# Evaluation of a Secure Smart Contract Development in Ethereum

DANIEL DA ROCHA MAIA DIAS

Outubro de 2020

# Evaluation of a Secure Smart Contract Development in Ethereum

Daniel Dias

A dissertation submitted in fulfillment of
the requirements for the degree of Master in Computer
Engineering, Specialisation Area of Software Engineering

Supervisor: Isabel Azevedo

Porto, October 15, 2020

# Abstract

In the Ethereum Blockchain, Smart Contracts are the standard programs that can perform operations in the network using the platform currency (ether) and data. Once these contracts are deployed, the user cannot change their state in the system. This immutability means that, if the contract has any vulnerabilities, it cannot be erased or modified. Ensuring that a contract is safe in the network requires the knowledge of developers to avoid these problems. Many tools explore and analyse the contract security and behaviour and, as a result, detect the vulnerabilities present.

This thesis aims to analyse and integrate different security analysis tools in the smart contract development process allowing for better knowledge and awareness of best practices and tools to test and verify contracts, providing a safer smart contract to deploy.

The development of the final solution that allows the integration of security analysis tools in smart contracts was performed in two stages. In the first stage, approaches, patterns and tools to develop smart contracts were studied and compared, by running them on a standard set of vulnerable contracts, to understand how effective they are in detecting vulnerabilities. Seven existing tools were found that can support the detection of vulnerabilities during the development process.

In the second stage, it is introduced a framework called EthSential. EthSential was designed and implemented to initially integrate the security analysis tools, Mythril, Securify and Slither, with two ways to use, command line and Visual Studio Code. EthSential is published and publicly available through PyPI and Visual Studio Code extensions.

To evaluate the solution, two software testing methods and a usability and satisfaction questionnaire were performed. The results were positive in terms of software testing. However, in terms of usability and satisfaction of the developers, the overall results did not meet expectations, concluding that improvements should be made in the future to increase the developers' satisfaction and usability.

**Keywords:** Ethereum, Smart Contracts, Solidity, Security, Analysis, Vulnerabilities

# Resumo

Em Ethereum, contratos inteligentes são programas que permitem realizar operações na rede utilizando a moeda digital (ether) e os dados armazenados na mesma. Assim que estes contratos são enviados para a plataforma, o utilizador é impedido de alterar seu estado. Esta imutabilidade faz com que se o contrato tiver alguma vulnerabilidade, não poderá ser apagado ou modificado. Para garantir que um contrato seja considerado seguro, requer um conhecimento dos programadores em lidar com estas vulnerabilidades. Existem muitas ferramentas que exploram e analisam a segurança e o comportamento do contrato de forma a detectar as vulnerabilidades presentes.

Esta tese tem como objectivo analisar e integrar diferentes ferramentas de análise de segurança no processo de desenvolvimento de contratos inteligentes. De forma a permitir um melhor conhecimento e consciência das melhores práticas é necessário analisar as ferramentas de teste e verificação de contratos, proporcionando assim um contrato mais seguro.

O desenvolvimento da solução final foi realizado em duas fases. Na primeira fase, foram estudadas abordagens, padrões e ferramentas para desenvolver contratos inteligentes, e comparar essas ferramentas, executando-as num conjunto de contratos vulneráveis, para entender o quão eficaz são na detecção de vulnerabilidades. Neste estudo foram encontradas sete ferramentas que podem apoiar a detecção de vulnerabilidades durante o processo de desenvolvimento.

Na segunda fase, é apresentada uma aplicação denominada EthSential. A aplicação foi desenhada e implementada de forma a integrar, inicialmente, as ferramentas de análise de segurança Mythril, Securify e Slither. A aplicação permite duas formas de uso, através da linha de comandos e através das extensões do Visual Studio Code. A aplicação foi publicada e disponibilizada publicamente através das ferramentas PyPI e Visual Studio Code.

Para avaliar a solução, foram realizados dois métodos de teste de software e um questionário de usabilidade e satisfação. Os resultados finais foram considerados positivos em termos de teste de software. No entanto, em termos de usabilidade e satisfação dos programados, os resultados não correspoderam às expectativas. Concluindo assim que algumas melhorias devem ser feitas no futuro para aumentar a satisfação dos programadores e a respectiva usabilidade da solução.

**Palavras-chave:** Ethereum, Smart Contracts, Solidity, Segurança, Análise, Vulnerabilidades

# Acknowledgement

I would like to express my deep and sincere gratitude to my advisor Professor Isabel Azevedo. I thank her for her patience and availability. Her immense knowledge and critical thinking process have been of great value for me and to this dissertation.

I'd also like to thank all my colleagues with whom I had the opportunity to share all these years that, in a way, helped me achieve this goal.

