`0x56`	**JUMP**	*Alter the program counter*	8
`0x57`	**JUMPI**	*Conditionally alter the program counter*	10
`0x70`	**PUSH**	*Stack operations*	3
`0x80`	**DUP**		
`0x90`	**SWAP**		
`0xf0`	**CREATE**	*Create a new account with associated code*	32,000
`0xf1`	**CALL**	*Message-call into an account*	25,000

Table 2.2: Table of the most used opcodes.

2.3.2.1 Memory Usage in Ethereum

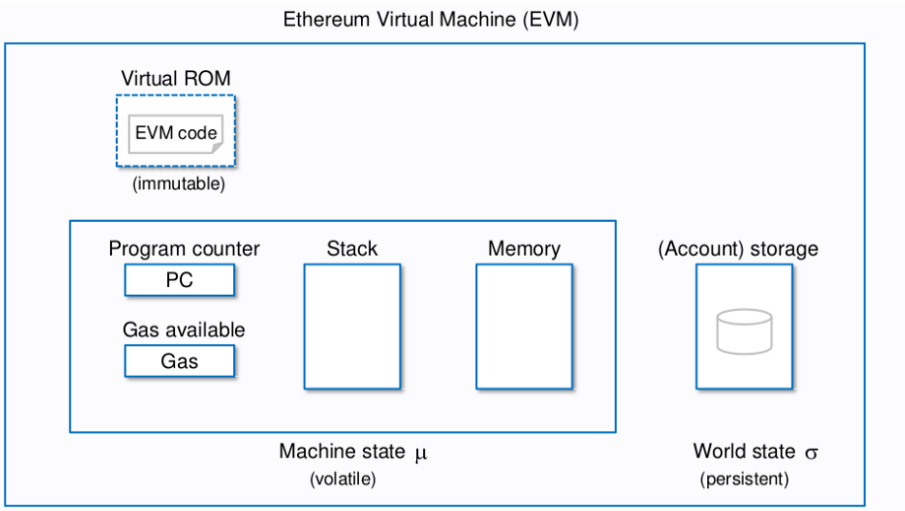

Figure 2.20: Ethereum Virtual Machine Architecture.

As the Figure 2.20 indicates, the EVM has three areas where it stores data. The EVM is a stack machine as a matter of fact, all computations are carried out on a data area called **stack** that contains aal the smart contracts state variables. It has a maximum size of 1024 elements and contains words of 256 bits. The access to this memory area us limited to the top end and it pursues a process that allows to copy one of the top 16 elements to the top of the stack or swap the top element with one of the 16 ones below it. All the other operations, in general, use the fist two elements from the stack and push the result onto this volatile memory. In addition, it is possible to move stack elements to storage or memory to obtain deeper access to the stack but it is not possible to log in arbitrary elements deeper in the stack without removing the top one. In this region,the opcodes to operate with it are **SLOAD** and **STORE**.

The **memory** is the second data region. A contract gets a new cleared instance of it for each message call. Memory is linear and can be addressed at byte level, but reads are restricted to a width of 256 bits, whereas writes can be either 8 bit or 256 bits large. Memory is expanded by a 25-bit word when accessing a unmarked memory word. At the time of expansion, the cost in gas must be paid and its cost steadily increases according to its growth. **MLOAD**, **MSTORE**, and **MSTORE8** are the only three instructions that enable the access to memory area.

Each account has a data area called **storage**, that is *persistent* between function calls and transactions. It is a key-value store which maps 265-bit words to 256-bit words. Its operations are extremely expensive so the user minimize the amount of data stored in this persistent storage to what the contracts has to run. In this kind of memory, it is stored data like derived calculations, caching, and aggregates external of the contract. A contract can neither read nor write to any storage apart from its own. It can be accessed with different instructions, such as **PUSH**, **POP**, **COPY**, **SWAP**, and other ones.

Vulnerabilities

In this chapter, we provide information about the concept of **vulnerability** in this research area and we give a wide explanation of what kind of vulnerabilties are *already found*. The analysis of this topic makes us aware of the diffusion of issues that are hidden in the whole blockchain engine.

Moreover, the chapter tskes into account analysis methods that enable the developing strategies which aim to detect and mitigate the ever-detecting security flaws and their consequenties.

In Section 3.1, we explain of the most relevant **surveys** related to **Ethereum vulnerabilities** so far made, then we introduce the classification that allows us to choose the kind of vulnerability took into account in this research.

In Section 3.2, we analyze how **Ethereum** faced with different **security attacks**. Once we analyzed forking concept and its different categories, we take into account each fundamental hard fork that has characterized Ethereum evolution. Following this, we show several known attacks that have occurred in Ethereum blockchain.

In Section 3.3, we illustrate several types of **technical methodologies** used to implement their security analysis on smart contract. Once we have introduced the exisiting surveys of this research area, we focus our attention on **tools** that use static analysis methods because they represent the perfect starting point for the progress of the research.

3.1 Relevant Surveys

The identification and the analysis of vulnerabilities in Ethereum smart contracts give the possibility to overcome security problems that are linked to them and have an impact on the whole system.

The main surveys that are taken as points of reference are the:

- N. Atzei, M. Bartoletti and T. Cimoli[7] group sixteen vulnerabilities in three classes, according to the level where they are introduced (Solidity, EVM bytecode or blockchain respectively).
- M. Bartoletti and L. Pompianu[9] illustrate an empirical analysis of smart contracts, then they categorize them into five types according to their intended application domain.
- X. Li, P. Jiang, T. Chen, and X. Luo[26] review the security of blockchain systems. Moreover they analyzed the security issues of smart contracts in the Ethereum network.

Starting from these important researches, other surveys on smart contracts vulnerabilities have been conducted. The most relevant one is the one given by H. Chen, M. Pendleton, L.Njilla, and S. Xu[12].

It explains vulnerabilities detected by the analysis of important attacks that have damaged Ethereum network and its *engine*.

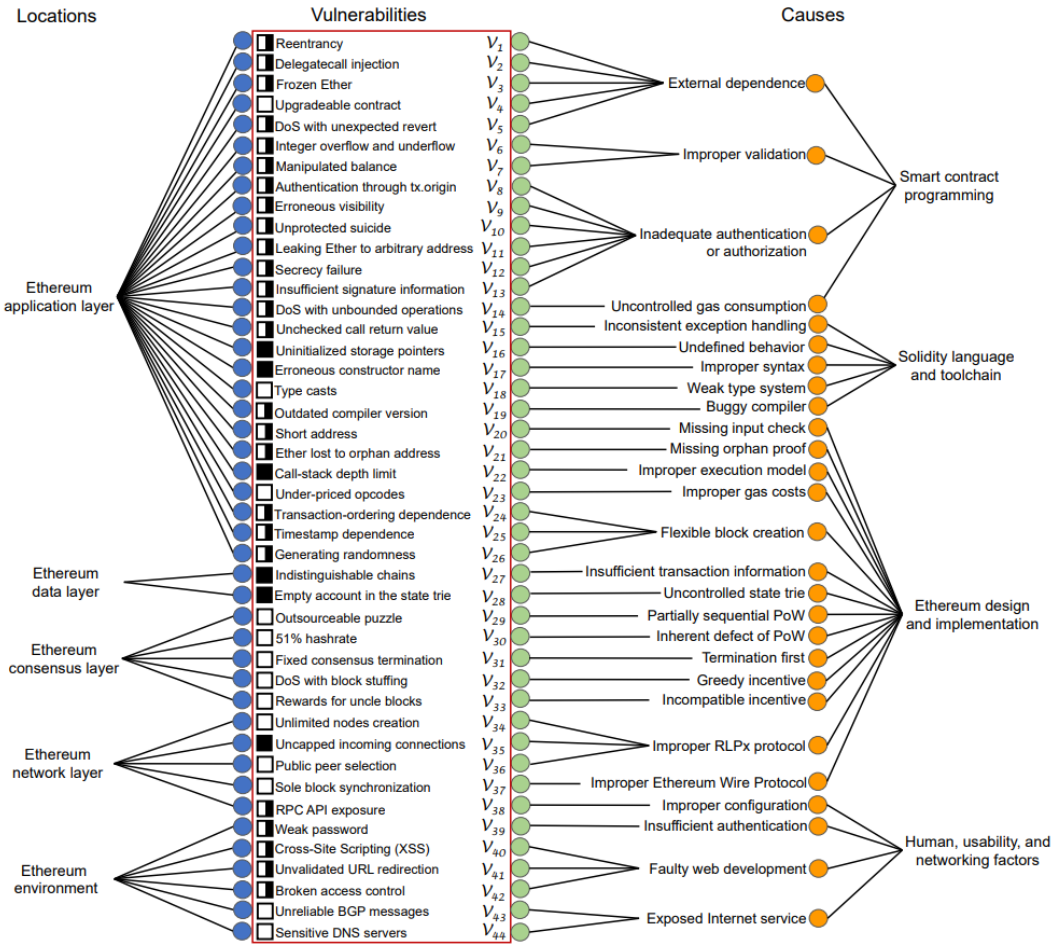

Figure 3.1: A classification of Ethereum vulnerabilities and their treatments.

They analyze the Ethereum system as a layered architecture and they list 44 vulnerabilities categorizing them according to the layer to which they belong, as it is shown in the Figure 3.1 [1]. Once we had a clear vision of the amount of vulnerabilities detected in Ethereum blockchain, we classify them in order to get to the causes. Getting this notion allowed us to understand the lack of analysis and validation that Ethereum scenario had.

[1] There are three different categories of vulnerabilities' situation: the completely filled squares stand for the already eliminated issues, the partially ones for those avoidable and the unmarked are the pitfalls which have to be eliminated.

Location	Causes	**Field**
Ethereum Application Layer	enternal dependence	**Smart contract programming issues**
	improper validation	
	inadequate authentication	
	uncontrolled gas consumption	
	inconsistent exception handling	**Solidity language and toolchain**
	undefined behaviour	
	improper syntax	
	weak types system	
	buggy compiler	**Ethereum Design and Implementation**
	missing input check	
	missing orphan proof	
	improper execution model	
	improper gas costs	
	flexible block creation	
Eth. Data Layer	insufficient transaction information	
	uncontrolled state trie	
Ethereum Consensus Layer	partially sequential PoW	
	inherent defect of PoW	
	termination first	
	greedy incentive	
	incompatible incentive	
Ethereum Network Layer	improper RLPx protocol	
	improper Ethereum Wire Protocol	
	improper configuration	**Human, usability, and networking factors**
Ethereum Environment	insufficient authentication	
	faulty web development	
	exposed Internet service	

Table 3.1: Categorization of attacks.

The most common issues of blockchain environment help us to list the scope, as shown in *Table 3.1*, which we should undertake in order to make a contribution that overcome a certain pitfall. They show all the vulnerabilities that have been detected since 2015 so it is possible to visualize already patched pitfalls, the ones that can be avoided by best practices and the open ones.

The research is precise and the additional causes classification gives the possibility to group the vulnerabilities into areas of interest. Looking at the application layer detected vulnerabilities, they could be listed in six groups:

1. the ones that are caused by **external dependencies**,
2. those produced by **improper validation**,
3. the ones that are originated by the **inadequate authentication**,
4. those linked to **wrong exception** handling,
5. the problems followed by the difficult **creation of flexible blocks**,
6. the issues of the **gas consumption**.

3.2 Ethereum Scenario

Despite all the security enhancements, Ethereum has faced with several attacks which aim to obstruct the natural flow or even fully destroy the network. Pitfalls relating to cryptocurrency wallets, smart contracts, transaction authentication, mining pools, and blockchain networks are frequently exploited by adversaries.

Ethereum has incurred on several stages in order to update the whole blockchain engine and provide solution to detected vulnerabilities. The whole blockchain engine has modified to provide solution in the detection of noticed vulnerabilities.

Forking is a process that involves the need to make technical improvements and updates to the code. It implies some temporary or permanent divergences in Blockchain environment. Public blockchains consist of open source development communities which carry on the codebase and the miners who run the software and validate the data structure. In this scenario, forks are common processes for the development of public blockchains.

It can assume two different forms and both provoke different consequences The former is caused by an inernal division of the community (i.e.Ethereum and Ethereum Classic) wherease the latter generates the creation of a new blockchains with important alteration compared to the source code (i.e. Bitcoin and Zcash).

3.2.1 Different Types of Forks

From a visual point of view, the fork process produces a split of the chain into two different branches. According to the nature of change, the fork con be categorized into:

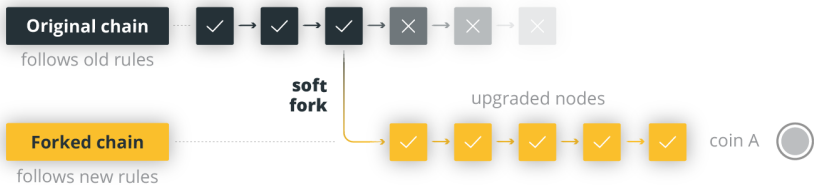

Figure 3.2: Soft fork mechanism.

- **Soft fork** is an update on the blockchain protocol that is *backward compatible*.

  It introduces some changes in the code that do not affect the functional continuity with older version but it may provoke a temporary divergence in the Blockchain.

  According to this, the new forked chain will follow new regulation but the previous ones are still valid. This type of modification requires only a majority of the miners upgrading the rules[2]. New transaction types can be added as soft forks, requiring only that the participants understand them.

- **Hard fork** consists of changing the blockchain protocol in a way that is *backward incompatible* and it introduces a discontinuity with older version.

  It is clear from the Figure 3.3, that the nodes running the previous versions will not be accepted by the newest one.

[2]By contrast, the hard fork can be done when all nodes have to upgrade ad agree on the new version.

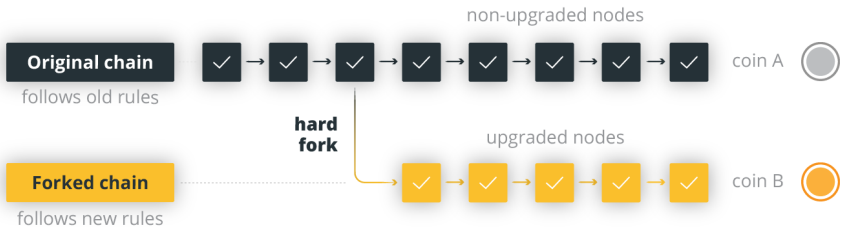

Figure 3.3: Hard fork mechanism.

A hard fork is a complete modification of the protocol that sets the validity or invalidity of blocks and transactions.

Transactions on the new chain will not be valid on the older chain and the whole network will have to upgrade to the last versions of the protocol in order to be in the new froked chain. Hard Fork is pursued when the major part of the mining community.

3.2.2 Ethereum Roadmap

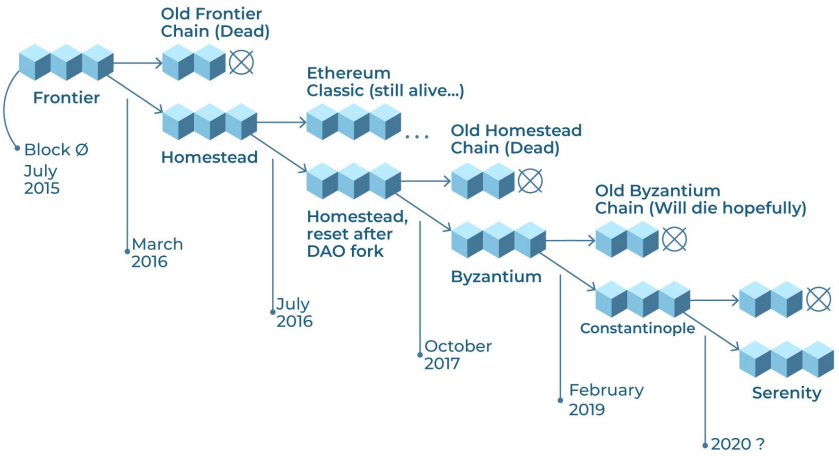

Figure 3.4: Ethereum improvements table.

From the beginning, Ethereum has undergone significant changes over the time in the form of ***hard forks*** that enables the development of the whole network. In July 2015, there was the launch of **Frontier** that is the original release of the Ethereum network. It is a prototype of decentralized applications and enables the mining of Ether. The launch consisted of a *Genesis block*, that included 8893 transactions for those who bought Ether durign the presale. Initially, Ethereum had a fixed `gasLimit` of 5,000 gas per block. Each transaction had a basic cost of 21,000 gas, according to it, Ethereum blockchain only allowed for mining blocks. Once this limit was changed, the `gasLimit` value raised to 3 million of gas, subsequently block #46,147 contained Ethereum's first transaction. The first production release of Ethereum was **Homestead** which is introduce in March 2016 and introducing Solidity as smart contract programming language, provided an improvement of development capability. Moreover, it expanded user accessibility with the establishment of the Mist wallet and a better interface the procedure of sending and receiving Ether. On 17_{th} June 2016, a vulnerability detected inside the `DAO` contract

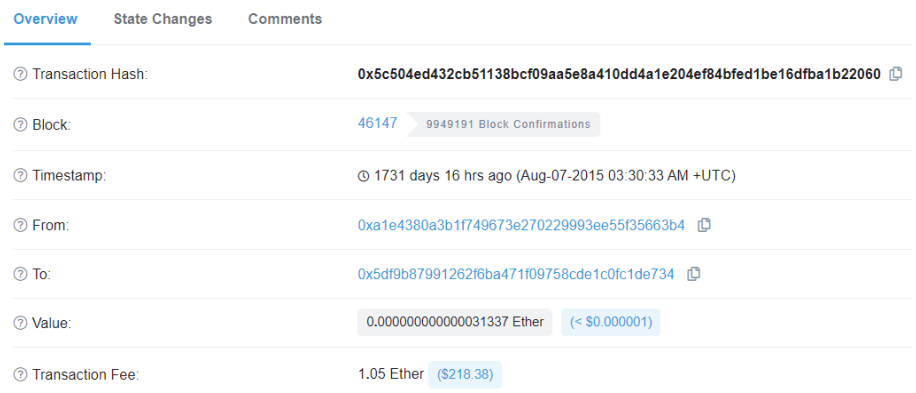

Figure 3.5: Ethereum's first transaction.

had been exploited to drain 3.6 million ETH from the fund. This amount of ETH funds was frozen for 28 days before they could be transferred. If no action would have been taken, the attacker would have owned around 4.4% of the total supply of ETH. A controversial proposal (EIP 779) was drawn up in order to make a countermeasure and change the code of the attacker's lockup contract. On July 20^{th}, a major part of mining power supported a fork that allowed the EIP 779 proposal, while the minority decided to split off and rename the old chain to *Ethereum Classic*.

Metropolis represented the third stage of Ethereum roadmap. It aimed to improve security, scalability, privacy, flexiility, and efficiency to the blockchain network.

It is divided into two parts:

1. **Byzantium** that was *backward compatible* upgrade that introduced *zk-SNARKS*, a privacy-preserving technology used by Zcash. Another upgrade concerned the growth of the delay of the "difficulty bomb" which increased block times in order to reduce block rewards from 5 to 3 ETH. The reduction was an incentive to make progress towards the proof-of-stake.

2. **Constantinople** which was an hard fork that was deployed and introduced several upgrades around maintenance and operations of the chain. It has different phases.

   (a) During **St.Petersburg**, there were pursued improvements that made cheaper to do thing on-chain. The block reward was reduced from 3 to 2 ETH.

   (b) The **Istanbul** fork introduced more privacy capabilities like a significant re-balancing of the gas pricing with the computation costs of the EVM opcodes. In addition, it enhanced denail-of-service attack resilience and made layer 2 solutions more perfomant.

Serenity is the final stap of Ethereum's development that will make the final transaction from the Proof-of-work consensus algorithm to a Proof-of-Stake engine. This new consensus mechanism will reduce the computational and electrical resources needed to secure the Ethereum network.

3.2.2.1 Ethereum Improvement Proposals

Ethereum community enables anyone to make its own contributio in order to improve Ethereum structure and performances.

Ethereum Improvement Proposals[1], known as EIP, are strandards that specifies potential features or processes for Ethereum environment. They hold technical specifications for the proposed modifications which have to be discussed by the whole community. There are three kind of EIP[3]:

1. **Standards Track EIP** reports any change that affect ethereum implementations (i.e. modification to the network protocol, in block or transaction directives or any other proposed application standards/ conventions).