And finally, but not least, a special thanks to my family for all the support, patience and continuous encouragement throughout all my academic and professional life. Without them, none of this would be possible!

# Contents

# List of Figures

# List of Tables

# List of Source Code

# List of Acronyms

| | |
|---|---|
| AAA | Arrange-Act-Assert. |
| API | Application Programming Interface. |
| BDD | Behaviour Driven Development. |
| CLI | Command Line Interface. |
| CSAT | Customer Satisfaction Score. |
| DAO | Decentralized Autonomous Organization. |
| DSR | Design Science Research. |
| EOA | Externally Owned Accounts. |
| EVM | Ethereum Virtual Machine. |
| FEI | Front End of Innovation. |
| FFE | Fuzzy Front End. |
| IDE | Integrated Development Environment. |
| LSP | Language Server Protocol. |
| NCD | New Concept Development. |
| PC | Program Counter. |
| PoS | Proof of Stake. |
| PoW | Proof of Work. |
| QFD | Quality Function Deployment. |
| SDK | Software Development Kit. |
| SDLC | Software Development Life Cycle. |
| SWC | Smart Contract Weakness Classification. |
| VS Code | Visual Studio Code. |

# Chapter 1

# Introduction

## 1.1 Context

Since the creation of the first Blockchain technology in 2009, Bitcoin [6], the term Blockchain has increased its popularity in the community [7].

A Blockchain provides a "distributed software architecture where a network of participants that do not know or trust each other can establish agreements on shared states for decentralised and transactional data without the need of a central point of control or supervision" [8].

When dealing with financial aspects, identity management, supply chains, or crowdfunding, blockchain technologies are a newly emerging alternative gaining more ground in these areas. These emerging technologies provide different cybersecurity and optimisation, through the business process, with the integration of distributed resources connected using cryptography and cryptocurrencies.

Many companies, as well as governments, either directly or indirectly influenced by the community, have been considering the possibility of using a data structure based on Blockchain and starting to analyse its potentials [9]. A few number of companies already started using these technologies [10] in the last couple of years. However, despite the constant awareness and improvements made, companies still need to obtain further knowledge about these technologies to make informed decisions in their projects.

Currently, thousands of different blockchains are under development, and only a few blockchain technologies have reached the production stage and are ready to be used in real scenarios. Some of these technologies include the first blockchain Bitcoin, but also, Ethereum and Hyperledger Fabric.

An important use of these blockchains for developers is the creation of smart contracts. Smart contracts provide the use and manipulation of transactions in the form of contracts to exchange currencies, properties, share data, or initiate an operation. However, one of the major concerns in using smart contracts is its insecurity in their coding languages. A high number of attacks have been reported in smart contracts that exploited the vulnerabilities of their coding languages and blockchain networks [11].

Over the years, multiple programming languages have been proposed to implement smart contracts in Ethereum. Two remain active to this day, however some were deprecated and considered harmful to use. The most common and famous language used is Solidity [12], a contract-oriented language similar to JavaScript. The second and most recently created one is called Vyper [13], a python-like programming language with the intent to be a more secure and straightforward language than Solidity, offering a different approach to smart contracts development.

## 1.2 Problem Description

The increased amount of discussion on blockchain technology has brought a higher interest in smart contracts, particularly smart contracts on the Ethereum platform. As a program that runs on a blockchain, a smart contract can be executed by a network without the need of an external entity. One of the most important key characteristics of smart contracts is that once has been deployed, it cannot be modified or deleted, unless, as a last resort, it was programmed to behave in that matter. These characteristics can be seen as both an advantage and a disadvantage to the community and smart contract developers. One of the main disadvantages of an immutable contract is the adversity of having a faulty contract deployed on the network. An immutable faulty contract can become a huge risk to the application of the contract. In 2018, Nikolic et. al [14] analysed the 970,898 Ethereum smart contracts in the network of which 34,200 are vulnerable. Due to this risk, testing and auditing smart contracts before being deployed to the network becomes a high necessity to develop secure smart contracts.

The potential challenges that developers face when developing and testing smart contracts have not yet been clearly explored [15]. The lack of knowledge and awareness of best practices and tools to test and verify contracts during the development process is one of the challenges that should be addressed. Without understanding these challenges, practitioners and researchers may spend many efforts developing techniques and tools that are not appreciated by developers and so underused in practice [15]. There is not enough work focused on understanding the development from a developer/human perspective, and new tools should seek to explore this issue, facilitating a safer smart contract development [16].

Even though the community is gaining awareness of the exploits and tools available to analyse and test smart contracts [17] [18] [16], the work done for Ethereum languages, like Solidity and Vyper, is still not sufficiently well spread. Without knowledge of the best practices, developers can continue to develop insecure contracts and new attacks can occur.