   There are four sub-categories of Standards Track:

   - **Core** covers developments that require consensus fork or that concern the miner/node strategy changes;
   - **Networking** includes network protocol specifications;
   - **Interface** involves advances related to client API/RPC specifications and standards;
   - **ERC** belongs to application-level standards and conventions like wallet format, token standards (ERC-20), name registries or URI schemes.

2. **Meta EIP** describes a process relating to Ethereum or suggest a change to a process. Compared to the Standards ones,they are used in different area of use[4].

   They present an implementation and they need the community consensus. They are not just guidelines but they have to be followed.

3. **Informational EIP** reports an Ethereum design issue, provides common guidelines or gives advices to the community but it *does not propose a new feature.*

3.2.3 Main Attacks

We take into account twelve known attacks which have occurred in Ethereum envornment in the previous years. We analyze the most famous attacks belonging to the application layer, then we introduce a network layer attack, the *Eclipse attack*, which tries to change the correct peers communication. Moreover, we look over several attacks against the whole environment that concern decentralized platforms.

- In Ethereum network, a **DAO** is a Decentralized Autonomous Organization. Its aim is to order rules and decision, making apparatus of an organization and creating a structure with decentralized control.This Organization follow specific stages.

  First of all, there is the writing of the code of smart contracts, then the Initial Coin Offering (ICO) step starts. During this period, users add cash to the DAO by acquiring tokens that represent ownership to allow the store of the needed resources. Following this, the DAO starts to operate. The token holders propose to the DAO on how to spend the money and the members. Once a proposal is adopted by the majority, the amount of money is sent to the proposer's account. Moreover, the quantity of money of those did not agree with the proposal is distributed to each of them via contract accounts creation processes and mechanism is implemented in the `splitDAO()` function.

[3] `https://github.com/ethereum/EIPs`
[4] The Standards Track EIPs are focused on the Ethereum protocol field.

On 17^{th} June 2017, the `DAO` was attacked by using the *reeentrancy* vulnerability, and the attacker started draining Ether from the main address where it was stored. In detail, as it is shown by the Figure 3.6, the attack follows this process:

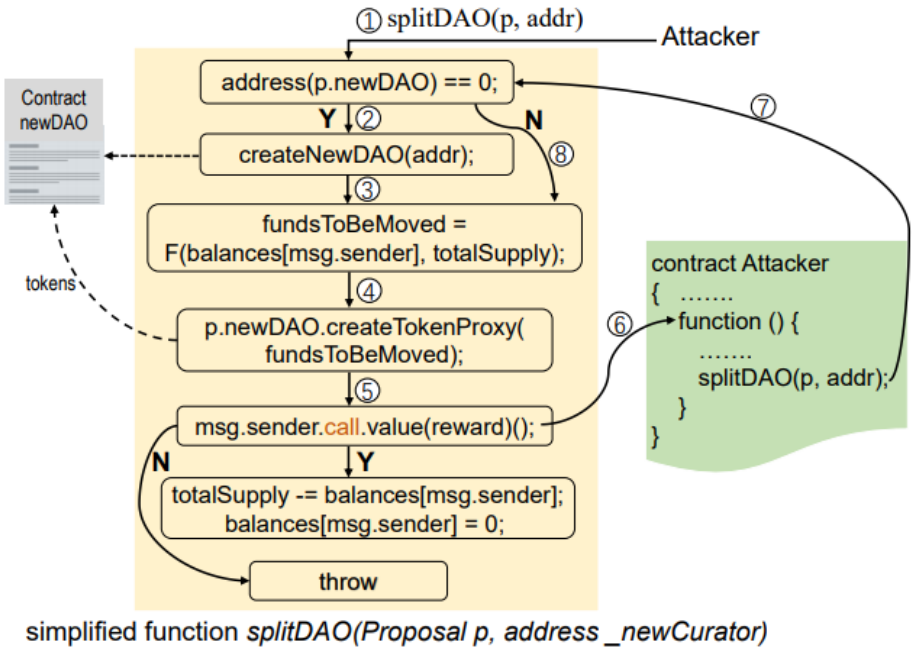

Figure 3.6: DAO attack explanation.

1. the minority requests for a refund, the `DAO` contract creates a new `DAO` contract account (Step 1 and 2);
2. the investor's money is tranferred to the new `DAO` contract (Step 3 and 4);
3. the requesting investor may receive the reward for the previous contribution (Step 5 and 6).
   The issue is verified because the value of the refunded tokens for the inverstor is strictly related to the state variables `balances[msg.sender]` and `totalSupply` that are updated at the end of the function after the `msg.send.call.value()`.
4. The attacker may recursively call the function (Step 7) before the state variables are updated and he may draw more money than he has to obtain.

- A *multisignature wallet* is a particular smart contract that need multiple private keys in order to unlock a wallet that protects Ether.

```
1     contract Wallet {
2     address _walletLibrary = new WalletLibrary();
3     address owner;
4     ...
5     function() payable {
6         if (msg.data.length > 0) _walletLibrary.delegatecall(msg.
              data);
7     }
8     }
9
```

```
10      contract WalletLibrary {
11      ...
12      function initWallet(address[] _owners, uint _required, uint
            _daylimit){
13      initDaylimit(_daylimit);
14      initMultiowned(_owners, _required);
15      }
16  17 }
```

It supported by Parity client contains two contracts:

1. the `WalletLibrary` that has all the main functions of the wallet;
2. the `Wallet` which holds a reference (to the library) that sends all the unmatched function calls to the library contract using the `delegatecall`.

The Parity multisignature wallet was compromised twice in 2017. The first attack used the *delegatecall injection* and the *erroneous visibility* vulnerabilties. The attacker became the ownership of the `Wallet` contract and it sent a transaction to the contract using the `msg.data` from the `initWallet()`. Since the contract `Wallet` did not check the function, the contract's fallback function[5] was triggered to delegate the wallet initialization task to the library. Subsequently, the library replaced the original multi-owner of the contract `Wallet` with the attacker's address (which is specified in `msg.data`). In this attack the `initWallet()` function was not an *internal* one so it can be externally call via `delegatecall`, the library is a stateful contract and it can change the state of the `Wallet`, the Wallet's fallback function did not verify the function being called.

In the second attack, it was exploited the *unprotected suicide* and *frozen* Ether pitfalls.In order to overcome the issues caused the first attack, the Parity developers added a modifier, `only_uninitialized` that protect function`initWallet()`. However, also the library was defined uninitialized and this enalbed an attacker to bypass the modifier and set himself as the owner of the library. Once taking over the library, the attacker invoked the suicide method to kill the library, causing all of the Wallet contracts relying on the library unusable.

- `BECToken`[6], ERC-20 contract, was attacked on 22^{nd} April 2018 by the exploitation in the *integer overflow* vulnerability and it caused an amount of token stolen and a temporary shutdown of token trading at exchange.

```
1       function batchTransfer(address[] _receivers, uint256 _value)
            public whenNotPaused returns (bool) {
2       uint cnt = _receivers.length;
3       uint256 amount = uint256(cnt) * _value;
4       require(cnt > 0 && cnt <= 20);
5       require(_value > 0 && balances[msg.sender] >= amount);
6
7       balances[msg.sender] = balances[msg.sender].sub(amount);
8       for (uint i = 0; i < cnt; i++) {
```

[5] From the Solidity documentation provided by the Community[4], it is possible to know that a `fallback` function has no arguments, cannot return anything and must have an `external` visibility. Its execution is done during a call to the contract none of the other functions match with the input signature, or if no data is supplied at all or the is no *receive Ether function*. It always receives data but in order to get also Ether has to be set as `payable`. It can only rely on 2300 gas whether it is used in place of a receive function.

[6] https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d#code

```
9           balances[_receivers[i]] = balances[_receivers[i]].add(
                _value);
10          Transfer(msg.sender, _receivers[i], _value);
11      }
12      return true;
13  }
```

The function `batchTranfer()` has an issue. It enables the tranfer of tokens to multiple recipients through two arguments:

1. `_receivers`: the array of the recipients' address;
2. `_value`: the number of tokens.

The statement in Line 3 determined the number of tokens the sender should give for a certain transaction, but it was incorrect as a matter of fact it may have an integer overflow: if the `_value` is set to 2^{255} and `_receivers` to two account linked to an attacker, the attack overflows to 256-bit variable `amount` and makes it zero. According to this, the attack bypasses the ceckes that are in Line 4 and 5 and allows the send of a large amount of tokens.

- `GovernMental` contract is an array-based pyramind Ponzi scheme, where the last partecipant wins a jackpoi whether no one gets the scheme within 12 hours after the last participant.

  In the contract are several pitfalls. Firstly, the *DoS with unbounded operations* vulnerability which is verified when the array storing the participants is too large and the quantity of gas needed for operating on that array will not respect the *gasLimit*. As consequence of this issue, the winner cannot receive the 1,000 ETH jackpot.

  Secondly, the *unchecked call return value* vulnerability is detected whether the contract does not check the returned value when sending profits to the winner. The owner of the contract fails the payment:

  – calling 1024 contracts before the target callee contract of the payee that produces the callee contract to return FALSE and this means that it does not get any payment;

  – the caller contract supposes to check this returnd value and then go on but it does not do it. So, the is a loss of money belonged to the caller contract's owner.

  Another attack exploits *transaction-ordering dependence* vulnerability which a malicious miner can discard same transaction linked to `GoverneMental` contract or reorder transaction in order to become the last player in each round.
  The last attacke uses the *timestamp dependence* vulnerability which a miner can modify `block.timestamp` to be the winner.

- `HYIP` is a Ponzi scheme which is "*an investment fraud that involves the payment of purported returns to existing investors from funds contributed by new investors. The organizers of these scheme often chatch new investors by promising to invest funds in opportunities claimed to generate high returns with little or no risk. Ponzi schemes require a constant flow of money from new investors to continue and they inevitably collapse.*"[8] This process is performed by the function `performPayouts()` that contains the *DoS with unexpected revert* vulnerability.

```
1   contract HYIP {
2       uint constant INTERVAL = 1 days ;
3
4       struct Investor {
5           address addr ;
6           uint amount ;
7       }
8       Investor [] private investors ;
9       ....
10      function performPayouts () {
11          ...
12          uint idx ;
13          for ( idx = investors.length ; idx - - > 0; ) {
14              uint payout = (investors[idx].amount*33)/1000;
15                  if (! investors[idx].addr.send(payout)) throw ;
16      }
17  }}
18
19
20  contract Mallory {
21      address victim = 0 x23 ...;
22      address private owner ;
23      bool private attack = true ;
24      function () payable {
25          if (attack) throw ;
26      }
27      function stopAttack () {
28          if ( msg.sender == owner )
29          attack = false ;
30      }
31  }
```

The attack follows this steps. First of all, the attacker writes the `Mallory` contract which is the explotation one where the attacker invests and throws an exception in the fallback function (Line 24). Following this, the function `perfomPayout()` is used to pay the investors, then the fallback function is invoked, throws an exception which provokes a reversion of the money transfer (Line 15) and DoS to `HYIP`.

Subsequentialy, the attacker can blackmail `HYIP` by undoing the `throw` operation (Line 25) via function `stopAttack()` that can be done by the owner of teh contract.

- `Fomo3D`[7] was a very popular Ponzi game in 2018, where the last participiant who buys key before the timer runs out won the jackpot. The price of the keys constantly rises with the number of buyers. When a key was sold, the countdown lasts 30 seconds. Moreover, Fomo3D implemented an airdrop lottery to get participants. Each purchase over 0.1 ETH, the buyer has a random chance to be selected for a profit from the prize pool.

  The first attack is focused on the airdrop mechanism.

  It exploited the *generating randomness* vulnerability.

```
1       function airdrop()
2         private
3         view
4         returns(bool)
```

[7] `https://etherscan.io/address/0xa62142888aba8370742be823c1782d17a0389da1#code`

```
5       {
6           uint256 seed = uint256(keccak256(abi.encodePacked(
7
8               (block.timestamp).add
9               (block.difficulty).add
10              ((uint256(keccak256(abi.encodePacked(block.coinbase))))
                    / (now)).add
11              (block.gaslimit).add
12              ((uint256(keccak256(abi.encodePacked(msg.sender)))) / (
                    now)).add
13              (block.number)
14
15          )));
16          if((seed - ((seed / 1000) * 1000)) < airDropTracker_)
17              return(true);
18          else
19              return(false);
20      }
```

The `airdrop()` generates a random seed by performing a deterministic computation on the current block state and the address of `msg.sender`.

If the seed satisfies a specific condition (Line 16), the current key buyer obtains the airdrop. But the block information is predictable, so the attacker can simply pre-compute the address of new contracts and bruteforces for the successful seed.

The second attack involves tha winning procedure. The attack expolited the *DoS with block stuffing* vulenrability and allows the attacker to win the prize of around US$3M. The attack follows this procedure: when the timer of the game shows three minutes, the attacker bought a key and then sent multiple transactions to his accounts with enough *gasPrice*. Because of the choice of miners, these transactions were first stored into blocks. The maximum amount of gas consumption for each block is determined ans any transactions related to `Fomo3D` were not collected into blocks. This situation congests the network until the end of the game and the attacker becomes the last player.

- The `ERC-20 signature replay` attack exploits the *insufficient signature information* vulnerability. Looking in detail, during the tranfers of ERC-20 tokens, the user has to own enough Ether to cover the transaction fee, which can be difficult when the user does not have any Ether. The proxy transfer method is introduced in order to overcome this issue as a matter of fact it enabled userc to pay transaction fee in tokens, as opposed to paying only in ether in generic ERC-20 contracts. A user can authorize a proxy to carry out a transaction and reward the proxy. We will explain this scenario through an example which has Alice as sender and Bob as recipient[12]. As shown in Figure 3.7, Alice has to transfer 100 MTC tokens to Bob. First of all, She sends a signed message off-chain to a proxy, then the proxy makes the transaction to transfer 100 tokens to Bob and it receives 3 tokens from Alice as service fee. The signature is verified using the function `transferProxy()`.

```
1       function transferProxy(address _from, address _to, uint256
            _value, uint256 _fee,
2       uint8 _v, bytes32 _r, bytes32 _s) public returns (bool){
3
4       if(balances[_from] < _fee + _value || _fee > _fee + _value)
            revert();
5
6       uint256 nonce = nonces[_from];
```

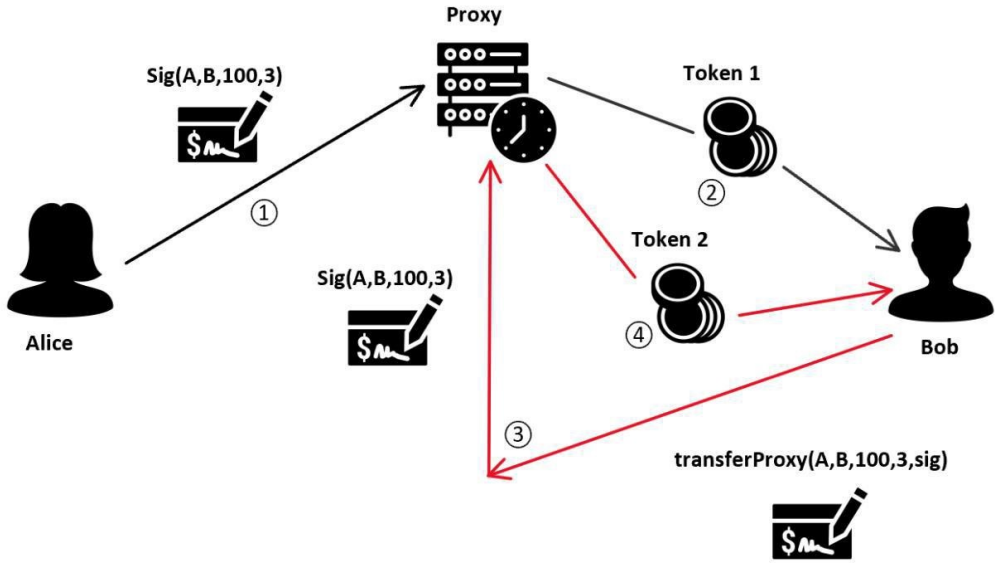

Figure 3.7: ERC-20 signature replay atteck via `transferProxy()`.

```
7       bytes32 h = keccak256(_from,_to,_value,_fee,nonce);
8       if(_from != ecrecover(h,_v,_r,_s)) revert();
9
10      if(balances[_to] + _value < balances[_to]
11          || balances[msg.sender] + _fee < balances[msg.sender])
                revert();
12      balances[_to] += _value;
13      emit Transfer(_from, _to, _value);
14
15      balances[msg.sender] += _fee;
16      emit Transfer(_from, msg.sender, _fee);
17
18      balances[_from] -= _value + _fee;
19      nonces[_from] = nonce + 1;
20      return true;
21  }
```

At this point, Bob gets 100 Tokens from Alice and, following this, he replays with a new transaction of 100 MTC tokens from Alice. Subsequently the Proxy carries out the new transaction without Alice's autorization. This function uses Solidity function `ecrecover()`[8] in order to identify Alice's address that issued the signature. But Alice's address, which is bound to her signature, is not proved in the off-chain message so the signature can be accepted as valid with respect to any token contract address (i.e. MTC or GGoken). This means that Bob can replay the signed message to require other Tokens and obtain extra money to Alice, like it is illustrated in the third and fourth step respectively.

- `Rubixi` contract is a Ponzi scheme which contains the *erroneous constructor name* vulnerability. According to it, the constructor function has an incorrect name that allows anyone th become the owner of the contract. This issue has been primarily addressed in the Solidity compiler versions prior to `0.4.22`. Indeed, the

[8] "`ecrecover()` *is a function that recovers the address associated with the public key from elliptic curve signature or return zero on error.*" (Ethereum, 2018)

contract's constructor is a function declared with the same name as the contract that is executed one-time upon the creation of the contract.

```
1     contract Rubixi {
2         //Declare variables for storage critical to contract
3         uint private balance = 0;
4         uint private collectedFees = 0;
5         uint private feePercent = 10;
6         uint private pyramidMultiplier = 300;
7         uint private payoutOrder = 0;
8         address private creator;
9         //Sets creator
10        function DynamicPyramid() {
11                creator = msg.sender;
12        }
13        modifier onlyowner {
14                if (msg.sender == creator) _
15        }
16        struct Participant {
17                address etherAddress;
18                uint payout;
19        }
20        Participant[] private participants;
21        //Fallback function
22        function() {
23                init();
24        }
25        //init function run on fallback
26        function init() private {
27                //Ensures only tx with value of 1 ether or greater
                      are processed and added to pyramid
28                if (msg.value < 1 ether) {
29                        collectedFees += msg.value;
30                        return;
31                }
32                uint _fee = feePercent;
33                //50% fee rebate on any ether value of 50 or
                      greater
34                if (msg.value >= 50 ether) _fee /= 2;
35                addPayout(_fee);
36        }
37        //Function called for valid tx to the contract
38        function addPayout(uint _fee) private {
39                //Adds new address to participant array
40                participants.push(Participant(msg.sender, (msg.
                      value * pyramidMultiplier) / 100));
41                //These statements ensure a quicker payout system
                      to later pyramid entrants, so the pyramid has a
                      longer lifespan
42                if (participants.length == 10) pyramidMultiplier =
                      200;
43                else if (participants.length == 25)
                      pyramidMultiplier = 150;
44                // collect fees and update contract balance
45                balance += (msg.value * (100 - _fee)) / 100;
46                collectedFees += (msg.value * _fee) / 100;
47                //Pays earlier participiants if balance sufficient
48                while (balance > participants[payoutOrder].payout)
                      {
49                        uint payoutToSend = participants[
                              payoutOrder].payout;     participants[
```

```
                                payoutOrder].etherAddress.send(
                                payoutToSend);
50                          balance -= participants[payoutOrder].payout
                                ;
51                          payoutOrder += 1;
52                  }
53          }
54          ....
55  }
```

If the name is incorrect, the constructor becomes a public function that can be invoked by any EOA. `Rubixi` contract was originally named `DynamicPyramid` and it was later renamed by the developer to `Rubixi`, but, as we can see in the thirteenth line, the constructor was not updated. In this scenario, anyone can calls the public function `DynamicPyramid()` to become the owner of the contract and therefore steal its funds. The vulnerability has been eliminated, starting Solidity version `0.4.22`, the keyword `constructor` is been introduced in order to distinguish the constructor function from other regular functions.

- `Eclipse Attack`[38] enables an attacker, who can divert the connection of some nodes of the P2P network in order to isolate those nodes from the whole network.

  This action can be done manipulating the targeted nodes' routing tables which exploits *unlimited node creation* and *uncapped incoming connections* vulnerabilities.

  Once the client has restarted the system, it has no connection and the attacker starts incoming connection from previously created node IDs to the victim.

  A node is eclipsed when its conection slots[9] contained incoming connection from the attacker that monopolizes the link of communication and reaches the upper bound limit of the incoming TCP connections.

- `EtherDelta` is a famous extrachange for users to market ERC-20 tokens in a trustless manner in return for a small percentage fee charged on each transaction. It was affected by a code injection which caused the *Cross Site Scripting* vulnerability in September 2017, producing the loss of thousands of dollars in cryptocurrency.

  The attacker created a new token contract that has several malicius JavaScript code in the token's name.

  EtherDelta pulls out a newly created token's name from the token contract's code and shows it on the website. Moreover, in order to perfom the token-trade transaction, a user has to insert his private key and his account address to the browser to sign the transaction.

  As consequences of these actions, when the name of the new token was shown on the user's browser, the malicious JavaScript code started stealing his private key that caused the loss of money protected by the privete key.

- `Enigma` is a decentralized investiment platform that was attacked during its ICO on 21^{st} August 2017 because of the exploitation of the *weak password* vulnerability.

[9] 25 by default

The attacker obtained the Enigma founder's password and he took control of the company's comunication channels, email lists and Google account hosting the ICO's presale.

Moreover, the attacker changed the offical ICO contract address with its own one and sent messages to request buyers in fraud presales.

- `CoinDash`, a portfolio management platform, was compromised owing to an explotation of the *broken access control* vulnerability during the course of its ICO[10] in July 2017. It caused a loss of US$7M worth of Ether within a few minutes.

  The attack is composed by getting in the infected web application that hosts CoinDash's website and replacing the ICO contract address with one controlled by the attacker.

- `MyEtherWallet`, the web's popular client-side Ethereum wallet, has been compromised by a DNS attack on 24^{th} April 2018. The attack exploits a joint BGP

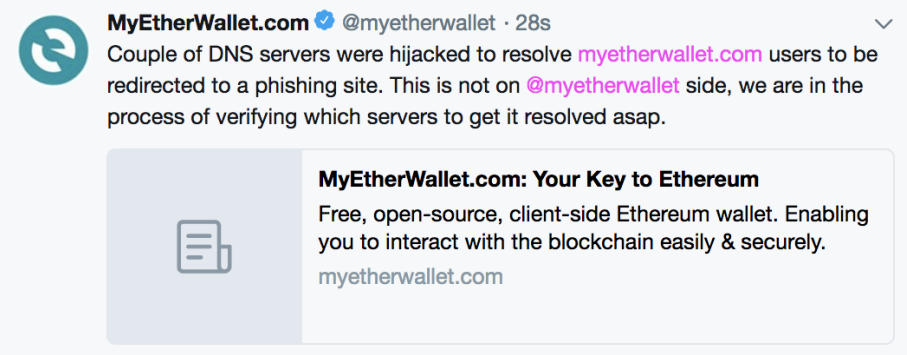

Figure 3.8: MyEtherWallet team confirmation on Twitter.

and DNS hijacking to mislead users to a counterfeit version of the website and compromised tha victims' wallet and stole US$17M.

When users enter in their MyEtherWallet, their requests were directed to the fake DNS servers that returned IP addresses ti direct users to a phishing website.

The users proceeded to login to this site expoising their passphrases and their keys.

[10] ICO is basically blockchain crowdsale, the cryptocurrency version of crowdfunding.

3.3 Security Analysis Methods

In Ethereum environment, smart contracts have a fundamental role in the execution of transactions. They facilitate the application of the blockchain technology, there are several security risks and vulnerability in the smart contracts.

Their **immutability** is the **most critial challenge**, indeed whether one of the engaging parties modifies the digital agreement, the trust in the contract will valish. Moreover, this kind of contracts can not be cancelled, even if in their codes are detected sofware bugs or vulnerabilties.

Futhermore, it is also hard to test them during their run-times because they constantly interact with other samrt contracts and they repearlty invoke external chain services. So, it is clear that smart contracts must be in-depth tested before their release. As a follow-up, a large number of tools have been developed and they use different approaches and techniques that identify pitfalls in the Ethereum environment.

From the literature [31], the security analysis methods of smart contract are categorized in three different type:

- *Static Analysis*
- *Dynamic Analysis*
- *Formal Verification Methods.*

Starting from this categorization, we focus on static and dynamic groups and we take into consideration different methodologies for each of them.

3.3.1 Static Analysis

Static analysis is a technique of analyzing a computer program or compiled code in a **non run-time environment**. It inspects the programming code without executing the program. In general, it examines:

- all possible behaviours,
- vulnerable patterns,
- flaws what it is expected in run-time.

By contrast, one of the negative aspect of this method is that it can not detect vulnerabilities that occur during the execution time. Moreover, it may introduce false positive results. The most common methodologies of this type of analysis are:

- *Symbolic execution*
- *Control Flow Analysis*
- *Pattern Recognition*
- *Rule-base Analysis*
- *Compilation*
- *Decompilation*

3.3.1.1 Symbolic Execution

Symbolic execution is a technique for systematic exploration of the behavior of a program that uses symbolic inputs. The power of this kind of analysis as systematic bug finding has been undersood only during the last decade. Symbolic execution translates the values of program variables as *symbolic expressions* of the input symbolic values. Every symbolic path has a formula that is a condition in order to progress. A path is infeasible whether its path condition is unsatisfiable. Otherwise, the path is feasible.

It has been shown as symbolic execution gives better results than other methodologies (e.g. dynamic program analysis) because it takes one path at time and this allows to achieve better precisions. However, whether inputs take the same path through the program, there is savings over testing each of them. In addition, it allows the discovery and exploration of potential paths in a program but this positive aspect becomes a drawback in scenario where the amount of detected paths are exponential. Indeed, one of the main limitations of it is the *path execution* problem. In this situation, the number of feasible paths grow exponentially with the subsequent increase in program size and there is a possible generation of unbounded loop iterations.

3.3.1.2 Control flow Analysis

According to the research provided by F.E.Allen[6] in this field, "*Any static, global analysis of the expression and data relationship in a progrma requires a knowledge of the control flow of the program.*" In order to produce optimized programs, control flow analysis has been embedded in manu compilers. Control flow analysis codifies the flow connection in the program and its outcome of is the ***Control Flow Graph*** (CFG) that describes the control relations between certain source code elements of the application. A CFG is a directed graph in which the **nodes** represent basic blocks and the **edges** constitute control flow paths.

A ***directed graph***, G can be denoted by $G = (B,E)$ where:

- B is the set of blocks $\{ b_1,b_2,....,b_n \}$
- E is the set of directed edges $\{(b_i,b_j),(b_k,b_l),....\}$ Each directed edge is composed by an ordered pair (b_i,b_j) of nodes[11] which specify that there is a directed edge between node b_i and b_j.

A **subgraph** of a directed graph, $G = (B,E)$, is a directed graph $G' = (B',E')$ in which $B' \subset B$. A **path** is a directed graph is a directed subgraph, P, of orderde nodes and edges obtained by succesive application of the successor[12] function.
As Figure 3.9 shows, one of the many supgraphs of G is G' = (B',E') in which
B'= {(2,3),(2,4),(3,5),(4,5),(5,2) }.
G' can be depicted by the subgraph shows by the Figure 3.10.
The basic blocks are connected with directed edges representing the jumps in the control flow.

The nodes of a CFG are basic blocks represented statements of the code that are sequentially executed after each other without any jumps. Branching can only exist at the end of blocks, after the execution of their last statement. The first step in the

[11] Nodes can not be distincts.

[12] A node, q, is said to be a *successor* of a node, p, if there exists one or more paths P=(p,...,q) and P= $(b_l,...,b_n)$ for which b_l = p and b_n = q.

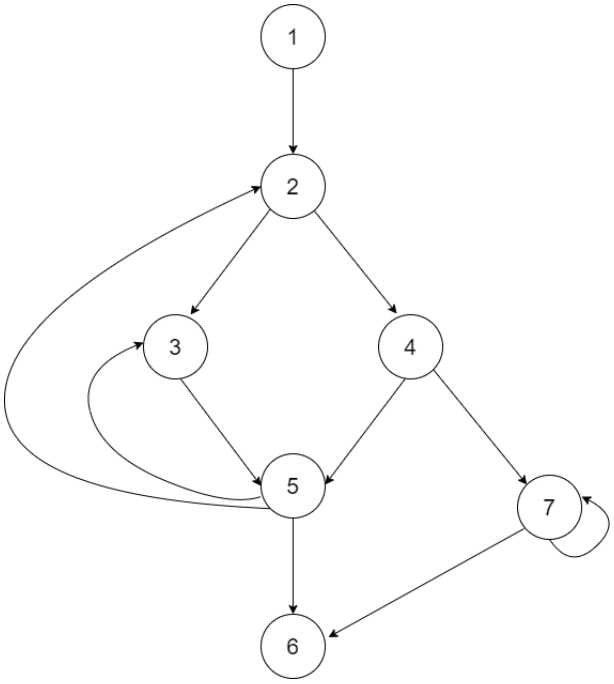

Figure 3.9: Example of a Directed Graph.

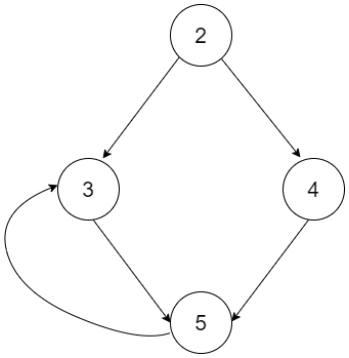

Figure 3.10: Example of a subgraph.

control flow creation is to establish the starting points of basic blocks, known as leaders. A leader can be:

- the first statement of a program,
- any statement that is target of a conditional or unconditional branch statement,
- any statement that immediately follows a method invocation statements.

In order to determine the blocks by enumerating their statements from one leader to another, it is important to know the sequence of statements and the leaders of basic blocks of a program. Once the compiler has constructed the Abstract Syntax Tree (AST) which implicitly describes the sequence of statements, it is possible to control graph flow with higher granularity and examine the evaluation order to the expressions. In general, the control flow information of methods, procedures or the subroutines of a program are separately represented. Owing to technical reasons, each of these has two type of basic blocks:

- the **entry block** shows the entering of a procedure,
- the **exit block** represents the returning from a called procedure.

The **call edges** outline the potential control flows among procedures. A connected control flow graph of a procedure with the call information produces the **interprocedural control flow graph** (ICFG) of a program. In a ICFG, call edges are represented as arrow-headed dashed lines between the call site, the Entry block of the called procedure, the Exit block and the return statement in the caller ICFG component.

A CFG is a useful methodology for code optimization techniques (e.g. unreachable code elimination, loop optimization or dead code elimination). Although the basic structure of a CFG is quite common, the methods constructing it for applications are

rather language dependent. Identifying control dependencies in special structures of the target language may result in special algorithms. Moreover, some program elements or applications may require minor modifications in the structure of the CFG (e.g. nodes like entry nodes).

3.3.2 Dynamic Analysis

Dynamic analysis is a method which checks a programming application while it is executing or in the run-time. Its behaviour is like an attacker who searches vulnerabilities in vulnerable code by feeding malicious code or anonymous input to the required function in a program. The most common methodologies of this type of analysis are:

- *Execution trace at run-time*
- *Transaction Graph Construction*
- *Symbolic Analysis*
- *Validation of true/false positives*

3.4 Static Analysis Tool engine

A static analysis tool for Ehtereum smart contracts should respect several properties:

- **correct level of abstraction**: the framework should have a correct level of abstraction in order to detect common patterns. Indeed, if the tool is focused too much on the detection of specific issue, it is difficult to append new detectors and improve the types of analyses;
- **robustness**: the framework should able to take into account real-world contracts without incurring any problems;
- **performance**: the analysis should be fast for any type of contracts.
- **accuracy**: the framework should enable to find potential pitfalls and its outcomes should have a low number of false positives.

A **compiler** is a software program that translates a high-level source language program into a form which can be execute on a computer. The compilation is a complicated process and early in the development of compilers, designers introduced IRs, intermediate representation also known as intermediate languages. IRs enable the management of the complexitiy of the compilation process. The use of an IR allows the compiler to be broken up into several phases and components, thus taking advantage of modularity.

In general, an IR is any kind of data structure that represent all the information of the program and its execution has to be accurately conducted. It is the common interface among the compiler components. Each compiler defines its own form and details of its IR because its usage is internal to it. The IR should be general because it enables the rapresentation of program traslated from different languages. The **semantic content of programming languages** is defined as *high* from the compiler developers. By contrast, the **semantic content of machine-executable code** is considered *low* because it has got the nececcary information from the original program to perform its correct execution. The compilation process involves the gradual lowering of the program

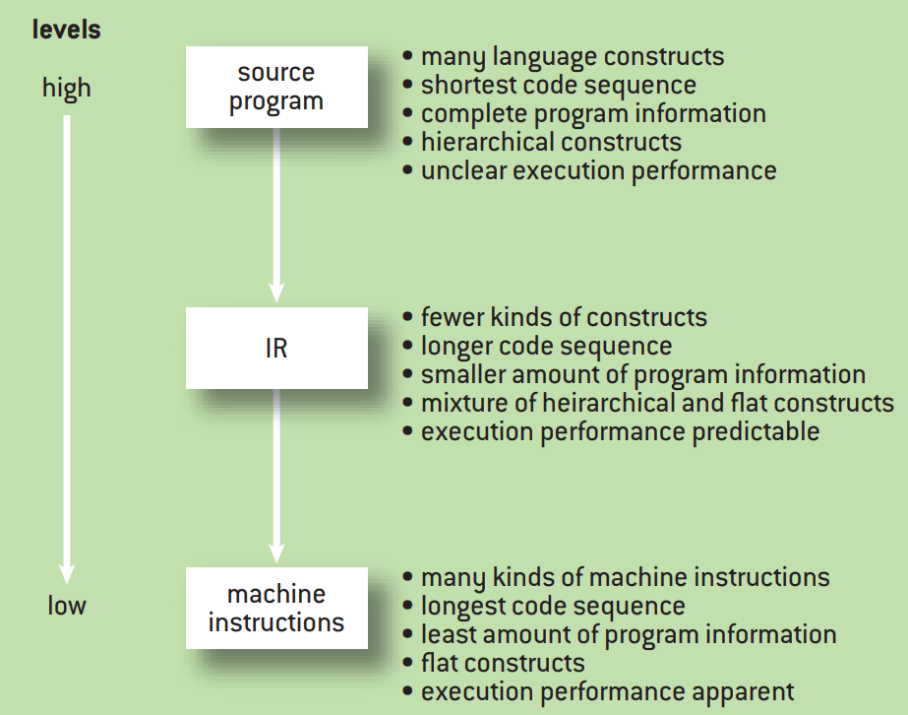

Figure 3.11: Smart Contract System.

rapresentation as it is evident from the Figure 3.11 from high-level human-readable constructs to low-level instructions. An IR has to be able to represent multiple languages, so it needs to be closer to the machine level in orther to constitute the behaviour of all the languages.

From the literature [14], it is clear that a well-designed IR should be translatable into several forms and its execution has to be done on different platforms. For instance, if the parget id a processo or CPU, the IR needs to be traslated into the assembly language of the specifc processor that is a one-to-one mapping to the processor's machine instructions. IR has to be at a higher level than typical machine instructions and not assume any peculiar machine characteristic.

3.4.0.1 Different IR forms

An IR reports the correct execution of the original program. Each instruction in an IR reproduces an simple operation. The constructs composing the IR are fewer than any typical programming language because it does not need to make programming use easier form developers. Indeed, imposing canonical forms in IRs decreases the amount of code patterns that the compiler manages in performing code generation and optimization. IR's instructions may map many-to-one to a machine instruction because one machine instruction may perform multiple operations.

The form of an IR can be categorized as **hierarchical** or **flat**. The former enables nested structures (i.e. program control flows like if-the-else or do-loops and arithmetic expressions), is closer to the typical programming language, and cand be internally represented in the form of trees without loos od accuracy. The latter is seen as the instructions of an abstract or virtual machine. Looking in detail, the instructions are executed sequentually and control flows are specified by branch or jump instructions.

Each instruction takes an amount of operands and produces a result. Both these form of IRs use the language of an abstract stack machine where each operand for arithmetic computation is specified by an instruction that pushes the operand onto the stack. Each arithmetic expression evaluation is done on the operands extracted from the top of the stack, and the result is polled back onto the stack.

There are additional information that have different aim than representing code execution. The compiler compiles the namespace in the original program into a collection of symbol names. Variables, functions, and type information are part of these symbol tables, and they encode information that controls the legality of certain optimizing transformations. They also gives information needed by several tools such (i.e. debuggers or program analyzers). The symbol tables is an additional structure to the IRs.

3.4.0.2 IRs for program delivery

With the spread of networked computers, it has become essential ensure the use of neutral processors and operating systems. The distribution and delivery process is easier with programs that can run on any machine. This is made possible using virtual machine execution model to decrease and fix the diversity of system hardware. Interpretive execution contributes to some loss of performance compared with compiled execution, and initially it made sense only for applications that are not computation-intensive.

The perfomances of machines rise significantly and the advantages of the write-once, run-anywhere approach outweigh potential loss in many applications.
As a result of this,the popularity of universally deployed programming languages (i.e. Java) grow sharply. As far as Java language is concerned, it defines the Java bytecode, which is a form of IR, as its distribution medium. Moreover, Java bytecode can be run on any platform where the JVM (Java virtual machine) software is installed. With the rise of the mobile Internet, applications are downloaded toever smaller devices to be run instantly. Since IRs have less storage than machine executables, they reduce network transmission overhead and allow hardware-independent program distribution.

3.4.0.3 Just-in-time compilation

Due to the widespread acceptance of virtual machine execution model, it has gained importance finding ways of speeding up the execution. One method is **J**ust-**I**n-**T**ime compilation, a dynamic approach, which improves the performance of interpreted programs by compiling them during execution into native code to increase the speed of the execution on the machine. Since compilation at runtime incurs overhead that slows down the program execution, it would be convenient to take the JIT route only if there is a high likelihood that the reduction in execution time more than offsets the additional compilation time. In addition, the dynamic compiler cannot spend too much time optimizing the code, as optimization incurs much greater overhead than translation to native code. Most JIT compilers compile only the code paths that are most frequently taken during execution.

Dynamic compilation has several posirtive aspects compared to static compilation. Firstly, dynamic compilation can optimize the generated code in a more effective way using realtime profiling data. Secondly, whether the program behavior changes during execution, the dynamic compiler can recompile to adjust the code to the new profile. Finally, with the prevalent use of shared libraries, dynamic compilation has become the

only safe means of performing whole program analysis and optimization, in which the aim of compilation spans both user and library code. From these aspects, it is evident that JIT compilation has become an indispensable part of the execution engine of many machine.

3.4.0.4 Standadizing IRs

There is a specific link between the compiler and its IR. Generally, IRs are translatable and it is possible to translate the IR of compiler A to that of compiler B, so compiler B can benefit from the work in compiler A. In the past two decades, the diffusion of open source software ever-incresed, and also compilers have been open sourced and exposed their IR definition to the world. Because IRs can solve the object-code compatibility issue among different processors by simplifying program delivery while enabling maximum compiled-code performance on each processor, standardizing on an IR would serve the computing industry well.

A generic IR resolves two different issues reported to the computing industry:

- *Software compatibility.* In general two pieces of software are incompatible if they are in different native code of different Instruction Set Architectures.

  Having the same ISAs is not a sufficient condition to make two softwares compatible as a matter of fact they may have been built using different Application Binary Interfaces or under different operating systems with different object file formats. It is evident that there are many different incompatible software ecosystems and the definition of a standard software distribution would be the best solution for the computing industry.The distribution medium can be based on the IR of an abstract machine and it will be rendered executable on a particular platform through AOT or JIT compilation. Computing devices supporting this standard will be able to run all software distributed in this form. This standardized software ecosystem will create a level playing field for manufacturers of different types of processors, thus encouraging innovation in hardware.