The language Vyper was created to suppress some of the vulnerabilities of Solidity, but it is not sufficiently developed to be considered safe to use [19]. Security tools, like Slither, are working to support Vyper in their static analyser feature and vulnerability detectors [20]. However, there is still work to be performed in this area to verify this type of tools against the requirements of the current state of Solidity and Vyper.

## 1.3 Outline

To understand the development of smart contracts from the users perspective, it is necessary to analyse their practices and tools available to integrate into the process. An essential part of software development practices is the identification of vulnerabilities and code smells in the code. Following this identification, the main question of the dissertation is provided:

Is it possible to reduce the vulnerabilities and code smell in a smart contract code by integrating and combining tools in the development process?

## 1.4 Document Structure

This document is divided into 8 chapters, each referring to a specific part of the project's development consisting of several sections and subsections.

- Introduction - The first chapter introduces the context of the project, with a description of its problem and objectives.
- Background - Describes the key concepts of Blockchain and Ethereum that will help to understand the project.
- State of the Art - Represents the state of the art regarding the development and practices of Ethereum smart contracts, more specifically in Solidity and Vyper languages, using a literature review gathering data from multiple sources and databases.
- Value Analysis - Presents the analysis of the requirements and value of the project.
- Problem Statement - Describes the objectives with the main points of focus in this dissertation and the approach addressed to fulfil said objectives.
- Ethereum Development Research - The research meant to study the Ethereum development practices.
- Framework Design and Implementation - Describes the design approach and technical details of the solution with information relative to specific aspects of the system and architecture.
- Evaluation - Describes the necessary steps to evaluate the achievement of the objectives of the project and performs a case study with the evaluation steps to test the feasibility of the solution.
- Conclusions - In the final chapter, a summary of the overall project is presented with the description of achieved objectives, difficulties found during the project and possible future work.

# Chapter 2

# Background

The purpose of this chapter is to introduce an overview of essential topics for understanding the principles of Blockchain, Ethereum and smart contracts development.

## 2.1 Blockchain Fundamentals

The main idea of a Blockchain was first conceptualised in 2008 by Satoshi Nakamoto, in a white paper called "Bitcoin: A Peer-to-Peer Electronic Cash System" [21]. In his work, he described a chain of blocks as an "ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work" [21].

A blockchain, in simple terms, is a timestamped network of immutable transactions of data gathered together in ordered blocks, that are designed to be secured on the chain using cryptography. The blockchain data can be stored as a single file, a simple database, or a shared database distributed across multiple computers.

The blockchain structure is very similar to linked lists or binary trees structure. The blocks are linked to each other using pointers to the previous block in the chain. The first block is the foundation of the stack, called the genesis block, that must be hardcoded at the same time that the Blockchain was started. When a block is published in the chain, its data cant be changed, and it becomes publicly available to any user in the network.

Each block in the Blockchain is a set of transactions uniquely identified by a block hash or block header. The block header is composed of a hash to the previous block; timestamp is a reference to the time the block was created; nonce represents a number generated and used only once and the Merkle root that represents the root hash of all transactions included in the block. The structure of a block is illustrated in Figure 2.1.

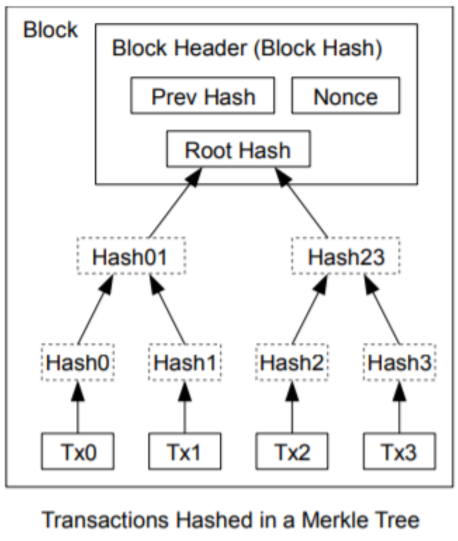


Figure 2.1: Block structure [21].

A transaction is a record of an event represented by a hash, with inputs and outputs depending on the type of Blockchain in use. When a transaction is executed on the Blockchain, it contains the private key used by the sender with instructions to operate. This private key is used to digitally sign the sender and ensures that the data of the transaction was not tempered during the process.

### 2.1.1 Types of Blockchain

There are three types of Blockchain [22][23]:

- Public Blockchain

  Anyone who has access to the internet can sign-in in a public blockchain and be authorised to participate as a node in the decision-making process. All transactions that occur on the network are completely opened, meaning that anyone can view the transaction details. Bitcoin and Ethereum are examples of public Blockchains.

- Private Blockchain

  A private Blockchain is a restrictive Blockchain that is used only in a closed network. In this network, the participants need consent to join the networks, and the transactions are only visible to them. There are many examples of this type of Blockchain, like HydraChain, Corda, and Hyperledger fabric.