- *Compiler interoperability.*The algorithm that a compiler uses may fit for one program but not for another. As consequence of this idea, developing a compiler requires a huge effort and making further enhancements is inevitable. In this scenario, IR translation is a way of allowing compilers to work together. A standard IR allows to combine the strengths of the different compilers that use it which will no longer need to incorporate the full compilation functionalities. Using a standard IR has several positive aspects. First of all, it would lower the input barrier for compiler writers, because their projects could be conceived at smaller scales, enabling each compiler writer to focus on his specialties. Moreover, the usage of a standard IR would also make it easier to compare compilers because they would produce the same IR as output and its consequences could revolutionize compiler industry.

There are two visions for an IR standard. The former is focused on the computing industry, by contrast, the latter on the compiler industry. The first is centered on the virtual machine aspect, and the second on providing good support to several aspects of compilation. It is evident that an IR standard should take into account both goals.

3.4.0.5 IR design attributes

IR has several design attributes that belong to both visions taken into account or to one of the them. Five attributes are shared by both visions:

- *Completeness.* The IR must is a clean representation of all programming language constructs, concepts, and abstractions for detailed execution on computing devices.
- *Semantic gap.* The semantic gap between the source languages and the IR must be large enough that it is not possible to recover the original source program, in order to protect intellectual property rights. This implies the level of the IR must be low.
- *Hardware neutrality.* The IR must not have built-in assumptions of any special hardware characteristic. Any execution model apparent in the IR should be a reflection of the programming language and not the hardware platform. This peculiarity ensures it can be compiled to the widest range of machines, and implies that the level of the IR cannot be too low.
- *Manually programmable.* Programming in IRs is close to assembly programming. This enables programmers to hand-optimize the code.
- *Extensibility.* Programming languages are constantly evolving, there will be demands to support new programming paradigms. The IR definition should provide room for extensions without breaking compatibility with earlier versions.

From the compiler's perspective, there are three attributes that are important for the IR to be used as a program representation during compilation:

- *Simplicity.* The IR should have few constructs that enable the rapresentation of all computations translated from programming languages. Compilers often perform the canonicalization process that translates input program into canonical forms before performing several optimizations. Having the fewest possible ways of representing a computation is good for the compiler, because there are fewer code variations for the compiler to cover.
- *Program information.* The most complete program information exists in the source form in which the program was originally written, some of which is derived from programming language rules. Translation out of the programming language will contribute to information loss. A good IR should preserve any information in the source program that is helpful to compiler optimization.
- *Analysis information.* Program transformations and optimizations rely on additional information generated by the compiler's analysis of the program (i.e. data dependency, use-def, and alias analysis information). Encoding this kind of information in the IR enables it to be used by other compiler components, but it can be invalidated by program transformations. Indeed this information needs to be maintained throughout the compilation, which puts additional burdens on the transformation phases.

From this detailed analysis, it is clear that a standard IR that enables target-independent program binary distribution and that is internally used by all compilers may be idealistic, but it is a good solution that holds promise for the whole computing industry.

3.4.1 Tools

Testing is the most widely employed method to find vulnerabilities in real-word software programs. Several frameworks have been made to enable to analyze Ethereum smart contracts and detect potential pitfalls. They are based either on *static* or *dynamic* analysis.

We take into consideration static framework because the fit with the need of the research. In the following section, we introduce the **most significant frameworks** analyzed during the first step of the research. We aim to detect limited class of vulnerabilities, so, we take into account *static analysis* tools that enable the detection of specific gategories of pitfalls.

- **Oyente**[13] is a framework that uses symbolic execution methods. It takes as input bytecode of a smart contract taken from public repositories.

  It pinpoints the specific line of the smart contract source code which contains any security vulnerability.

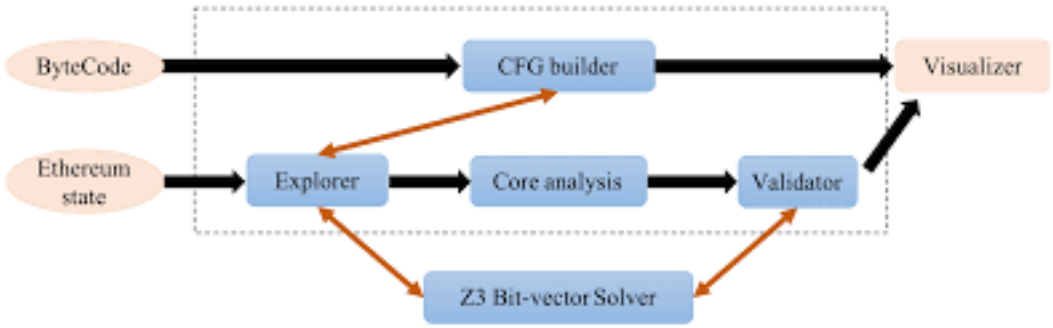

Figure 3.12: Oyente Architecture.

The Figure 3.12 explains its modular design composed by 4 structures:

1. `CFGBuilder` that sets up the shape of the **control flow graph** where each *node* is the execution of a basic block and the *edge* represents the jumps between the blocks[14].
2. `Explorer` enables the interpretation of loops that get a state to run and, then, sybolically executes a single instruction in the context of that state.

   In general, conditional jumps take boolean expressions and may alter the flow of the program counter so the framework introduces a solver module. **Z3**[15] determines if the branch condition is either true or false in that path; if so, the program counter is updated to the target address otherwise both branches are possible and more edges are added to the CFG. The solver eliminates infeasible resulting traces.

   The outcome of this stage is a set of symbolic traces associated with a path constraint.

[13] `https://github.com/melonproject/oyente`

[14] Some edges cannot be determined statically at this analysis step, so they are constructed on the fly during symbolic execution in the following steps.

[15] **Z3** is an efficient **S**atisfiability **M**odulo **T**heories Solver available from Microsoft Research. solver[16] in order to detect false positive. An SMT solver verifies the satisfiability of formulas in these theories. It enables applications such as extended static checking, predicate abstraction, test case generation, and bounded model checking.

3. `CoreAnalysis` has logic of the identified vulnerabilities.
   The module has sub-components that detect each group of vulnerabilities.
4. `Validator` is a still in progress filter and it aims to extract flase positives from the results.

The tool sets a timeout for the symbolic execution of 30 mins per contract and the timeout for each Z3 request is sets to 1 second.

- **Vandal**[16] is a security analysis framework[10] for Ethereum smart contracts. It converts low-level EVM bytecode to *semantic logic relations*. It performs a security analysis in a declarative way.

  Vandal facilitates a logic-driven static program analysis. As we can see from the Figure 3.13, Vandal follows a specific process. Firstly the `Extractor` converts the

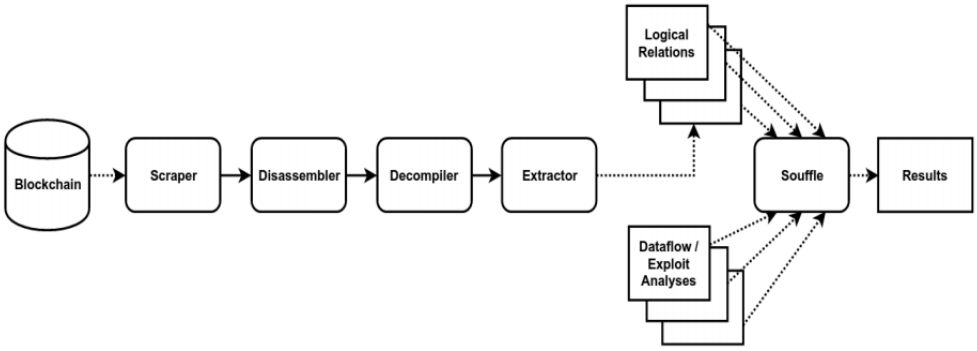

Figure 3.13: Vandal Pipeline.

program to a relational format, an Extensional Database (EDB) and it translates low-level bytecode to logic relations for the logic-driven security analysis. The `Extractor` consists of:

  – a **scraper** that retrieves EVM bytecode from the blockchain:
  – a **disassembler** that translates bytecode into opcodes (from bytecode to mnemonics);
  – a **decompiler** that converts low-level bytecode to register language.
    This stage is composed by two different phases: in the former, a block is symbolically analyzed whereas, in the latter it is incrementally built a CFG.
  – an **extractor** which transfers the register tranfer language to a logic semantic relations.

  Secondly, a Datalog engine computes the result of the program analysis from the EDB and the set of rules. It is a set of logic specifications for security analysis issues. It synthesizes highly preformant C++ code from logic specifications. This result is known as Intensional Database (IDB) and contains intermediate and final results of the analysis.

[16] `https://github.com/usyd-blockchain/vandal`

- **Securify**[17] is a scalable security verifier for Ethereum smart contracts.
  It is developed by the SRI Systems Lab (ETH Zurich). It takes as input the byte-

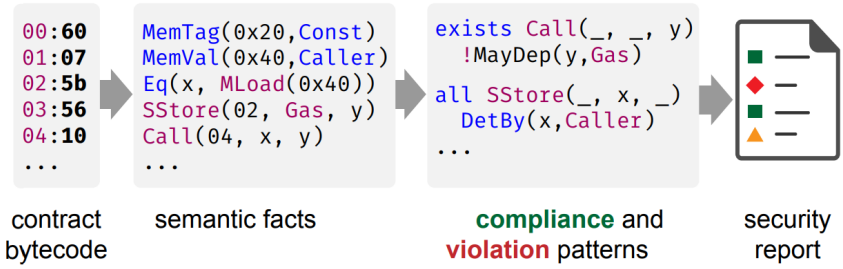

Figure 3.14: Securify Architecture.

code level. Once it has parsed and decompiled the EVM bytecode, it translates the code to *semantic facts* using static analysis. Following this step, it compares the facts with a list of patterns in order to detect common vulnerabilities.

- **SmartCheck**[18] is a static analysis tool[36] for Ethereum smart contracts implemented in Java. It takes in input Solidity source code and improves lexical and syntactical analysis. It uses ANTLR[19] generator and a custom Solidity grammar in order to produce an XML parse tree as intermediate rapresentation. It detects vulnerabilities using XPath queries on the IR. It mechanism follows several stages. First of all the whole source code of the contract is completely translated to the IR, then its elements can be searched with XPath matching. Line numbers are saved as XML attributes and this structure allows to discover pitfalls in the source code. In addition, IR attribute can be improved with further information once the new analysis methods are implemented.

  SmartCheck supports other smart contract languages including an ANTLR grammar and a pattern database while the IR-level algorithm is the same. The tool detect all the known code issues that are categorized as vulnerabilties of

  – high severity (i.e. re-entrancy or unchecked external call problems),
  – medium severity (i.e. costly loop, unsafe type inference, or balance equality scenario),
  – low severity (i.e. implicit visibility level, compiler version not fixed pitfalls).

  Smartcheck is tested on a set of 4,600 verified contracts from Ehterscan. It analyzes the dataset in 7644 seconds and the outcomes are optimal: 99.9% of contracts have issues and 3.2% of them have critical vulnerabilties.

- **Osiris**[20] combines *symbolic execution* and *taint analysis*, in order to accurately find integer bugs in Ethereum smart contract. From the Figure 3.15, it is evident

[17] https://securify.chainsecurity.com/
[18] https://github.com/smartdec/smartcheck
[19] **AN**other **T**ool for **L**anguage **R**ecognition[30] is a *public-domain parser generator* that combines the flexibility of hand-coded parsing with the convenience of a parser generator. It can generate parsers for many context-sensitive language. It enable the reading, processing, or translating of structured text or binary files.
[20] https://github.com/christoftorres/Osiris

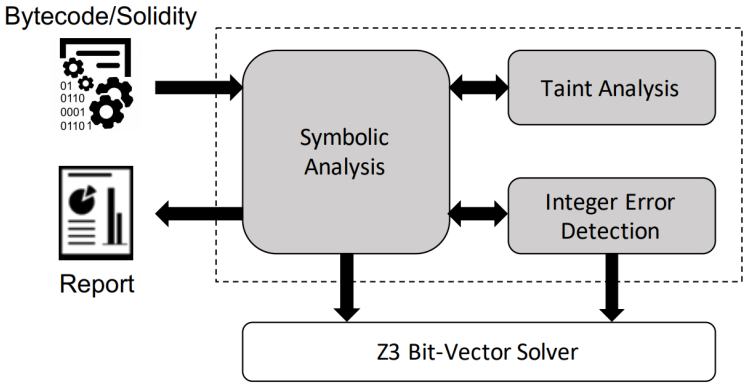

Figure 3.15: Osiris Architecture.

that Osiris engine is composed by three different components.

1. **Symbolic analysis** process starts by constructing a CFG from the bytecode. Osiris produces a visual representation of the CFG depicting the individual path conditions and highlighting the basic blocks that include **integer bugs**. Once the graph is costruited, the symbolic engine starts by executing the entry node of the CFG.

   It is an interpreter loop that gets a basic block as input and symbolically executes every single instruction within that basic block.

   The loop continues until all the basic blocks of the CFG have been executed or a timeout is reached. In the case of a branch, the symbolic execution engine queries **Z3** in order to determine easible pathes. If both paths are feasible, then the symbolic execution engine explores both paths in a Depth First Search (DFS) manner. Loops are terminated once they exceed a globally defined loop limit.

2. **Taint analysis** component is responsible for introducing, propagating and checking of taint across stack, memory and storage.
   It checks if the executed instruction is part of the list of defined sources.

3. **Integer error detection** checks whether an integer bug is possible within the executed instruction.

- **Gasper**[13] is a static tool and it examines *gas costly pattern* from the existing smart contract, taking as input the bytecode.

  It runs **symbolic execution** on bytecode to find all the reachable code blocks in a specific contract.

  It uses the disassembled outcomes of the pre-processing step and constructs the **control flow graph** (CFG).

  After this stage, it starts the symbolic execution from the root node of the control flow graph and analyses the whole structure. The tool uses Z3 and classifies the patterns into two different categories:

  – *useless code related pattern* that introduces additional cost because of the increased size of bytecode during the deployment and the removable bytecode in runtime;

– *loop related pattern* which involves the use of expensive operations in loop structures.

The Gasper process expects to add the gas-costly patterns to contracts source code, and then checks whether the patterns are converted into gas-efficient ones in the generated bytecode.

- **MadMax**[21] is a static program analysis framework[20] that detects gas-focused vulnerabilities in smart contracts.

  The tool analyzes the whole blockchain in ten hours. It is composed by several

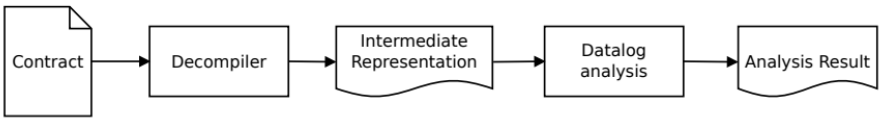

Figure 3.16: MadMax analysis pipeline.

stages as the Figure 3.16 illustrates. First of all, a control-flow-analysis-based **decompiler** which converts from low-level EVM bytecode to a structured intermediate language. MadMax uses the *Vandal* decompiler which accepts EVM bytecode as input and produces a standard structured intermediate CFG rapresentation.

Moreover, a logic-based analysis specification produces a high-level program model.

The procedure consists of three analysis layers that progressively infer higher-level concepts about the analyzed smart contract.

  – **STEP 1:** From the 3-address-code representation, structura such as loops, inductio variables and data flow are recognized;
  – **STEP 2:** An examination of memory and dynamic data structura is performed. During this step, there is the modelling of the EVM dynamic data description;
  – **STEP 3:** Concepts concerning the analysis of gas-focused vulnerabilities (i.e. loop with unbounded mass storage) are inferred.

  The decompiler produces 3-address relations in a normalized form that is given as input to the Datalog-based database.
  The tool uses *Soufflé*[24] as Datalog engine that compiles its Datalog input program into a C++ application.

  The whole approach utilizes two different techniques: the former is teh **abstract-interpretation-based** to pursue the decompilation and the latter is the **declarative program** for the higher-level analysis.

- **GASOL**[5] is a *custom Ethereum Virtual Machine* that facilitates symbolic execution of contract bytecode. It is an optimization detector which infers the maximal number of iterations for loops and generates accurate gas bound which are valid for any possible execution of the function. The tool has a specific flow: once extracted the CFGs, it decompiles the graphs into a high-level representation in order to have upper bounds that are produced by analyzers and solvers. Its analyser component is **GASTAP** that is *cost model* to compute the overall

[21] `https://github.com/nevillegrech/MadMax`

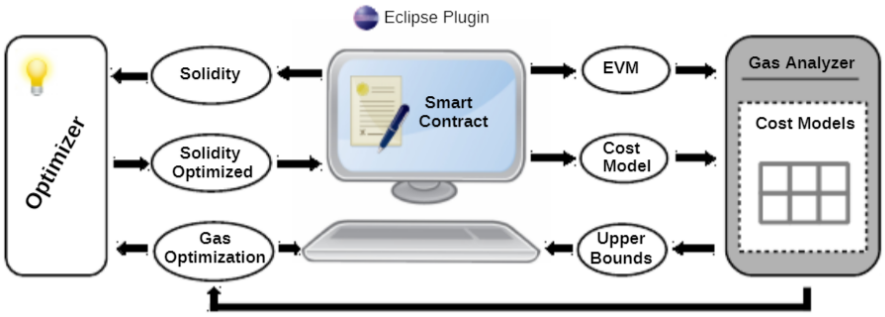

Figure 3.17: GASOL components.

gas consumption of the function. GASOL is used as a *gas analyzer* in order to estimate the gas consumption according to a gas model or it serves as *complexity analyzer* and estimates the number of bytecode instructions executed.
The tool aims to reduce of the gas consumption associated to the usage of storage. It replaces the multiple access to the same storage data within a fragment of code by one accesses to such memory position and a final update to the storage if it is necessary.

- **Slither**[22] is a static analysis tool[18] which extracts information from Ethereum smart contracts. It converts Solidity smart contracts into the *SlithIR* intermediate representation that uses *Static Single Assignment* (SSA) form and a produces less complex instruction sets.
  Slither aims to define *automated detection* of vulnerabilities, provides a *code optimization procedure*, and enables *code review* mechanisms.

3.4.2 The efficient of smart contract analysis tools

According to the research provided by Ghaleb A. and Pattabiraman K.[19], over the last tem years, a number of static analysis tools have been developed for finding security bugs in smart contracts.

However,it is not still carried out a *systematic method to evaluate those tools regarding their efficacy in finding security bugs.*

The study illustrates that tools can have false-positive and false-negative outcomes and how is important to analyse them in order to improve static analysis methodologies. Moreover, several studies[35] of software defects have noticed that many of the vulnerabilities can be detected by static analysis tools in theory, but are not pratically detected due to constraints of the tools.

The Canadian research proposes *SolidiFI*[23], an automated and systematic approach which avaluate smart contracts' static analysis tools. Their aim undetected bugs and it studies false-positives of the tools. They take into account six freely available static analysis which operate on smart contracts written in Solidity, Oyente, Securify, Mythril, SmartCheck, Manticore, and Slither.

The experiment involve the use of a dataset of fifty verified smart contracts, chosen from the public repository Etherscan. Smart contracts are chosen taking into consider-

[22] https://github.com/crytic/slither
[23] https://github.com/DependableSystemsLab/SolidiFI

ation three specific criteria:

- the **code size**: there are selected contracts with different size in order to have a realistic scenario of the contracts founded in Etherscan;
- the **compatibility with Solidity version 0.5.12**: it is evident that only 321 out of 500 verified smart contracts in the public repository supported Solidity 0.5x and higher;
- the **range of functionality**: there are selected games contracts or wallet ones.

From this huge dataset, the researchers have selected 50 contracts in order to cut the time and the effert needed to analyze them. They inject bugs of seven different bug types belonging to the detection scope of the chosen tools. The process of injection bug takes as input the Abstract Syntax Tree (AST) of the smart contract. As the Figure 3.18 illustrate, SolidiFI follows three stages. First of all, there is the identification of the

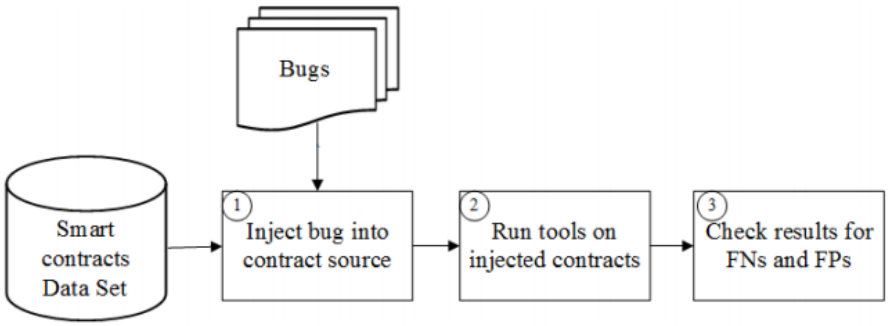

Figure 3.18: SolidiFI Workflow.

potential locations for the introduction of bugs. Looking in detail, the AST is passed to a *Bug Location Identifier* which, for a specific vulnerability, identifies all possible injection locations in the target contract. Following this, one bug type at a time is injected into all decided locations in order to generate buggy contracts. In order to get relevant results, the introduced bugs are as different as possible and various set of code snippets with several data inputs and function calls are prepared to get this aim. Subsequently, the number of the inserted bugs that were not detected by each tools were stored. In order to verify the bug, it is possible to use the chosen tools and analyze their results. The bug is considered correctly detected by the tool if and only if it catches both the line of code of the insert bug and the bug type. As we con see from the Figure 3.19 [24], in many cases, the tool identifies the line of code whereas it misidentifies its category.

So, it is evident from the analysis that a significant number of **false negatives** occurs for all the evaluated tools. In addition, the outcomes highlight an important number of undetected bugs in the tools. By contrast, **false positives** are verified when a tool report detects bug which there is not. Discovering this kind of issue is really challenging because it is not possible to assume that the dataset of contracts has no bugs. The experiment carries out an analysis of false positives. Firstly, bugs which are

[24]Numbers within brackets are bugs with incorrect line of code or unreported.

Security bug	Injected bugs	Oyente	Securify	Mythril	SmartCheck	Manticore	Slither
Re-entrancy	1343	1008 (844)	232 (232)	1085 (805)	1343 (106)	1250 (1108)	✓
Timestamp dep	1381	1381 (886)	NA	810 (810)	902 (341)	NA	537 (1)
Unchecked-send	1266	NA	499 (449)	389 (389)	NA	NA	NA
Unhandled exp	1374	1052 (918)	673 (571)	756 (756)	1325 (1170)	NA	457 (128)
TOD	1336	1199 (1199)	263 (263)	NA	NA	NA	NA
Integer overflow	1333	898 (898)	NA	1069 (932)	1072 (1072)	1196 (1127)	NA
tx.origin	1336	NA	NA	445 (445)	1239 (1120)	NA	✓

Figure 3.19: False negative for each tool.

not reported by the vast majority of tools are manually analyzed. There are a number of benefits and drawbacks in relation to this approach. The main negative aspect of it is the possible underestimation of the number of the false positives. Secondly, an significant number of bugs is manually insoected in order to decide their false positivity. Following this, 20 bugs belonging to each bug category are randomly selected and inspected.

Bug Type	Threshold	Oyente			Securify			Mythril			SmartCheck			Manticore			Slither		
=		Reported	FIL	FP	Reported	FIL	FP	Reported	FIL	FP	Reported	FIL	FP	Reported	FIL	FP	Reported	FIL	FP
Re-entrancy	4	0	0	-	12	12	12	54	54	43	0	0	-	6	6	6	79	79	71
Timestamp dep	3	0	0	-				12	12	0	0	0	-				12	12	0
Unchecked send	2				7	4	4	14	3	3									
Unhandled exp	3	10	10	10	0	0	-	0	0	-	6	6	6				0	0	-
TOD	2	32	24	24	121	97	97												
Over/under flow	3	947	943	801				17	3	3	3	2	2	9	9	9			
Use of tx.origin	2							0	0	-	3	1	0				4	2	0
Miscellaneous		0			318			144			1520			169			1807		

Figure 3.20: False positive by each tool.

The results presented in Figure3.20, illustrate that the tools with low numbers of flase negatives have high false positives. From the table, the column belonging to *Slither* is highlighted. Its results show that it is the only tool that successfully detected all the reentrancy bugs and it reported an important amount of false positives. This trend casts doubt on whether the high detection rate was "*simply a result of overzealuosly reporting bugs by the tool.*" In conclusion, *SolidiFI* identifies meaningful gaps in static analysis tools for smart contracts. The reseach proves that static analysis tools should be optimised in order to able to **detect bugs** and **maintain low false positive rates**.

Optimization Detectors

This chapter is the core of the thesis where we take into account the specific security analysis method improved in the chosen tool and we select a restrict list of vulnerabilities related to optimization gas-patterns Here, we present a real case of smart contract testing study as an example of the testing step of the research. The analysis of real contracts and the scanning of those outcomes aim to point out the need of the diffusion of common guidelines which should be followed during the development of contracts.

In Section 4.1, we focus our attention on a static analysis framework, **Slither**. Once we got all the information about its scopes, we test it and we analyzed those outcomes.

In Section 4.2, we introduce the concept of **gas patterns** and their possible *optimizations*.

In Section 4.3, we explain the stages related to the **testing phase** of the framework. This process has incurred into several steps to select the Solidity smart contract analyzed in the Section 4.4. We try out all Slither developments and then we introduce an *ad-hoc detector* to detect unused storage variables in the line code of the targeted contract.

4.1 Slither

Figure 4.1: Smart Contract System.

Slither is a static analysis framework that gives information about Ethereum smart contract. The tool is used for four different scopes:

- **automated detection of vulnerabilities**: the framework catches out 44 type of vulnerabilities as Table 4.1 and Table 4.2 shown. Each vulnerability is categorised in term of *imapct* and *confidence*
- **automated code optimizations detection**:the framework points out code optimizations;

- **code analyzing**: the framework allows to print contracts in order to analyze the codebase;
- **assited code review**: through its API, user can interact with the framework.

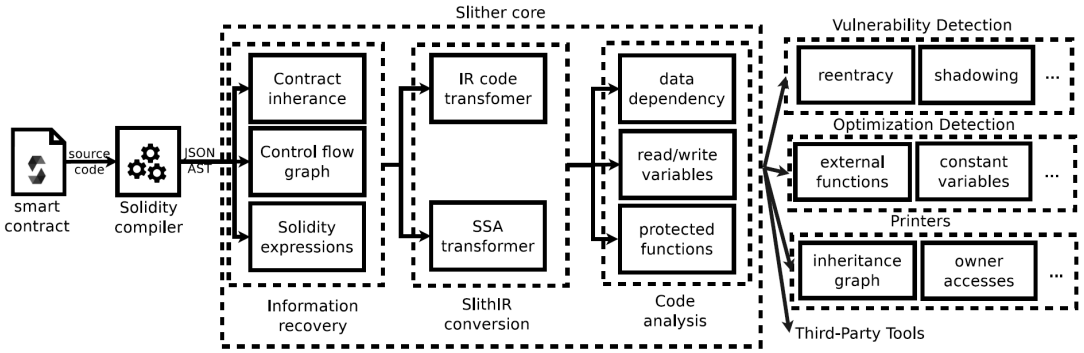

Figure 4.2: Smart Contract System.

The Figure 4.2 illustrates all the stages that composed Slither architecture. It analyses contracts using static analysis in a modular procedure. The framework takes as input the Solidity **Abstract Syntax Tree** (AST) which is generated by the Solidity compiler from the contract source code.

First of all, Slither gets import information like the contract's inheritance graph, the Control Flow Graph and the list of expressions. For each contract, function or node of the control flow graph,it is possible to recover read or written variables and filter by local or state variables.

Following this, it transforms the code of the contract to SlithIR that is an internal representation language. SlithIR uses Static Single Assessment (SSA) to semplify the computation of several code analyses. The last step concerns of the computation of a set of pre-defined analyses that give information to the other modules.

Slither translates Solidity into the SlithIR intermediate representation in order to obtain a more precise analysis via a simple API.

The framework includes *printers* that enable users to understans the structure of the contract. Using the added `-- print` option to a general run it is possible to:

- export several kind of **graph-based rapresentation** such as inheritance graph, control flow graph and call graph of each contract;
- download a **human-readable summary** of the contracts that outlines the number of issues found and information concerning the quality of the code;
- a list of the **autorization accesses** and the **variables** that can be changed by the contract's owner.

Slither provides a specific module that checks for ERC's conformance of the analyzed contract which provides guidelines for the proper management of the Ethereum blockchain.

Using the command `slither-check-erc` it is possible to explore if:

Detector	*What it Detects*	*Impact*	*Confidence*
`name-reused`	*Contract's name reused*	High	High
`rtlo`	*Right-To-Left Override control character is used*	High	High
`shadowing-state`	*State variable shadowing*	High	High
`suicidal`	*Function allowing anyone to destrict the contract*	High	High
`uninitialized-state`	*Uninitialized state variables*	High	High
`uninitialized-storage`	*Uninitialized storage variables*	High	High
`arbitrary-send`	*Functions that send ether to arbitrary destinations*	High	Medium
`controlled-delegatecall`	*Controlled delegatecall destination*	High	Medium
`reentrancy-eth`	*Reentrancy vulnerabilties (theft of ethers)*	High	Medium
`erc20-interface`	*Incorrect ERC20 interfaces*	Medium	High
`erc721-interface`	*Incorrect ERC721 interfaces*	Medium	High
`incorrect-equality`	*Dangerous strict equalities*	Medium	High
`locked-ether`	*Contracts that lock ether*	Medium	High
`shadowing-abstract`	*State variables shadowing from abstract contracts*	Medium	High
`tautology`	*Tautology or contradiction*	Medium	High
`boolean-cst`	*Misuse of boolean constant*	Medium	Medium
`constant-function-asm`	*Constant functions using assembly code*	Medium	Medium
`constant-function-state`	*Constant functions changing the state*	Medium	Medium
`divide-before-multiply`	*Imprecise arithmetic operations order*	Medium	Medium
`reentrancy-no-eth`	*Reentrancy vulnerabilities (no theft of ethers)*	Medium	Medium
`tx-origin`	*Dangerous usage of* `tx.origin`	Medium	Medium
`unchecked-lowlevel`	*Unchecked low-level calls*	Medium	Medium
`unchecked-send`	*Unchecked send*	Medium	Medium
`uninitialized-local`	*Uninitialized local variables*	Medium	Medium
`unused-return`	*Unused return values*	Medium	Medium
`shadowing-builtin`	*Built-in symbol shadowing*	Low	High
`shadowing-local`	*Local variables shadowing*	Low	High
`void-cst`	*Constructor called not implemented*	Low	High
`calls-loop`	*Multiple calls in a loop*	Low	Medium
`reentrancy-benign`	*Benign reentrancy vulnerabilties*	Low	Medium
`reentrancy-events`	*Reentrancy vulnerabilities leading to out-of-order Events*	Low	Medium
`timestamp`	*Dangerous usage of* `block.timestamp`	Low	Medium

Table 4.1: List of all implemented detectors

Detector	*What it Detects*	*Impact*	*Confidence*
`assembly`	*Assembly usage*	Inf.	High
`boolean-equal`	*Comparison to boolean constant*	Inf.	High
`deprecated-standards`	*Deprecated Solidity Standards*	Inf.	High
`erc20-indexed`	*Un-indexed ERC20 event parameters*	Inf.	High
`low-level-calls`	*Low level calls*	Inf.	High
`naming-convention`	*Conformace to Solidity naming conventions*	Inf.	High
`pragma`	*If different pragma directives are used*	Inf.	High
`solc-version`	*tIncorrect Solidity version*	Inf.	High
`unused-state`	*Unused state variables*	Inf.	High
`reentrancy-unlimited-gas`	*Reentrancy vulnerabilties through send and transfer*	Inf.	Medium
`too-many-digits`	*Conformance to numeric notation best practices*	Inf.	Medium
`constable-states`	*State variables that could be declared constant*	Opt.	High
`external-function`	*Public function that could be declared as external*	Opt.	High

Table 4.2: List of all Informational and Optimizational detectors

- all the functions are present;
- all the events are present;
- retuns function are correctly write;
- functions' visibility are correctly implemented;
- events' parameters are accurately indexed;
- the functions emits the events;
- derived contracts meet the standard.

Slither ERC's conformance module supports:`ERC-20`, `ERC-223`, `ERC-777`, `ERC-721`, `ERC-165`, `ERC-1820`.

Ethereum is based on the use of tokens which can be bought, sold, or traded. Ethereum tokens are smart contracts that make use of the Ethereum blockchain. **ERC-20** [37] is the most significant Ethereum tokens and has become a technical standard. It is used on Ethereum blockchain for token implementation and provides several rule that Ethereum-based tokens must respect. The `ERC20+` contract is a concise declaration which is in line with the ERC20 standard.

Looking in detail, ERC-20 standard draws up a set of six different functions.

```
1 contract ERC20 {
2     function totalSupply() constant returns (uint theTotalSupply);
3     function balanceOf(address _owner) constant returns (uint balance);
```

```
marghe_ren@marghren-VirtualBox:~/slither$ slither-check-erc  tests/contracts/unca.sol StandardToken
# Check StandardToken

## Check functions
[✓] totalSupply() is present
        [✓] totalSupply() -> () (correct return value)
        [✓] totalSupply() is view
[✓] balanceOf(address) is present
        [✓] balanceOf(address) -> () (correct return value)
        [✓] balanceOf(address) is view
[✓] transfer(address,uint256) is present
        [✓] transfer(address,uint256) -> () (correct return value)
        [✓] Transfer(address,address,uint256) is emitted
[✓] transferFrom(address,address,uint256) is present
        [✓] transferFrom(address,address,uint256) -> () (correct return value)
        [✓] Transfer(address,address,uint256) is emitted
[✓] approve(address,uint256) is present
        [✓] approve(address,uint256) -> () (correct return value)
        [✓] Approval(address,address,uint256) is emitted
[✓] allowance(address,address) is present
        [✓] allowance(address,address) -> () (correct return value)
        [✓] allowance(address,address) is view
[ ] name() is missing (optional)
[ ] symbol() is missing (optional)
[ ] decimals() is missing (optional)

## Check events
[✓] Transfer(address,address,uint256) is present
        [✓] parameter 0 is indexed
        [✓] parameter 1 is indexed
[✓] Approval(address,address,uint256) is present
        [✓] parameter 0 is indexed
        [✓] parameter 1 is indexed
```

Figure 4.3: Screen of the ERC-20's conformance on StandardToken contract in unica.sol file.

Figure 4.4: ERC Logo.

```
4       function transfer(address _to, uint _value) returns (bool success);
5       function transferFrom(address _from, address _to, uint _value)
6       returns (bool success);
7       function approve(address _spender, uint _value) returns (bool
            success);
8       function allowance(address _owner, address _spender) constant
9       returns (uint remaining);
10      event Approval(address indexed tokenOwner, address indexed spender,
11      uint tokens);
12      event Transfer(address indexed from, address indexed to,
13      uint tokens);
14  }
```

According to the summary provided by the authors, the standard "*gives basic functionality to transfer tokens, allows tokens to be approved so they can be spent by another on-chain third party.*"
It defines the following functions:

- `totalSupply`: this function enables an instance of the contract to calculate and return the amount of the token that exists;
- `balanceOf`: this function takes as parameter the address which should be public;
- `transfer`: this function enables the owner of the contract to send a certain amount

of token to another address as a cryptocurrency tranasaction

- `transferFrom`: this function allows a smart contract to automatize the transfer process and send a given quantity of the token on behalf of the owner.
- `approve`: the owner calling this function authorizes (or approves) the address to withdraw instances of token from the owner's address;
- `allowance`: the owner calling this function return the amount that the `_spender` is still allowed to withdraw from the owner;
- it has several optional fields in order to improve usabilitiy[1] like:
  - *token name*:

    ```
    function name() public view returns (string)
    ```

    This function returns the name of the token.
  - *token symbol*:

    ```
    function symbol() public view returns (string)
    ```

    This function returns the symbol of the token.
  - *number of decimals*:

    ```
    function decimals() public view returns (uint8)
    ```

    This function returns the number of decimals the token uses.

In addition, there are two specific defined events:

- `Approval`

```
event Approval(address indexed tokenOwner, address indexed
    spender,
uint tokens);
```

- `Transfer`

```
event Transfer(address indexed from, address indexed to, uint
   tokens);
```

that can be invoked or emitted when a user is granted rights to withdraw tokens from an account, and then the tokens are actually transferred. In conclusion, ERC-20 standard significantly reduces the effort needed to create and issue a digital token. The number of the ERC-20 token contracts has experienced an exponential growth since 2015. In 2018, there were around 40,000 ERC-20 contracts on the Ethereum network and now they are more than 160,000.

4.1.1 SlithIR

SlithIR is the *hybrid intermediate representation* used by Slither that represents Solidity code. SlithIR converts each node of the control graph flow that holds a Solidity expression to a set of instructions.

This convertion makes easier the analyses and keeps critical semantic information from the Solidity source code.

[1] They should not be in interface and other contracts.

- `StateVariable`
- `LocalVariable`
- `Constant`
- `Solidity Variable`
- `TupleVariable`
- SlithIR variables:
  - `TemporaryVariable`
  - `ReferenceVariable`: Solidity enables the manipulation of mapping and arrays which accessed through deferencing. So, SlithIR uses this variable in order to *store the result of deferencing*. The index operator allows deferencing of a variable:
    - `REF` ← `Variable [Index]`

    The member operator allows to access to a structure:
    - `REF` ← `Variable` · `Member`

`LV` and `RV` represent a variable which is assigned (left-value) and a variable which is read (right-value). In addition, a variable can be a Solidity one or a temporary one created bt SlithIR.

Moreover, Slither provides an in-depth information about calls and has nine call instructions[2]:

- `LV = L_CALL Destination Function [ARG..]`, *low-level Solidity call*
- `LV = H_CALL Destination Function [ARG..]`, *high-level Solidity call*
- `LV = LIB_CALL Destination Function [ARG..]`, *library Solidity call*
- `LV = S_CALL Function [ARG..]`, *call to a inbuilt-Solidity function*
- `LV = I_CALL Function [ARG..]`,*call to an internal function*
- `LV = DYN_CALL Event [ARG..]`,*call to an internal dynamic function*
- `LV = E_CALL Event [ARG..]`,*event call*
- `LV = Send Destination`, *Solidity send*
- `Transfer Destination`, *Solidity transfer*

Furthermore, Slither includes additional instrictions such as `PUSH` for array manipulation, `CONVERT` for type conversion, and other operators to manipulate tuples.

Slither uses the **SSA representation** in order to compute data dependencies of all variables. The dependencies are first analyzed in the context of each function. Following this, a fixpoint is computed across all the functions of the contract that determines whether there is a dependency in a multi-transaction context. The framework classifies some variables as **tainted** and this means that they depend on a user-controlled variable, and can be manipulated by the user. Finally, the data dependency takes into consideration the protected function heuristic.

[2]Some calls can have additional arguments, for instance, `H_CALL`,`L_CALL`, `Send` and `Transfer` can have *value* associated to the call, representing the quantity of Ether for the transaction.

4.1.2 SSA

Static Single Assignment [33], SSA , is a suitable intermediate rapresentation that enables to optimize and simplify the input program. The main quality of SSA form is thath there is only on assignment to each variable in the whole program text so it is easy to reason about variable, indeed if two variables have the same name, they also hold the same value.