- Hybrid/Semiprivate Blockchain

  It is a combination of both types of private and public Blockchain. It allows a user to control the access to the network and gives the flexibility to choose the data they want to be public, keeping the remaining part of the data restricted. There is one example of this type, called Dragonchain.

### 2.1.2 Consensus Algorithms

As mention in section 2.1, a transaction needs a verified digital signature to be able to be executed. This verification uses a process called mining, where the transaction sent to miners can digitally sign the transaction and verify if the transaction meets the requirements to be added to the network. These requirements are made upon a consensus mechanism that is responsible for the task. To incentive miners to verify transactions, after completing the verification, a reward is given to them in the currency of the network. Once the consensus is achieved from all the networks, the transaction can be added to the Blockchain.

There are many different types of consensus algorithms, in which the two most famous are:

- Proof of Work

  The initial concept of Proof of Work (PoW) was developed in 1993 when Cynthia Dwork and Moni Naor published a paper On "Memory-Bound Functions For Fighting Spam".[24] In their paper, they theorised that "If I do not know you and you want to send me a message, then you must prove that you spent, say, ten seconds of CPU time, just for me and just for this message." [24]. However, the concept was only applied to Blockchain in 1999 when a paper called "Proofs of work and bread pudding protocols "Markus Jakobsson and Ari Juels published [25]. This means that when a transaction is sent to the network, it relies on proof that significant computational resources need to be spent before being valid in the network.

  To prove this concept, miners need to solve a complex computational problem and consume resources so that once a miner finds the solution, it will be broadcast to the other miners in the network. The other miners will then verify that the solution is correct, and the transaction will be confirmed and saved in the network.

  The number of resources required to solve a computational problem can be costly and time-consuming and produce only a single piece of data. The second problem of this concept is centralisation. If a company can have a mining pool with hundreds of CPUs will have a better chance of solving the problem, then someone who only has one CPU. This means that the network will be more centralised in the mining pool, going against the fundamentals of the Blockchain.

- Proof of Stake

  The concept of Proof of Stake (PoS) was created as an alternative to PoW to prevent high costs of resource consumption and mining equipment. The concept is based on the idea of a voting system that depends on a validator's economic stake in the network [25].

  It designates that a validator with a high balance or more extended time in the network has a higher advantage to validate the next block. "The validator takes turns proposing and voting on the next block, and the weight of each validator's vote depends on the size of its deposit (i.e., stake)" [3]. In this concept, the validators do not receive a reward for validating the block. Instead, they collect transaction fees, and therefore they must own and support the currency they are verifying. Comparing to PoW, the PoS can have a lower risk of centralisation and resource consumption.

  Ethereum is one of the networks that will use Casper, a proof-of-stake-based consensus protocol.

Other consensus algorithms are commonly used in the Blockchain like Proof of Elapsed Time, Proof of Deposit, Proof of Importance, and Proof of Activity.

## 2.2 Ethereum

Ethereum was introduced by Vitalik Buterin's paper [26] in 2013. Ethereum intends to be a public distributed computing platform with an embedded Turing complete system that allows creating, store, and run smart contracts decentralised applications. "Ethereum can create arbitrary rules for ownership, transaction formats, and state transition functions" [26]. In more simple terms, it means that the Ethereum blockchain is a set of connected blocks that contain code or programs that can be executed without requiring a trusted entity to execute them.

The following subsections discuss the essential aspects of Ethereum.

### 2.2.1 Turing Complete

"The Turing complete machine is the theoretical mathematical machine or model of computation invented by Alan Turing in 1936 that defines an abstract machine that manipulates symbols on a strip of tape according to a table of rules." [27]

"Alan Turing further defined a system to be Turing complete if it can be used to simulate any Turing machine, and it called it a Universal Turing machine (UTM)." [3]

Therefore Ethereum's ability to function as a computer that can execute any program, in a state machine called the Ethereum Virtual Machine, given the limitations of finite memory and time, makes it a Turing complete system [3].

### 2.2.2 Ethereum Virtual Machine

Ethereum Virtual Machine (EVM) is a runtime environment engine that handles smart contracts deployment and execution [3] by decoding the compiled contracts in bytecodes and executing them on the Ethereum network [27]. It is an isolated engine from the network, and so it has limited access to smart contracts. The execution model of EVM is presented in the following Figure 2.2:

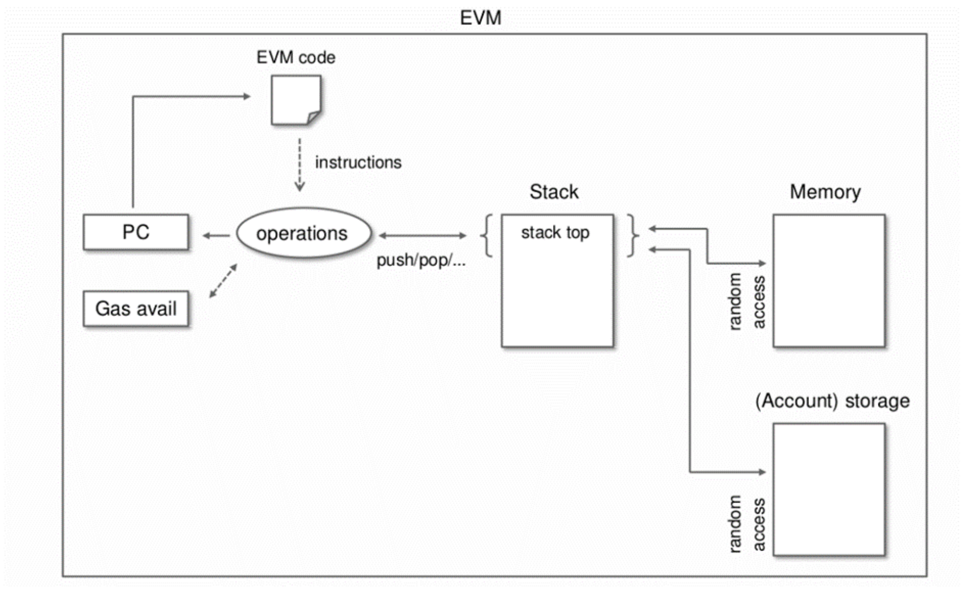


Figure 2.2: EVM execution model. From [28]

Every time a code is executed or transaction is created in EVM, an amount of gas and Program Counter (PC) is calculated. Gas is the unit cost of needed to execute a single operation on the Ethereum network and a PC tracks the current operation in its program sequence. The amount of gas defined on the transactions is used to limit the amount of computational power that can be executed per program in the EVM. So, the more gas used in a program, the faster it will be executed.

Once the contract is called, the EVM will load the bytecode instructions, called opcodes, into memory, loop through each bytecode instruction while checking the amount of gas and incrementing its program counter, execute all operations and store them on the EVM stack. "The size of every transaction on the EVM stack is 256 bits with a maximum size of 1024." [29]

### 2.2.3 Accounts

The Ethereum network has two types of accounts, Externally Owned Accounts (EOA) and Smart Contract Accounts. EOA is associated with an external entity controlled by a public-private key pair. The Smart Contract Accounts, have associated code that is kept in storage, and are triggered by transactions or messages received from other contracts.

Accordingly to Ethereum Yellow Paper [30], all accounts in Blockchain must have the following four elements:

- The nonce, representing the number of transactions or contract-creations made from the account.
- The balance, the number of *wei* owned by an account.
- The *storageRoot*, a 256-bit hash of the root node of the storage contents of the account, called the storage tie.

- The *codeHash*, The hash of the EVM code of this account. For EOA this code will be empty.

### 2.2.4 Transactions, Messages, and Ether

A transaction in terms of the global Blockchain was explained in Section 2.1. However, for Ethereum, there are some differences. For Ethereum, a transaction refers to the signed data package that stores a message to be sent from an Externally Owned Accounts (EOA) to another account (EOA or smart contract) [31].

There are two types of transactions: those who can send message calls to other contracts and those which result in the creation of new accounts with associated code (smart contracts) [30]. Unlike transactions, messages are virtual objects and will not be recorded.

Ether (ETH) is the currency of the Ethereum network, used to pay transaction fees. Each time a smart contract is deployed in the network, the account must pay in Ether when they call a transaction or a smart contract. The smallest sub denomination of Ether, and thus the one in which all integer values of the currency are counted, is the Wei, one Ether is defined as $10^{18}$ Wei [30].

### 2.2.5 Ethereum Improvement Proposals

"An EIP is a design document providing information to the Ethereum community, or describing a new feature for Ethereum or its processes or environment." [32] It can be proposed by any Ethereum community member and then discussed by them. There are three types of EIPs [32]:

- *Standard Track EIP*

  *Describes any change that affects most or all Ethereum implementations, such as a change to the network protocol, a change in block or transaction validity rules, proposed application standards/conventions, or any change or addition that affects the interoperability of applications using Ethereum. Furthermore, Standard EIPs can be broken down into the following categories. Standards Track EIPs consist of three parts, a design document, implementation, and finally, if warranted an update to the formal specification.*

- *Meta EIP*

  *Describes a process surrounding Ethereum or proposes a change to (or an event in) a process. Process EIPs are like Standards Track EIPs but apply to areas other than the Ethereum protocol itself. They may propose an implementation, but not to Ethereum's codebase; they often require community consensus; Examples include procedures, guidelines, changes to the decision-making process, and changes to the tools or environment used in Ethereum development.*

- *Informational EIP*

  *Describes an Ethereum design issue, or provides general guidelines or information to the Ethereum community, but does not propose a new feature. Informational EIPs* *do not necessarily represent Ethereum community consensus or a recommendation, so users are free to ignore Informational EIPs or follow their advice.*

### 2.2.6 Current State

Ethereum is under continuous development and new improvements are constantly being made by the community. Although Ethereum launched in 2015, it initially planned to be developed in four stages, as illustrated in Figure 2.3: Frontier; Homestead; Metropolis (Byzantium and Constantinople); Serenity.