To obtain a SSA form, in every assignment, the variable on the left-hand side has a unique name[3] and all its renamed are changed according to this. More complicated programs have branches and *join nodes*. As Figure 4.5 illustrates, several values of a

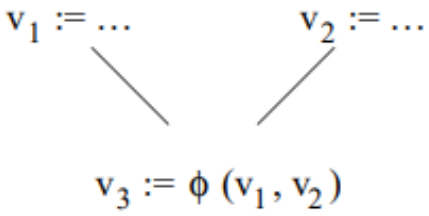

Figure 4.5: Values v_1 and v_2 mergetd into a unique v_3.

variable may reach the node via different branches, so these values have to be merged into one that reaches different uses of the variable which is one single assignment. For this purpose, assignments are generated with so-called ϕ-funcions on the rigth-hand side. Their operands are related to the number of branches that point the join node. In general, the meaning of a ϕ-funcion is: if control reaches the join node via the i-th branch, the value of the function is its i-th operand. Figure 4.6 show the *control*

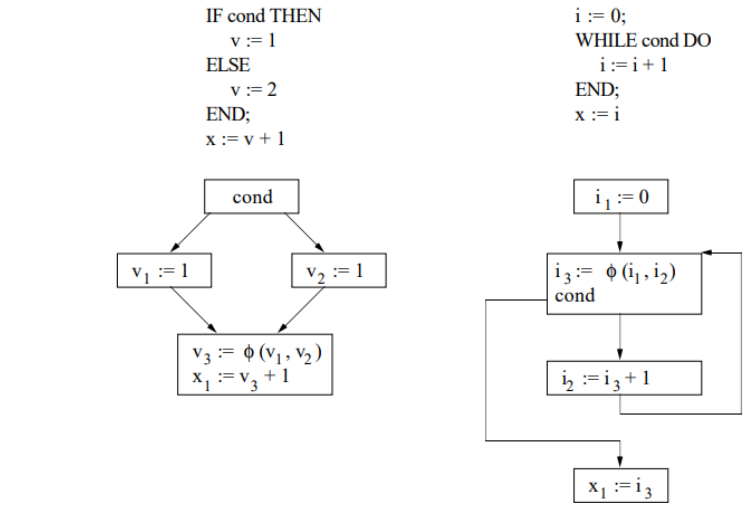

Figure 4.6: Control flow graph of an IF and a WHILE statement with instruction in SSA form.

graph flow for IF and WHILE statements with instructions in SSA form. The node of the graphs are *basic blocks* except possibly at the end of the sequence (for instance

[3]Subscripts are used in order to make unique variable names.

instruction sequences with a single entry adn no branch instruction).

The single-assignment property simplifies reasoning about variables. Because every assignment creates a new value name it cannot *kill* expressions previously computed from other values. In addition, single-assignment property is also helpful in instruction scheduling as it leaves only the essential data dependences in the code and avoids output dependencies.

From the literature there were sereval researches that aimed to generate SSA form and each of them use a peculiar approach. Looking in detail, Cytron et al. [15] have presented an efficient algorithm for generating SSA form from the instructions of a control flow graph and its dominator tree. The algorithm computes the dominance frontiers in the graph, that are the nodes where ϕ-functions have to be placed. It is the most efficient algorithm for a general flow graph but the instructions of the graph has to pass through stages.

By contrast, Rosen et al.[32] have improved a single-pass algorithm for generating SSA form. This algorithm requires a topologically sorted control flow graph and generates many unnecessary ϕ-assignments.

Another method was described by Johnson and Pingali[23] and it derived a SSA form computing the dependence flow graph of a program. Morever, they identified single-entry/single-exit regions to place merge nodes which are similar to ϕ-assignments.

4.2 Gas-costly patterns

Smart contracts run on the machines of miners who collect Ethers by contributing their resources. The creators and the users of smart contracts will be charged quantity of gas for obtaining the miners' power resources. The charge of a transaction is the moltiplication of the gas of the certain transaction and the Ether per unit which represents the price of gas.

Furthermore, also the deployment of a contract requires the consumption of gas related to the size of contracts in bytecodes. As result of the searchers done related to the scope, it is fundamental to make analysis concerning the achievement of optimized contracts.

From the literature[13], we have founded out that "*under-optimized smart contracts cost more gas than necessary, and therefore the creators or users will be overcharged.*" In order to overcome these issue, the solution is to observe **gas-efficient programming patterns**. It is difficult to reach this aim because of the lack of common guideline used by the whole Ethereum community.

Moreover, this research field is complexed as a matter of fact the identification of gas-costly bytecode and the replacement of gas-efficient one required the knowledge of several notions such as the EVM's instruction, the amount of gas needed for each operations or the quantity of data read/written.

This stage of the research was composed by two main stages. The former has concerned the identification, the explaination and the optimization of several gas-costly patterns related to specific aspect of Ethereum environment. The latter aimed to analyze space saving pattern category in order to optimize this aspect of smart contracts behaviour.

4.2.1 External Transactions

This class includes patterns associated with the creation of contracts and the sending transactiona from external address.

- **Proxy**: smart contracts are immutable. If a contract must be changed due to a detected bug or a needed extension, you a new contract must be deployed, and also update all contracts making direct calls to the old one. This approach is very expensive and a possible solution is the use of a proxy delegate patterns. It is a ste of smart contracts that interact one another in order to facilitate upgrading of smart contracts. The Proxy stores the addresses of referred contracts, in their state variables. The cost are decreased because only the references to the new smart contract must be updated.
- **Data Contract**: if a smart contract holds a significant amount of data must be updated and its data must be copied to the newly deployed contract and this action will consume a lot of gas. A possible solution to overcome this excessive cost is to keep data in a separate smart contract that is accessed by one or more smart contract, using the data and holding the processing logic. If the logic must be updated, the data remain in the Data Contract.
- **Event Log**: events could maintain important information about the system, which must be later used by the external system interacting with the blockchain. Storing this information in the blockchain can be very expensive, if the number of events is huge. If past events data are needed by the external system, but not by smart contracts, the external system should directly have access to the Event Log in the blockchain[4].

4.2.1.1 Storage

This category involves patterns related to the usage of Storing for saving permanent data.

- **Limit Storage**: storage is by far the most expensive type of memory, so its usage should be minimized. The data stored in the blockchain should be limited, for instance data should be saved in memory for non-permanent data. Furthermore, storage limit should be resticted: when executing functions, the intermediate results should be saved in memory or stack and update the storage only at the end of all computations.
- **Packing Variables**: in Ethereum, the minimum unit of memory is a slot of 256 bits. Even if the slot is not completed, it should be paid. So the possible solution is to pack variables. When declaring storage variables, the packable ones, with the same data type, should be declared consecutively. In this way, the packing is done automatically by the Solidity compiler[5].
- **Packing Booleans**: in Solidity, boolean variables should be stored `uint256` variables. In order to achieve this purpose, it is necessary to create functions that pack and unpacke the booleans into and from a single variable. The cost of running these functions is cheaper than the cost of extra Storage.

[4]The Event Log is not accessible by smart contracts, and that if the event happened far in time, the time to retrieve it may be long.

[5]This mechanism does not work for Memory location data because those variable cannot be packed.

4.2.2 Useless Code Related Patterns

This class analyzes the introduction of additional cost owing to the increased size of bytecode during the deployment and the removable bytecode in runtime.

- **Dead Code**: the removal of predicates that false under all circumstances. Solidity does not remove them during the generating bytecode phase, so, they represent a wasted cost.
- **Opaque Predicate**: the result of an opaque predicate is true or fasle without execution so, it should be removed for saving gas.

4.2.3 Loop Related Patterns

This type shows patterns that involve the use of expensive operations in the loop.

- **Expensive Operation**: they may execute multiple times in one invocation. A optimized solution could be the shift of the expensive operations out of the loop.
- **Constant result**: the result of a loop may be a constant that can be inferred in compilation.
- **Loop fusion**: the combination of several loops into one enables the reduction of the size of bytecode. This solution reduces the amount of operations, such as conditional jumps and comparison, etc., at the entry points of loops.
- **Repeated computations**: in some cases, there may be expressions that produce the same outcome in each iteration of a loop. The gas can be saved by computing the result once and then reusing the value instead of recomputing it in subsequent iterations, especially, for the expressions involving expensive operands.
- **Comparison with unilateral outcome**: a comparison is executed in each iteration of a loop but the final result of the comparison is the same even if it cannot be determined in compilation.

4.2.4 Saving Space

This group takes into account patterns concerning the optimization of both Memory and Storage space.

- **Uint* vs Uint256**: the EVM run on 256 bits at a time, every uint* integers [6] will first be converted to uint256 and this mechanism costs extra gas. So, it is more efficient to use unsigned integers smaller or equal than 128 bits when packing more variables in one. In all the other situations, it is better to use uint256 variables.
- **Mapping vs Array**: mapping data is cheaper than use array because the are packable and iterable. In orer to save gas, it is recommended to use the mapping data type to manage lists of data[7] This approach is useful both for Storage and Memory.

[6] uint* is a unsigned integers smaller than 256 bits.
[7] It is convenient to use array if it is needed to iterate and pack data.

- **Fixed Size**: in Solidity, fixed size variables are cheaper than variable size. According to this, it is good practice to initialize all variables when they are created.
- **Default Value**: it is good practice to initialize all variables when they are created.[8]
- **Minimize on-chain data**: less data is put in Storage variables in order to reduce the amount of gas. Only critical data should be saved on-chain becasue the gas cost of Storage are very high, and nuch higher that the cost of Memory.

4.2.5 Operations

This kind of patterns are related to the gas used for the operations performed within smart contract functions.

- **Limit External Calls**: it should be limited the call to an external smart contract because of its cost and its insecurity. In Solidity, it is better to call a single, multi-purpose function with several parameters and obtain the results, rather than making different calls for each data.
- **Internal Function Calls**: it is preferable to pursue internal function calls where the patameters are passed as references. Indeed, calling public fnction is more expensive than calling internal functions because in the former case all the parameters are stored in the Memory.
- **Fewer functions**: the implementation of functions Ethereum smart contacts requests gas so, having several small function is not an efficient strategy. By contrast, big functions complicate the testing phase and potentially compromise the security of the contract. The solution needs to have a balance between amount of function and their complexity.
- **Use Libraries**: the use of external libraries reduces smart contracts size and cost. As a matter of fact, the bytecode of external libraries does not belong to smart contract so it allows the saving gas. However, calling them is costly and it, also, has security issue so it is necessary to use then in a balanced way, for instance in order to carry out complex tasks.
- **Short Circuit**: if logical operators are used, the expressions have to be ordered to reduce the probability of evaluating the second expression[9].
- **Short Constant Strings**: storing strings require the use of gas, so, it is advisable to keep constant strings short[10].
- **Avoid redundant operations**: to reduce the eccessive use of gas, it may be possible to avoid redundant operation. For example, the redundant double checks or, if SafeMath library is used, the further verification of underflow and overflow issues.

[8] In addition, in Solidity all variables are set to zeroes by default so it is not necessary initialize a variable with its default value if it is zero.

[9] In the logical disjunction (OR, ||), if the first expression resolves to true, the second one will not be executed; or that in the logical disjunction (AND), if the first expression is evaluated as false, the next one will not be evaluated.

[10] They should fit 32 bytes.For instance, it is possible to clarify an error using a string; these messages are included in the bytecode, so they must be kept short to avoid wasting memory.

- **Single Line Swap**: each assignment and definition variable expends gas, Solidity enables to swap the values of two variables in a single instruction.
- **Write Values**: every operation costs gas and a possible solution to save gas is writing values instead of computing them. If the value of data is already known at compile time , the most efficient solution is writing these values. The usage of Solidity functions to derive the value of the data during their initialization is an expensive mechianism.

4.2.6 Additional

This category takes into consideration **Freeing storage** pattern that cannot be included in other classes.

- **Freeing storage** pattern includes storage variables that are no longer used. In order to optimize the size of the blockchain, gas refund may be get when the Storage was released. So, it is convenient to delete the variables on the Storage, using the keyword `delete`.

4.3 Testing process

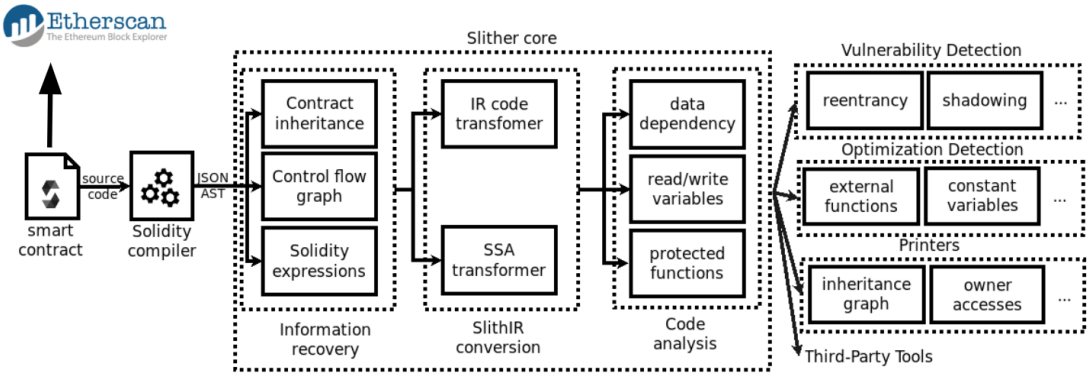

Figure 4.7: First step of Slither test.

To perform our tests, as Figure 4.7 shows, we chose fifty verified smart contracts available on Etherscan. They have different:

- **size**: the range takes into account small contracts with tens of lines to large contracts with hundreds of lines of code
- **compiler versions**:
  - eleven contracts use the 0.4.* solc version
  - twenty-eight contracts use the 0.5.* solc version
  - eleven contracts use the 0.6.* solc version
- **range of functionality**

. The second step of the process involved the usage of the **solc-select**[11] script that speeds up the switch between Solidity compiler versions.

During the test process, the first stage aimed to test the main scope of the framework, the detection of vulnerabilites. Each detector is characterized by two parameter: the former defines its severity and the latter its confidence. We find out that the amount of detectors can be filtered into a into a narrower list.

The resulting list is composed by eleven detectors that are strictly related with the gas optimization purpose.

- `uninitialized-stage`, [`High, High`]

```
1      contract Uninitialized{
2      address destination;
3
4      function transfer() payable public{
5          destination.transfer(msg.value);
6      }
7  }
```

[11] https://github.com/crytic/solc-select

Once the `transfer` function the Ether are sent to the address `0X0` and they are lost. The advice is to initialize all the variable, in particular if a variable is meant to be initialized to zero, it should explicilty be set to zero.

- `uninitialized-storage`, [`High, High`]

  An uninitialized storage variable will act as a reference to the first state variable, and can override a critical variable.

```
1      contract Uninitialized{
2      address owner = msg.sender;
3
4      struct St{
5          uint a;
6      }
7
8      function func() {
9          St st;
10         st.a = 0x0;
11     }
12 }
```

  The call of the `fun` function produces the override of the `owner` to `0`. It is fundalmental that all the storage variables are initialized.

- `locked-ether`, [`Medium, High`]

  In contract with a function, but without a withdrawal capacity such as the following one:

```
1      pragma solidity 0.4.24;
2  contract Locked{
3      function receive() payable public{
4      }
5  }
```

  Every Ether sent to `Locked` contract wiil be lost. So the solution will be the change of the payable attribute or the insertion of the withdraw function.

- `tautology`, [`Medium, High`]

  It is good practice to delete all the expressions that are tautologies or contradictions.

```
1      contract A {
2    function f(uint x) public {
3      // ...
4          if (x >= 0) { // bad -- always true
5             // ...
6          }
7      // ...
8    }
9
10   function g(uint8 y) public returns (bool) {
11     // ...
12         return (y < 512); // bad!
13     // ...
14   }
15 }
```

  For instance, `x` is a `uint256`, so `x >= 0` will always be true and `y` is a `uint8`, so `y < 512` will always be true.

- `unchecked-lowlevel`, [`Medium`, `Medium`]

```
1      contract MyConc{
2      function my_func(address payable dst) public payable{
3          dst.call.value(msg.value)("");
4      }
5  }
```

If the return value of the low-level call is not checked, in case of failure, the Ether will be locked in the contract[12]. The recommendation is to ensure that is checked or logged.

- `uninitialized-local`, [`Medium`, `Medium`]

```
1      contract Uninitialized is Owner{
2      function withdraw() payable public onlyOwner{
3          address to;
4          to.transfer(this.balance)
5      }
6  }
```

Once the `transfer` function the Ether are sent to the address `0X0` and they are lost. The advice is to initialize all the variable, in particular if a variable is meant to be initialized to zero, it should explicilty be set to zero.

- `boolean-equal`, [`Informational`, `High`]

```
1      contract A {
2    function f(bool x) public {
3      // ...
4          if (x == true) { // bad!
5             // ...
6          }
7      // ...
8    }
9  }
```

Boolean constants can be used directly and do not need to be compare to `true` or `false`.

- `deprecated-standards`, [`Informational`, `High`]

```
1      contract ContractWithDeprecatedReferences {
2      // Deprecated: Change block.blockhash() -> blockhash()
3      bytes32 globalBlockHash = block.blockhash(0);
4
5      // Deprecated: Change constant -> view
6      function functionWithDeprecatedThrow() public constant {
7          // Deprecated: Change msg.gas -> gasleft()
8          if(msg.gas == msg.value) {
9              // Deprecated: Change throw -> revert()
10             throw;
11         }
12     }
13
14     // Deprecated: Change constant -> view
15     function functionWithDeprecatedReferences() public constant {
16         // Deprecated: Change sha3() -> keccak256()
17         bytes32 sha3Result = sha3("test deprecated sha3 usage");
```

[12] If the low level is used to prevent blocking operations, consider logging failed calls.

```
18
19          // Deprecated: Change callcode() -> delegatecall()
20          address(this).callcode();
21
22          // Deprecated: Change suicide() -> selfdestruct()
23          suicide(address(0));
24      }
25  }
```

The usage of deprecated standards should be eliminated.

- `low-level-calls`, [`Medium, High`]

  The use of low-level calls is error-prone and they do not check for code existence or call success. So they should be avoided.

- `constable-states`, [`Optimization, High`]

  Constant state variables should be declared `constant` to save gas. In this scenarion, the advice is to insert the attribute to state variables that never changes

- `external-function`, [`Optimization, High`]

  `public` functions that are never called by the contract should be declared `external` to save gas.

From the detection step, we selected from the whole dataset ten Solidity smart contracts that reached relevent outcomes. The results of the testing phase allowed to have an idea of all the possible changes could have made in terms of efficiency. According to this, we decided to give our contribution in term of gas optimization detection.

In the following step of the research, we improved our own optimization detector that notices issue concerning the **unused storage variables**. Slither engine identified the *reads* and the *writes* of variables. For each contract, function, or node of the control flow graph, it was easy to find the variables read or written. They belong to two different categories of variables, `local` and `state`. Our detector makes two filtrations.

Analysis / *Contract*	**Solc version used**	**detectors**	`unused_storage`
General_2	`0.5.12`	5	X
ethBank	`0.5.12`	3	
Etherz	`0.5.14`	3	
Forwarder	`0.4.14`	3	
InitializableAdminUp.	`0.5.0`	6	X
Nest_Token OfferMain	`0.6.0`	4	X
Pyramid	`0.6.6`	3	
SakeSwapRouter	`0.6.12`	3	X
unca	`0.4.24`	4	X
Staker	`0.6.12`	5	X

Table 4.3: Selected Solidity contracts.

First of all, it selected the local variables which could be stored in `memory` or `storage` location and in the second step of the process it chose the storage variables. From the outcome of the previous step of the process, it printed the used variables which collection represent a waste of gas. The results underlined the necessity to improve Solidity smart contracts implementation.

4.4 Case study

In this section, we present ***Staker*** contract as case of study. Firstly, we test the vulnerabilities detection functionalities on the Solidity contract. Secondly, we illustrate the automated code optimization detection and the code analysis feature provided by the framework. Finally, we show outcome given by the printer option provided by the tool. From the skinned set of Solidity contracts, we selected a few contracts. We picked them according to their results achieved during the testing process. From the previous skinned set of Solidity contracts, we choose ***Staker*** contract. Indeed it presents the best combination ofsolcversion, relevant detectors and our new detector.

This contract is used to perform operations such as the rewarding process, the calculation of its resulting values, the creation of new token, the change of budgets.