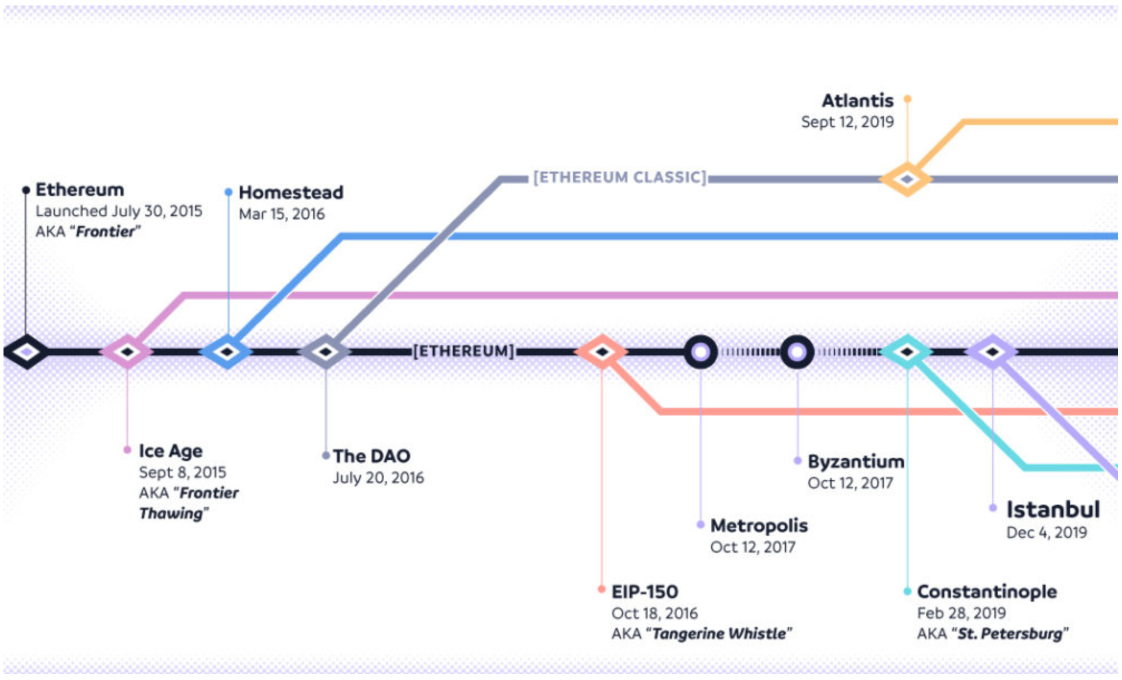


Figure 2.3: Ethereum roadmap. From [33]

Each phase is defined by a set of Ethereum Improvement Proposals (EIPs) that were accepted by the community.

The Frontier was the first release of the network in 2015 and allowed users to mine Ether, build simple centralised contracts, and make exchanges while helping the network to fix the errors.

The second stage was the Homestead upgrade, which was the first hard fork of the Ethereum network, leaving behind the frontier chain. It provided the resolve of the flaws found in Frontier and removed the centralisation of the contracts. Also, it provided improvements to the main protocols and introduced a wallet that allowed users to transfer Ether and deploy smart contracts.

After the release of Homestead, the Decentralized Autonomous Organization (DAO) attack occurred. The DAO was a smart contract that intended to be venture capital fund, based on Ethereum Solidity language. In June 2016, one hacker noticed the flaws of the DAO's code and stole 3.6 million Ether into an account similar to DAO. Ethereum decided to create a hard fork to refund all the Ether that was stolen in the DAO. Those who didn't agree with the Hard fork created the Ethereum Classic.

Metropolis was divided into two releases, Byzantium and Constantinople. Byzantium had a total of nine EIPs to improve the network's privacy, scalability, and security. Constantinople hard fork update provided five EIPs to add efficient alternatives for some of the functions on the Ethereum blockchain.

Not planned from the beginning was the Instanbul fork, which was created in December 2019 with a total of six EIPs. This fork introduces more privacy and scaling capabilities to Ethereum as well as a significant rebalancing of the gas pricing with the computation costs of the EVM opcodes [34].