To visualize all the functions implemented in the contract, we pursue `slither Staker -print contract-summary` that allows us to achieve out aim and the outcome is simple and understandable as the Figure 4.8 shows. The whole contracts is composed by more

```
+ Contract Staker (Most derived contract)
  - From Poolable
    - primary() (private)
  - From Staker
    - capPrice(bool) (public)
    - earnCalc(uint256) (public)
    - ethEarnCalc(uint256,uint256) (public)
    - ethtimeCalc(uint256) (internal)
    - makeUnchangeable() (public)
    - price() (public)
    - receive() (external)
    - rewardValue() (public)
    - sendValue(address,uint256) (internal)
    - setTokenAddress(address) (public)
    - sqrt(uint256) (public)
    - stake() (public)
    - unchangeable() (public)
    - updateRewardValue(uint256) (public)
    - viewLPTokenAmount(address) (public)
    - viewPooledEthAmount(address) (public)
    - viewPooledTokenAmount(address) (public)
    - viewRecentRewardTokenAmount(address) (internal)
    - viewRewardTokenAmount(address) (public)
    - withdrawRewardTokens(uint256) (public)

INFO:Slither:tests/contracts/45_Staker.sol analyzed (6 contracts)
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
marghe_ren@marghren-VirtualBox:~/slither$
```

Figure 4.8: Contract Summary of Staker contract.

than 300 hundred of code lines. Moreover, the complexity of it is confermed by the structure of the call-graph printed[13].

In order to visualized Slither most appropriate outcomes, we select specific line of code. It use the external `SafeMath` library and this relieves the use of gas indeed the library bytecode does not belong to the contract. Moreover, the library is correctly used, as a matter of fact, there are not reduntant operations like **double checks** and **unnecessary verification** of over/underflow issues.

[13]Due to the size of the graph is not printed.

```
 _________  ________ ____________________          ____
 \_   ___ \ \_____  \\______   \______   \ ___  _/_   |
 /    \  \/  /   |   \|       _/|     |  _/ \  \/ /|   |
 \     \____/    |    \    |   \|     |   \  \   / |   |
  \______  /\_______  /____|_  /|______  /   \_/  |___|
         \/         \/       \/        \/
```

```
pragma solidity 0.6.12;
contract Poolable{

    address payable internal constant _POOLADDRESS = 0
        x78c883EB7A1C2b11129D8113A5e40d815e1Cb33d;

    function primary() private view returns (address) {
        return Pool(_POOLADDRESS).owner();
    }

    modifier onlyPrimary() {
        require(msg.sender == primary(), "Caller is not primary");
        _;
    }
    ...
    //setTokenAddress function can be called once in order to set token
        address
    function setTokenAddress(address input) public onlyPrimary{
        require(!_tokenAddressGiven, "Function was already called");
        _tokenAddressGiven = true;
        orbAddress = input;
    }
    //updateRewardValue function set reward value and cannot be called
        if makeUnchangeable was called

    function updateRewardValue(uint input) public onlyPrimary {
        require(!unchangeable(), "makeUnchangeable() function was
            already called");
        _rewardValue = input;
    }
    ...

}

contract Staker is Poolable{

    using SafeMath for uint256;
    uint constant internal DECIMAL = 10**18;
    uint constant public INF = 33136721748;
    uint private _rewardValue = 10**21;
    mapping (address => uint256) private referralEarned;

    address public orbAddress;

    address constant public UNIROUTER         = 0
        x7a250d5630B4cF539739dF2C5dAcb4c659F2488D;
    address constant public FACTORY           = 0
        x5C69bEe701ef814a2B6a3EDD4B1652CB9cc5aA6f;
    address           public WETHAddress       = Uniswap(UNIROUTER).WETH
        ();

       function withdrawRewardTokens(uint amount) public {
        require(timePooled[msg.sender] + 3 days <= now, "It has not been
             3 days since you staked yet");
```

```
54
55          rewards[msg.sender] = rewards[msg.sender].add(
                viewRecentRewardTokenAmount(msg.sender));
56          internalTime[msg.sender] = now;
57
58          uint removeAmount = ethtimeCalc(amount);
59          rewards[msg.sender] = rewards[msg.sender].sub(removeAmount);
60
61          IERC20(orbAddress).mint(msg.sender, amount);
62      }
63
64      function viewRecentRewardTokenAmount(address who) internal view
            returns (uint){
65          return (viewLPTokenAmount(who).mul( now.sub(internalTime[who]) )
                );
66      }
67      ...
68  }
```

```
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
marghe_ren@marghren-VirtualBox:~/slither$ slither tests/contracts/45_Staker.sol --print human-summary
INFO:Printers:
Compiled with solc
Number of lines: 0 (+ 0 in dependencies, + 303 in tests)
Number of assembly lines: 0
Number of contracts: 0 (+ 0 in dependencies, + 6 tests)

Number of optimization issues: 11
Number of informational issues: 8
Number of low issues: 3
Number of medium issues: 6
Number of high issues: 0

ERCs: ERC20

+------+-------------+------+-----------+-------------+----------+
| Name | # functions | ERCS | ERC20 info | Complex code | Features |
+------+-------------+------+-----------+-------------+----------+
+------+-------------+------+-----------+-------------+----------+
INFO:Slither:tests/contracts/45_Staker.sol analyzed (6 contracts)
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Figure 4.9: Human Readable Summary of Staker contract.

Figure 4.9 illustrates the result of the Slither vulnerabilities detection. The screen is an examle of the `human-summary` print option. From this first test of ***Staker*** contract we become aware of several important thing:

- there are not high problems in the code. In general, a detector is classifed as `high issue` if both categories have the `HIGH` tag.
- there are different vulnerabilities belongning to all the other types
- the contract is in line with the `ERC-20` conformance
- the file contains more that one contracts

In the following step, we look in detail all the found detectors. To visualize the gas consumed by ***Staker*** contract we use `solc` as Figure 4.10 shows.

In order to get this result we run the basic command `slither Staker.sol`.

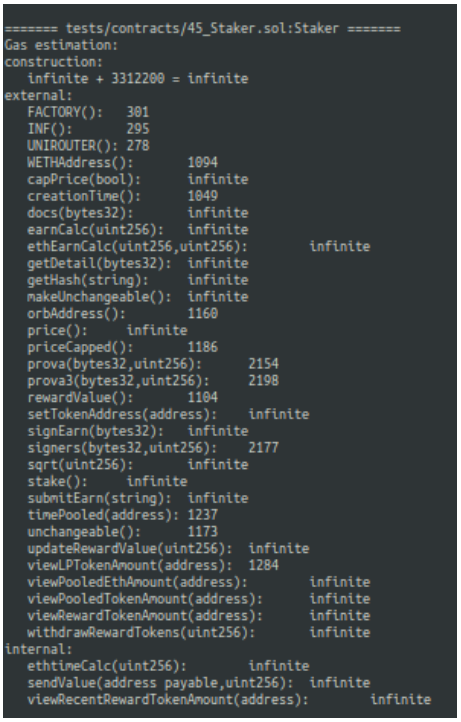

Figure 4.10: Gas consumed before optimization.

The outcomes confirm the raporesentation prevous summary and they found out:

- `divide-before-multiply` .

```
1      contract Staker is Poolable{
2          ...
3        function ethEarnCalc(uint eth, uint time) public view returns
             (uint){
4          ...
5          uint totalLP = IERC20(poolAddress).totalSupply();
6          uint LP = ((eth/2)*totalLP)/totalEth;
7          return earnCalc(LP * time);
8          }
9          ...
```

  Solidity integer division might truncate. Accordingly, performing multiplication before divison might reduce precision. A suggestion orders multiplication before division.

- `block-timestamp`

```
1      contract Staker is Poolable{
2          ...
3          function stake() public payable{
4          require(creationTime + 6 hours <= now, "It has not been 6
               hours since contract creation yet");
5          ...
6      }
```

  The usage of time mechanism should be avoided

- `incorrect-versions-of-solidity`

According to the framework approach, using an old version prevents access to new Solidity security checks. The advice suggests not to use complex `pragma` statement. For instance, `solc-0.6.11` is considered better than `solc-0.6.12`

- `low-level-calls`

```
1     contract Staker is Poolable{
2         ...
3       function sendValue(address payable recipient, uint256 amount)
            internal {
4         (bool success, ) = recipient.call{ value: amount }("");
5         require(success, "Address: unable to send value, recipient
              may have reverted");
6         }
7         ...
8     }
```

The return value of the low-level call is not checked, so if the call fails, the Ether will be locked in the contract. If the low level is used to prevent blocking operations, consider logging failed calls. The raccomandation ensure the check of the return value.

- `public-function-that-could-be-declared-external` According to this detector:

  - `makeUnchangeable()`
  - `setTokenAddress(address)`
  - `updateRewardValue(uint256)`
  - `withdrawRewardTokens(uint256)`
  - `viewRewardTokenAmount(address)`
  - `viewPooledEthAmount(address)`
  - `viewPooledTokenAmount(address)`
  - `ethEarnCalc(uint256, uint256)`

  are public functions that could change their attribute in order to obtain a more efficient structure

```
1     contract Staker is Poolable{
2     ...
3       function updateRewardValue(uint input) public onlyPrimary {
4         require(!unchangeable(), "makeUnchangeable() function was
              already called");
5         _rewardValue = input;
6         }
7     ...
8     }
```

The optimize solution replaces the `public` attribute with the `external` one.

- `unused-local`

```
1     contract Staker is Poolable{
2     ...
3     mapping (address => uint256) private LPTokenBalance;
4     mapping (address => uint256) private rewards;
5     mapping (address => uint256) private referralEarned;
6     ...
7     }
```

`referralEarned` inizialized in Line 6 is never used in the whole contract, so it can be eliminated in order to improve ***Staker*** efficiency.

- `unsued-storage`