The final planned Serenity stage is in development and is guided by five design principles: Simplicity, Resilience, Longevity, Security, and Decentralisation [35]. As a PoW blockchain, there was is a long-term plan to transition to a PoS in the Serenity stage.

Ethereum is currently implementing the Casper protocol, based on proof of work concept. Accordingly, to Ethereum documentation [25], Casper has two "flavours" of proof of stake: chain-based proof of stake and BFT-style proof of stake. In Chain-based proof of stake algorithms, the validators are selected pseudo-randomly to create the next block pointing to the previous block in the chain. In BFT-style proof of stake, the act of suggesting the next block and creating the next block are separated, so validators that suggest the next block are selected randomly.

## 2.3 Smart Contract

Smart Contracts accounts, mentioned in Section 2.2.3, can be executed using the code saved in storage, by calling a function containing the contract name. Its parameters are binary encoded and sent to the contract in the data field of the transaction.

In 1994, Nick Szabo introduced the term smart contract, idealising a digital representation of a traditional contract, described as follows: "A smart contract is an electronic transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimise exceptions both malicious and accidental, and minimise the need for trusted intermediaries." [36]

There have been several definitions of a smart contract over the years. However, the first rule of a smart contract is that it must be able to be programmed to perform any actions that blockchain users need and according to their specific business requirements [37]. Once a smart contract is deployed, its code and state are publicly available and cannot be changed, representing a block in the chain, as mentioned in Section 2.1.

Smart Contracts code in Ethereum can be written in many programming languages. The two main languages that will be analysed are Solidity and Vyper. However, other programming languages were created to be used in Ethereum EVM. These languages are:

- LLL

Lisp Like Language (LLL) is a low-level language similar to Assembly. [38] It behaves like a wrapper over coding in EVM directly, with direct access to memory and storage. It is not maintained since 2018.

- Serpent

  The Serpent was a high-level programming language inspired in Python, with a simple and minimal syntax, dynamic typing, and support for object-oriented programming [39]. In 2017, it was deprecated after an audit, claiming that it has a very low quality, untested, very little documentation, and flawed language design. [40]

- Mutan

  Mutan was a language targeting the EVM, offering a C like syntax [41]. Mutan has been deprecated since March 2015.

- Bamboo

  Bamboo was first released in 2017, and it is a smart contract language oriented to state transitions and with no iterative code involved [3]. It was created to avoid reentrancy problems by default, using each function declared within a state [42].

- Flint

  Created in 2018, Flint is a type-safe, capabilities-secure, contract-oriented programming language designed with novel contract-oriented features, such as caller capabilities, Asset types, and mutation functions [43].

### 2.3.1 Solidity

Solidity [12] was introduced to replace all three original languages (Mutan, Serpent, and LLL), "created by Dr Gavin Wood as a language explicitly for writing smart contracts with features to directly support execution in the decentralised environment of the Ethereum world computer". [3] It is a versioned statically typed language with a contract-oriented architecture inspired in JavaScript and C++. At the time of writing this document, Ethereum is in version 0.6.2.

#### Structure of a Solidity Program

Every Solidity program (shown in example listing 2.1) can have any number of contracts/libraries coded in a single file or multiple source files. The file extension for solidity code is *.sol*, and it must contain the solidity version to use (line 1) and the contract/library code (lines 4 - 14). It can also contain an import statement to import other sources files (line 2).

```
pragma solidity >=0.5.0 <0.6.0;
import "../mortal/mortal.sol";

contract Greeter is Mortal {
    string greeting;

    constructor(string memory _greeting) public {
        greeting = _greeting;
    }

    function greet() public view returns (string memory) {
        return greeting;
    }
}
```

Listing 2.1: Example of a contract in Solidity. [1]

An object-oriented language, in general, can contain declarations of Data Types, Variables, Control and loop structures, Functions and Objects. In Ethereum, the case is similar. "Each contract can contain declarations of State Variables, Functions, Function Modifiers, Events, Struct Types, and Enum Types" [44]. It can also define unique kinds of contracts called libraries that are deployed only once, along with the contract.

### State Variables and Data Types

The State Variables defined in any contract are values that are permanently stored in the contract storage. The variables can be used at multiple places within the code and will refer to the same value stored. Each state variable has a statically data type defined so that the Solidity compiler can allocate the memory. The most used data types are [3]:

- *Boolean (bool): can store boolean values, true or false, with logical operators*
- *Integers (int, uint): Include all subtypes of signed(int) and unsigned(uint) integers*
- *Fixed point (fixed, ufixed): Fixed-point numbers, declared with (u)fixedMxN where M is the size in bits (increments of 8 up to 256) and N is the number of decimals after the point (up to 18)*
- *Address: An Ethereum account address*
- *Fixed byte arrays: Include static byte arrays that hold a sequence of bytes from one to up to 32*
- *Dynamic byte arrays: Includes the dynamic byte array and the string type, a dynamically sized UTF8-encoded string.*
- *Enum: User-defined type for enumerating discrete values*
- *struct: User-defined data containers for grouping variables*
- *Array: An array of any type, either fixed or dynamic*

Solidity provides a set of functions and variables that can be globally accessed in contract execution. These variables are used to provide information about the state of the blockchain or the contract. The most used are the *block*, *msg*, and *tx* objects. The first one contains the information about the block; the second provides information about the message sent to a contract; the third and final object provides information about the transaction.