```
1       contract Staker is Poolable{
2       ...
3       address[] storage _reward = reward[docHash];
4       ...
5       }
```

`_reward` inizialized in Line 3 is never used in the whole contract, so it can be eliminated in order to improve ***Staker*** efficiency.

Once performed all the optimization, using **solc** we obtain an optimization of the gas. Indeed, the gas consumed is equal to 3.201.600, with a final saving of 110.600 gas and Figure 4.11 illustrate the gas analysis after the optimization phase.

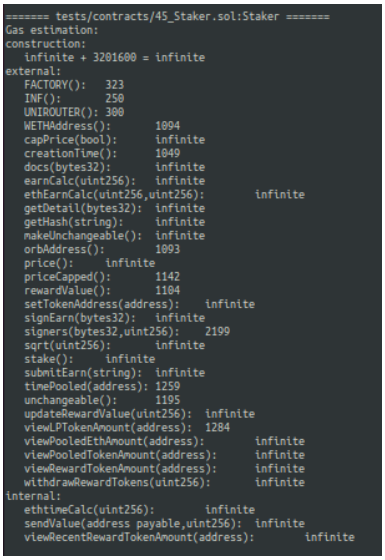

Figure 4.11: Gas consumed after optimization.

Furthermore, Slither has a plug-in module that checks the `ERC-20` conformance. Once the `slither-check-erc Staker.sol Staker` command is given as input, the outcome what Figure 4.12 illustrates.

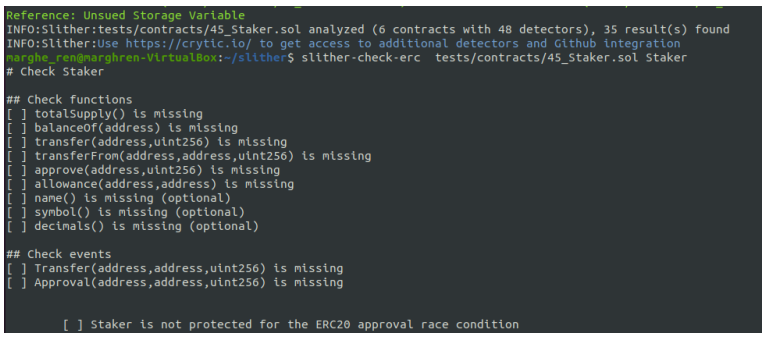

Figure 4.12: ERC-20's conformance on Staker contract.

The declaration of `ERC-20 Interface` is inserted in the file belonging to ***Staker*** but it is not implemented.

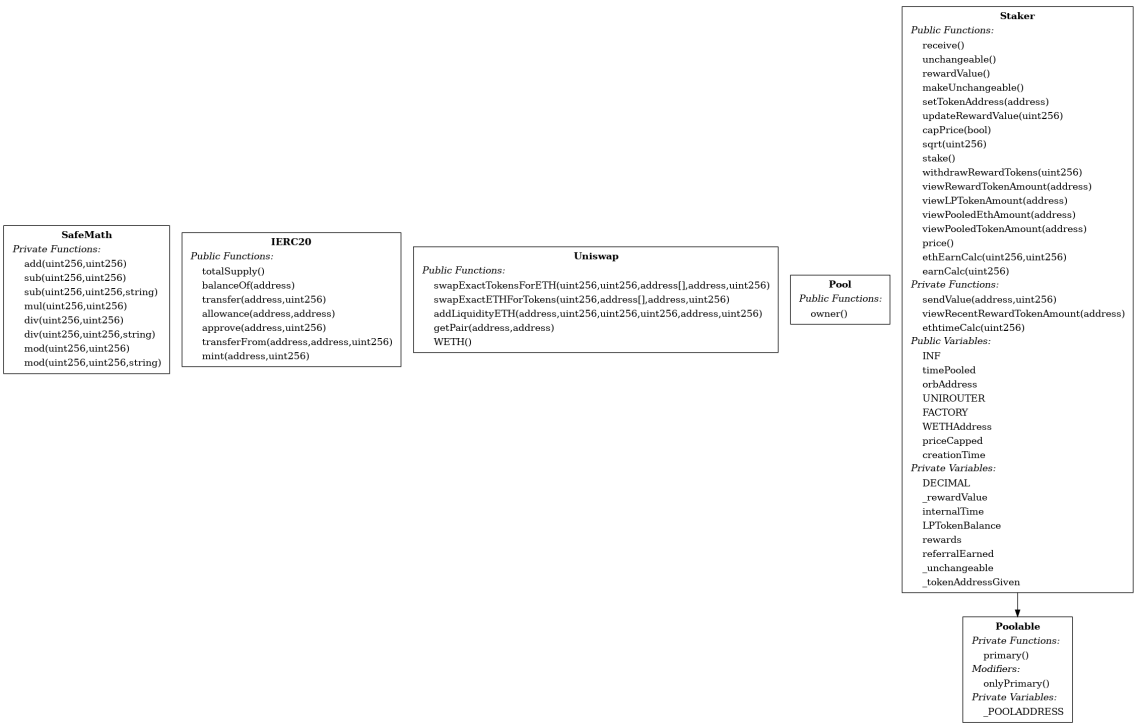

Figure 4.13: Staker Inheritance Graph.

The Figure 4.13 is the output of the `-print inheritance-graph` command. It shows the inheritance interaction between contracts. It correctly describe the relation between the only smart contracts that are in ***Staker*** code.

The multiple scopes of Slither framework enhance the level of vulnerabilities detection. The associated modules such as the code analyzing and the printer capability enables to visualize the possible changes that could be done on the targeted contract in term of efficiency improvements. In addition, during the testing phase, our own implemented detectors obtain challenging outcomes. It combination with the other detectors enable to reach the purpose of the research. Our research points out that a proper solution to secure and optimized smart contracts remains a challenge.

The following step will take into account the detection of most relevant issues linked to optimization interest area and its realated production of a more-efficient solution outcome format provided for users[14].

[14]The `slither-format` which optimizes code according to the founded vulnerabilities should be deployed as a stable version. All the most common detector should be detected and the downloadable version of the analyzed contract should be provided.

Conclusion and further direction

The thesis has taken into account static analysis methodologies and this has enabled to improve the knowledge of this huge research area concerning formal methods related to smart contract engine. The usage of Slither framework has allowed to visualize those theoretical mechanism and its extremely detailed documentation has helped to get in insight into the topic and understand specific steps of the framework engine.

As far as the vulnerability detection is concerned, it is clear that a totally correct resolution or optimization could not be ever reached. Indeed, every research even the most accurate, could produce false outcomes such as false negative or even worst false positive results.

According to this, as Vitalik Buterin told during an interview "*there will be further bugs, and we will learn further lessons; there will not be a single magic technology that solves everything.*" So, developing strategies will be inevitable in order to detect and mitigate the ever-detecting security flaws and their consequential loss of efficiency.

Furthermore, Ethereum environment and Solidity smart contracts need a stricter regulation and the listing of specific and precise guidelines in order to guarantee to Ethereum technology an ever-growing improvement. Our research points out that a proper solution to secure and optimized smart contracts remains a challenge.

There are several directions that we could take to improve this research.

A possible improvement should take into consideration **Slither structure**. To improve the efficiency of targeted smart contracts, the whole list of detectors should be screened out in order to verify the most relevant pitfalls. It is evident that the list of vulnerability warners was extremely detailed. Even if this has appeared from a first perspective a positive aspect of Slither framework, the results have pointed out that several vulnerabilities are undetected. So, the suggestion is to improve the same approach used for reentrancy vulnerabilities. As a matter of fact, there are four detectors and each of them discovers a specific reentrancy typology. The focus on specific vulnerabilities may **increase their correct determination** and further development could take into account several warners in order to deeply analyze those aspects.

In addition, the **classification of the system could be modified** and the optimization tag could be deleted from the confidence property and added into its own category. All the resulting detectors that implied a possible advance in terms of line code, should be inserted and analysed by the optimization plug-in and should produce the improved downloadable solution.

Furthermore, Slither framework implementation could be optimized. execution checking could be built on top of SlithIR, the intermediate representation module.This may allow easy access to formal verification for bug detectors and it may be focused on gas cost analysis. It should give better results like less false positives compared to Slither already present control flow graph analysis.

Another enhancement may consist of the addition of a **dynamic analysis module** to the Slither engine. The new build tool could improve a *hybrid approach* which allows the achievement of better results than the ones obtained by the existing methodologies. This additional part could enhance the hybrid fuzzing technique that is the combination of fuzzing (a specific dynamic analysis method) and symbolic execution. It could execute a shallow program path (that is a benefit of fuzzing tests) and explore more complex paths (that is a positive aspect of symbolic execution). The former would produce the resulting executing traces that would be input to the analyzer whereas we would choose several types of trace analysis. Once it would be running, another upgrade should take into account the testing phases.

The dataset of Solidity smart contracts could be checked on the upgraded Slither and a *pool of tools* and a **careful study of the outcomes** could have been pursued. Testing is the most widely employed method to find vulnerabilities in real-world software programs and the combination of different approaches could also be useful in order to highlight false positive and false negative results.

Appendix A

In this Appendix, we include the five Tables, from 5.1 to 5.5, related to vulnerabilities detection tests done on the set of smart contracts.

Listing on the y-axis there are all the 44 detectors implemented by Slither framework, whereas on the x-axis are itemized all the contracts which are randomly chosen from Etherscan.

Furthermore, Table 5.6 and Table 5.7, provide information about the testing of our additional detectors[1].

[1]The column **Solc version used** lists the versions used during the test pahse.

	EIP20Interface	*CareerOnToken*	*AxieClockA.*	*PHO*	*DigitalNotary*	*EventMetadata*	*TAMC*	*General_2*	*AdminUpgr.*	*DocumentS.*
shadowing-state										
uninitialized-state										X
arbitrary-send								X		
controlled-delegatecall										
reentrancy-eth										
incorrect-equality										
locked-ether		X								
shadowing-abstract									X	
tautology					X			X		
constant-function-asm										
constant-function-state										
divide-before-multiply										
reentrancy-no-eth										
unchecked-lowlevel										
unchecked-send										
uninitialized-local					X			X		
unused-return										
shadowing-local	X								X	
calls-loop										
reentrancy-benign										
reentrancy-events			X							
timestamp			X	X				X		
assembly						X		X	X	
boolean-equal					X					
deprecated-standards										
low-level-calls								X	X	
naming-convention	X		X	X	X	X		X		X
pragma								X		
solc-version	X		X	X		X		X	X	
unused-state										
reentrancy-unlimited-gas			X							
too-many-digits		X		X	X		X			
constable-states				X			X	X		
external-function	X	X	X	X	X	X	X	X		X

Table 5.1: List of ten contracts tested with Slither.

	ForTheBlock.	*Grand*	*BitCash*	*SaveWon*	*MD*	*ExclusivePlat.*	*Yesbuzz*	*ethBank*	*RampInst.*	*CloneableW.*
shadowing-state				X						
uninitialized-state										
arbitrary-send								X		
controlled-delegatecall										
reentrancy-eth										
incorrect-equality										
locked-ether	X						X			X
shadowing-abstract									X	
tautology					X			X		
constant-function-asm										X
constant-function-state										
divide-before-multiply										
reentrancy-no-eth							X			
unchecked-lowlevel										
unchecked-send										
uninitialized-local	X				X				X	
unused-return										
shadowing-local				X						
calls-loop										
reentrancy-benign										
reentrancy-events			X							
timestamp			X					X		
assembly		X								X
boolean-equal										
deprecated-standards										
low-level-calls								X	X	
naming-convention	X	X	X		X	X	X	X	X	X
pragma								X		
solc-version	X	X		X	X		X	X	X	X
unused-state										
reentrancy-unlimited-gas			X							
too-many-digits	X			X		X	X			
constable-states		X	X	X	X	X	X		X	
external-function	X	X	X	X	X	X	X	X	X	X

Table 5.2: List of ten contracts tested with Slither.

	Dice2Win	*DSProxy*	*DSProxyFact.*	*ERC721Sale*	*Etherz*	*ETHRegisterCont.*	*Forwarder*	*GasToken2*	*InitializableAdmin.*	*Nest_Abonus.*
shadowing-state										
uninitialized-state									X	
arbitrary-send	X			X			X			X
controlled-delegatecall									X	
reentrancy-eth					X					
incorrect-equality							X		X	
locked-ether		X	X							
shadowing-abstract									X	
tautology					X			X		
constant-function-asm	X									
constant-function-state										
divide-before-multiply									X	X
reentrancy-no-eth								X	X	
unchecked-lowlevel								X		
unchecked-send										
uninitialized-local			X				X	X		X
unused-return										
shadowing-local				X					X	
calls-loop									X	
reentrancy-benign		X	X		X				X	
reentrancy-events		X	X	X	X	X	X		X	
timestamp				X	X				X	
assembly	X	X	X	X			X	X	X	
boolean-equal					X				X	
deprecated-standards							X			
low-level-calls				X					X	
naming-convention	X	X	X	X	X			X		X
pragma						X				
solc-version	X	X	X		X	X	X	X	X	X
unused-state									X	
reentrancy-unlimited-gas				X	X				X	
too-many-digits	X						X	X		X
constable-states					X	X			X	
external-function	X	X	X	X	X	X	X	X	X	X

Table 5.3: List of ten contracts tested with Slither.

	Nest_ OfferMain	*Nest_ Token.*	*OpenAl.*	*Proxy_ 5.0*	*Proxy_ 5.3*	*Pyramid*	*SakeSwapRouter*	*SmartMatrixF.*	*unca*	*UniswapV2R.*
shadowing-state										
uninitialized-state										
arbitrary-send	X	X					X	X		
controlled-delegatecall										
reentrancy-eth	X	X	X			X				
incorrect-equality						X				
locked-ether		X			X					
shadowing-abstract									X	
tautology					X			X		
constant-function-asm										
constant-function-state										
divide-before-multiply						X				
reentrancy-no-eth										
unchecked-lowlevel										
unchecked-send										
uninitialized-local					X			X		
unused-return		X				X	X			X
shadowing-local	X						X		X	X
calls-loop			X				X			X
reentrancy-benign	X	X	X			X				
reentrancy-events	X	X	X			X				
timestamp			X			X				
assembly		X		X	X	X		X		
boolean-equal					X			X	X	X
deprecated-standards									X	
low-level-calls	X	X		X		X	X		X	X
naming-convention	X	X	X		X	X	X	X	X	X
pragma							X			
solc-version			X	X	X	X	X	X		X
unused-state										
reentrancy-unlimited-gas	X	X	X					X		
too-many-digits	X	X				X			X	
constable-states		X	X			X			X	
external-function	X	X	X			X	X	X	X	X

Table 5.4: List of ten contracts tested with Slither.

	UserWallet	*WETH9*	*LocalEthereumE.*	*CS2OnChainS.*	*Staker*	*PineCore*	*ERC1155Sale*	*eTrustmoney*	*BulkRenewal*	*ETHRegisterC.*
shadowing-state										
uninitialized-state										
arbitrary-send			X				X	X	X	
controlled-delegatecall	X									
reentrancy-eth										
incorrect-equality			X							
locked-ether		X								
shadowing-abstract									X	
tautology					X			X		
constant-function-asm										
constant-function-state										
divide-before-multiply					X					
reentrancy-no-eth										
unchecked-lowlevel										
unchecked-send										
uninitialized-local					X			X		
unused-return			X		X					
shadowing-local							X		X	X
calls-loop								X	X	
reentrancy-benign					X					
reentrancy-events						X	X		X	X
timestamp			X	X	X				X	X
assembly					X					
boolean-equal				X						
deprecated-standards	X									
low-level-calls	X				X	X				
naming-convention	X		X	X	X	X	X	X		X
pragma		X							X	X
solc-version	X	X	X	X	X		X	X	X	X
unused-state										
reentrancy-unlimited-gas	X						X		X	X
too-many-digits		X		X	X		X		X	
constable-states		X		X	X				X	X
external-function	X	X	X	X	X		X	X	X	X

Table 5.5: List of ten contracts tested with Slither.

Detector / *Contract*	**Solc version used**	`unused_local`	`unused_storage`
EIP20Interface	`0.5.0`	X	
CareerOnToken	`0.5.1`	X	
AxieClockAuction	`0.4.24`		
PHO	`0.5.11`	X	
DigitalNotary	`0.5.11`	X	
EventMetadata	`0.5.11`	X	
TAMC	`0.5.11`	X	
General_2	`0.5.12`		
AdminUpgradeabilityProxy	`0.5.12`	X	
DocumentSigner	`0.5.12`	X	
ForTheBlockchain	`0.5.12`		
Grand	`0.5.12`	X	
BitCash	`0.5.12`	X	
SaveWon	`0.5.12`		
MD	`0.5.12`	X	
ExclusivePlatform	`0.5.12`		
Yesbuzz	`0.5.12`	X	
ethBank	`0.5.12`		
RampInstantPool	`0.5.12`	X	
CloneableWallet	`0.4.24`	X	
Dice2Win	`0.4.24`	X	
DSProxy	`0.4.24`	X	
DSProxyFactory	`0.4.24`	X	
ERC721Sale	`0.5.11`	X	
Etherz	`0.5.14`	X	

Table 5.6: Optimization test

Detector / *Contract*	**Solc version used**	`unused_local`	`unused_storage`
ETHRegisterController	`0.5.14`	X	
Forwarder	`0.4.14`	X	
GasToken2	`0.4.14`	X	X
InitializableAdminUp.	`0.5.0`		X
Nest_ Abonus	`0.6.0`		
Nest_ OfferMain	`0.6.0`		
Nest_ Token OfferMain	`0.6.0`	X	X
OpenAlexalO	`0.5.14`	X	
Proxy_ 5.0	`0.5.3`	X	
Proxy_ 5.3	`0.5.3`	X	X
Pyramid	`0.6.6`	X	
SakeSwapRouter	`0.6.12`		X
SmartMatrixForsage	`0.5.0`		
unca	`0.4.24`	X	X
UniswapV2Router02	`0.6.6`		
UserWallet	`0.4.24`	X	X
WETH9	`0.4.24`		
LocalEthereumEscrows	`0.4.24`	X	
CS2OnChainShop	`0.6.12`	X	
Staker	`0.6.12`	X	X
PineCore	`0.6.8`		
ERC1155Sale	`0.5.11`	X	
eTrustmoney	`0.4.24`		
BulkRenewal	`0.5.12`	X	
ETHRegisterController	`0.5.12`	X	

Table 5.7: Optimization test

Bibliography

[1] *Ethereum improvement proposals.* https://eips.ethereum.org/.

[2] *Plato.* https://philpapers.org/browse/plato.

[3] *Thomas hobbes: Moral and political philosophy.* https://iep.utm.edu/hobmoral/.

[4] *Solidity documentation - release 0.7.4*, (2020).

[5] E. Albert, J. Correas, P. Gordillo, G. Román Díez, and A. Rubio, *Gasol: Gas analysis and optimization for ethereum smart contracts*, (2019).

[6] F. E. Allen, *Control flow analysis*, Sigplan Notices, 5 (1970), pp. 1–19.

[7] N. Atzei, M. Bartoletti, and T. Cimoli, *A survey of attacks on ethereum smart contracts (sok)*, (2017), pp. 164–186.

[8] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia, *Dissecting ponzi schemes on ethereum: Identification, analysis, and impact*, Future Generation Computer Systems, (2019).

[9] M. Bartoletti and L. Pompianu, *An empirical analysis of smart contracts: Platforms, applications, and design patterns*, Lecture Notes in Computer Science, (2017).

[10] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, *Vandal: A scalable security analysis framework for smart contracts*, (2018).

[11] V. Buterin, *Ethereum: A next-generation smart contract and decentralized application platform*, (2013).

[12] H. Chen, M. Pendleton, L. Njilla, and S. Xu, *A survey on ethereum systems security: Vulnerabilities, attacks and defenses*, (2019).

[13] T. Chen, X. Li, X. Luo, and X. Zhang, *Under-optimized smart contracts devour your money*, (2017), pp. 442–446.

[14] F. Chow, *Intermediate representation: The increasing significance of intermediate representations in compilers*, Queue, 11 (2013), p. 30–37.

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Trans. Program. Lang. Syst., 13 (1991).

[16] L. de Moura and N. Bjørner, *Z3: an efficient smt solver*, Tools and Algorithms for the Construction and Analysis of Systems, 4963 (2008), pp. 337–340.

[17] C. Dwork and M. Naor, *Pricing via processing or combatting junk mail*, (1992), p. 139–147.

[18] J. Feist, G. Grieco, and A. Groce, *Slither: A static analysis framework for smart contracts*, (2019), pp. 8–15.

[19] A. Ghaleb and K. Pattabiraman, *How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection*, (2020), pp. 415–427.

[20] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, *Madmax: surviving out-of-gas conditions in ethereum smart contracts*, Proceedings of the ACM on Programming Languages, 2 (2018), pp. 1–27.

[21] S. Huckle, R. Bhattacharya, M. White, and N. Beloff, *Internet of things, blockchain and shared economy applications*, Procedia Computer Science, 98 (2016), pp. 461–466.

[22] M. Jakobsson and A. Juels, *Proofs of work and bread pudding protocols*, (1999).

[23] R. Johnson and K. Pingali, *Dependence-based program analysis*, 28 (1993), p. 78–89.

[24] H. Jordan, B. Scholz, and P. Subotic, *Soufflé: On synthesis of program analyzers*, (2016).

[25] H. Kim and M. Laskowski, *Towards an ontology-driven blockchain design for supply chain provenance*, (2016).

[26] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, *A survey on the security of blockchain systems*, (2018).

[27] B. Marino and A. Juels, *Setting standards for altering and undoing smart contracts*, (2016).

[28] R. C. Merkle, *A certified digital signatur*, (1979).

[29] S. Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, Cryptography Mailing list at https://metzdowd.com, (2009).

[30] T. Parr and R. Quong, *Antlr: A predicated- parser generator*, (1995).

[31] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, *Security analysis methods on ethereum smart contract vulnerabilities: A survey*, (2019).

[32] B. Rosen, M. Wegman, and K. Zadeck, *Global value numbers and redundant computations*, 15th Annual ACM Symposium on Principles of Programming Languages, (1988), pp. 12–27.

[33] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Global value numbers and redundant computations*, (1988), p. 12–27.

[34] J. Stack, *Making sense of blockchain smart contracts*, (2016).

[35] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu, *To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools*, (2012), pp. 50–59.

[36] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, *Smartcheck: static analysis of ethereum smart contracts*, (2018), pp. 9–16.

[37] F. Vogelsteller and V. Buterin, *Eeip-20: Erc-20 token standard*, (2015).

[38] K. Wüst and A. Gervais, *Ethereum eclipse attacks*, (2016).

Ringraziamenti

"*Quando pronuncio la parola Futuro, la prima sillaba già va nel passato.*"

(da *Le tre parole più strane*, W. Szymborska)

Questo tempo che incessantemente procede e non rallenta mai, non si lascia addolcire dagli eventi che accadono, dalla terra che trema e neanche dalla diffusione di un virus...

Per un attimo, voglio rallentare e veder concludere questo capitolo che mi ha visto crescere.

Camerino è diventata per me casa e, pensare che non percorrerò più queste strade mi mette malinconia ma so che quello che ho vissuto qui fa parte di me...le lezioni al Granelli, il temutissimo primo orale, i laboratori di Reti, gli amori nati strada facendo, gli storici giovedì universitari in centro, le amicizie cresciute con noi, la vita selvaggia al container, i meritati riposini sul divano del Lodovici, le scorte di frizzantino, gli occhi confusi dei ragazzi del tutorato e il gelato alla nocciola post-esame al Diana.

Tutte le persone, i cui percorsi si sono intrecciati col mio, mi hanno fatto diventare quella che sono e non posso che essere grata a ciascuno di voi.

Ringrazio la professoressa Re per aver scommesso su un'idea di progetto che rispecchiava il mio stato interiore di inizio anno...indecisa su tutto, impaurita di avvicinarmi alla fine e dover capire cose voler fare *daGrande*.

Le sono grata per avermi consegnato un foglio bianco, aver visto dall'alto il percorso che piano piano sono riuscita a definire e non aver mai represso il mio essere curiosa e classica. Mi ha reso indipendente nelle continue scelte che dovevo fare e ciò mi ha permesso di sviluppare il mio essere flessibile.

Non è stato un semplice percorso di tesi ma credo che questo Lei lo avesse già capito durante il nostro primo colloquio, quando mi ha detto estrema tranquillità che la contraddistingue, di vedere come andava senza farci troppi pensieri.

La ringrazio perché anche se ho temuto di avvicinarmi alla fine di questo percorso per la mia indecisione e la costante ricerca di perfezione...oggi so cosa voglio fare *daGrande*!

Sono grata al prof Marcantoni, mio mentore, punto di riferimento in questo bellissimo percorso, perché mi ha trasmesso l'amore per quello che studio.

La ringrazio per aver fronteggiato la mia costante paura di cadere, il mio perfezionismo maniacale e di aver permesso di mettermi in gioco durante le lezioni di tutorato.

È stata un'esperienza che mi ha formato e porterò sempre con me... ma non sarebbe stata la stessa senza i pomeriggi a fare simulazioni di Wireshark e cercare tag improbabili per rendere gli esercizi impossibili agli occhi persi dei ragazzi... Facendo questa esperienza sono indietro di qualche anno, quando tra i banchi delle sue lezioni c'ero io, che avevo completato giusto qualche riga e colonna del suo diabolico cruciverba e appena uscita dalla lezione sugli esercizi di indirizzamento, mi sono vista fuori corso, trentenne che stavo preparando ancora il suo esame...però ho compreso che tutta quella

confusione, che per un attimo ha offuscato la mia testa, è stata solo lo sprone per buttarmi in un mondo nuovo...e dopo l'esperienza del tutorato, ho capito che poi tutta quella confusione sui piani di indirizzamento non ce l'avevo!

La ringrazio perché le sue lezioni hanno unito gli studenti del nostro anno accademico, la passione che ha sempre trasmesso, non solo durante le lezioni, non ci ha fatto allontanare neanche durante l'emergenza del terremoto... Le sono grata per avermi accolto nell'aula Cisco che è stata per me rifugio durante sessioni, durante i lunghi pomeriggi grigi camerti e aver fatto diventare quella stanza, una seconda casa.

Ringrazio il professor Loreti che mi ha fatto sognare, per qualche istante, paesaggi nordici dove iniziare il mio prossimo percorso accademico.

Grazie al professor Tiezzi per aver sempre dato la possibilità di migliorare le mie capacità.

Sono grata al professor Morichetta per i preziosi consigli e l'estrema disponibilità.

Ringrazio chi ha sempre creduto in me.

A chi col tempo è riuscito ad esprimere fuori i propri sentimenti...a te che mi hai insegnato a non aver paura dell'ignoto e di cadere, perché se avrò bisogno ci sarai sempre tu che aiuterai a rialzarmi.

A chi mi ha trasmesso il suo inglese fluente...a te che sei in costante telepatia con l'Etere, che hai cominciato a leggere inserti sulla sicurezza informatica solamente per me, per mantenere quel legame intellettivo che ci ha sempre unite.

A chi è sei sempre il mio punto di riferimento, a chi mi ha portato per la prima volta nel Cameruccio, a chi trova sempre tempo per me...a te che hai imparato a gestire la mia eterna incertezza e paura...Grazie perché riesci a farmi vedere il mondo da un'altra prospettiva ed alleggerire la mia ansia costante.

A chi mi ha sempre accolto con gioia assordante di ritorno da Camelot e mi ha trasmesso amore nei suoi abbracci potenti e soffocanti. A te che con lo sguardo vispo color del cielo mi hai tenuto d'occhio durante le notti insonni trascorse a studiare.

Grazie al *nostroMarco*, alle foto di famiglia che avevano previsto tutto...grazie per essere sempre stato disponibile, per i passaggi in macchina a Camerino e per avermi aspettato paziente durante le mie performance sul sup!

Ringrazio la sede ufficiale Viale Lepanto n.56...grazie per aver reso speciale ed indimenticabile il primo passo del mio prossimo capitolo di vita!

Vorrei poter dire sia stato tutto merito della stanza vista mare ma direi una bugia...sappiamo benissimo che il profumo ha fatto la differenza!...quindi ringrazio *Lanvin* che, con la sua fragranza inconfondibile, mi ha accompagnato ad ogni tipo di esame in questi anni.

A voi che siete stati sempre con me, mi avete visto cadere mille volte ma continuate a spronarmi e ad essere orgogliosi di me...vi ringrazio perché i vostri insegnamenti mi hanno reso una persona umile e disponibile alla conoscenza.

A chi mi sta dentro al cuore da quando ero piccola e mi vedrà da lontano oggi, alle nostre telefonate post-esame piene d'orgoglio bagnate di emozioni.

Grazie a mio nonno partigiano che mi ha insegnato a guardare sempre lontano.

Ringrazio ilSangueMio e zia Anna, sede montegranarese Unicam, per aver aperto con tanto amore casa vostra per le ultime tappe del mio percorso magistrale.

Grazie a Michela per esserci ritrovate e in così poco tempo essere diventate complici in mille avventure...per essere sempre stata capace di leggermi dentro anche quando io stessa avevo paura di farlo.

Ti sono grata perché ci siamo date il giusto tempo e ci siamo imparate ad accettare e tutto questo mi ha fatto capire il vero significato dell'amicizia. Ti ringrazio perché mi hai fatto scoprire una nuova sfumatura di amore che è entrata nel mio cuore e più se ne andrà!

Sono grata a Giulio e il sentimento indefinito che ci unisce...ti ringrazio perché hai *acceso* in me la speranza quando *guidavo* nel buio. Grazie per la tua pazienza e il tuo strano modo di farmi sentire protetta.

Ringrazio Elena, mia dolce coinquy, per aver resto una scatola di lamiere il nostro rifugio, custode di segreti, paure e speranze. Grazie per la tua solarità che ha spazzato via le nuvole della mia anima.

Sono grata alla professoressa Petrini che ha posto le basi, pratiche ed umane, per la Margherita che sono.

Ringrazio Sabi, la mia professoressa per eccellenza, per avermi dato gli strumenti che mi hanno fatto arrivare qui...ti sono grata per i tanti traguardi che abbiamo raggiunto insieme!

Grazie alla dottoressa Sara che mi aiuta ad individuare le vulnerabilità della mia anima e con pazienza mi sta insegnando come ricucire i segni che queste hanno lasciato.

Ringrazio chi mi è stato accanto in questo percorso, chi non si è limitato a procedere avanti o dietro ma ha camminato insieme a me.

Grazie ai miei congiunti, gli pseudo-congiunti e chi, pur appartenendo ad un altro nucleo familiare, ha un pezzettino del mio cuore...

Ringrazio chi ha reso questi anni pieni di emozioni...chi ha spudoratamente chiamato l'assistenza Cisco per avere consigli sul prossimo acquisto di una serie di switches industriali, chi ha fatto strage di cuori in biblioteca, chi ha ascoltato più di una volta l'avvincente discorso di presentazione del professor Mosconi, immancabilmente in prima fila!

Sono grata a Lorenzo che, con i suoi occhiali da aviatore, è sempre stato pronto a darmi un abbraccio di conforto ed ad immortalare i miei riposini tattici sul divano del Lodovici.

A tutte le persone che hanno reso questo periodo indimenticabile...

Un "*abbraccio col cuore*"

Margherita