### Access Modifiers

There are four different types of access modifiers for functions and variables in Solidity [3]:

- *Public: It can be called by other contracts or EOA transactions, or from within the contract.*
- *Private: Only visible internally by the declaring contract and not by derived contracts.*
- *Internal: Only accessible within the contract, cannot be called by another contract or EOA transaction*
- *External: Like public functions, except it cannot be called within the contract unless explicitly prefixed with the keyword this in the code.*

### Functions, Modifiers and Fallbacks

Solidity functions are used to create transactions and implement a certain logic. A function can take multiple parameters and return multiple values. It uses the access modifiers to specify the visibility, and it can be one of the three types: pure, view, or payable. These types define the scope of changes allowed within the Ethereum global state.

A view function is used to view state variables of the contract and cannot change them. A pure function ensures that it cannot read or modify a state variable. The compiler will throw an error in case the pure function tries to perform any operation on state variables. Finally, the payable function, that provides a mechanism to receive funds in Ether.

The syntax we use to declare a function in Solidity is as follows [3]:

*function FunctionName([parameters]) public|private|internal|external [pure|constant|view|payable] [modifiers] [returns (return types)]*

When a Solidity contract is compiled into EVM bytecode, its functions are identified by a signature derived from its name and arguments. When a function is invoked, and no matching function can be found, or no signature is provided at all, the fallback function of a contract is invoked instead [45].

A fallback function cannot be called explicitly, doesn't have a name and it cannot accept any arguments or return any value. An example of a fallback function is present in the following listing 2.2:

```
pragma solidity 0.5.0;

contract Test {
    function() external { x = 1; }
    uint x;
}
```

Listing 2.2: Example of a fallback function in Solidity. [2]

Function modifiers can be used to modify the behaviour of a function. They check that a function can only be executed in a specific context. For instance, to create a modifier that allows only the contract owner to run a given function, like shown in Listing 2.3:

```solidity
pragma solidity 0.5.0;

contract Owner {
   address owner;
   constructor() public {
      owner = msg.sender;
   }
   modifier onlyOwner {
      require(msg.sender == owner);
      _;
   }
   modifier costs(uint price) {
      if (msg.value >= price) {
         _;
      }
   }
}
```

Listing 2.3: Example of a function modifier in Solidity. [3]

### Events

Every transaction in Ethereum keeps the information about events that occurred. When an event is called, its arguments are stored in the transaction log data structure of the blockchain. The log keeps the information of the contract address, but the data is not accessible from any contract. Solidity can create events by declaring the keyword event, followed by an identifier and any number of parameters to send. An event can index up to three parameters by adding the keyword indexed in the parameter defined. An example of a contract creating an event is shown in listing 2.4:

```solidity
pragma solidity 0.5.0;

contract SmartExchange {
    event Deposit(address from, bytes32 to, uint indexed  value);
    event Transfer(bytes32 from, address to, uint indexed value);

    function deposit(bytes32 to) payable public {
        emit Deposit(msg.sender, to, msg.value);
    }

    function transfer(bytes32 from, address payable to, uint value)
     payable public{
        to.transfer(value);
        emit Transfer(from, to, value);
    }
}
```

Listing 2.4: Example of a event in Solidity. [4]

### 2.3.2 Vyper

Vyper is a pythonic smart contract language that runs on top of the EVM. It follows the principles that should be simple to build a secure smart contract and with an auditable code [46]. Vyper is currently in beta version v0.1.0-beta.16, and new developments are being made to release the language.

#### Structure of a Vyper Program

Comparing to Solidity, smart contracts in Vyper are written in a single file. A contract in Vyper can have state variables, functions, decorators, structs, and events.

Listing 2.5 shows an example of a simple crowdfund written in Vyper that allows other contracts to participate, refund everyone, and end the crowdfunding.

```
struct Funder :
  sender: address
  value: wei_value
funders: map(int128, Funder)
nextFunderIndex: int128
beneficiary: address
deadline: public(timestamp)
goal: public(wei_value)
refundIndex: int128
timelimit: public(timedelta)

@public
def __init__(_beneficiary: address, _goal: wei_value, _timelimit: timedelta):
    self.beneficiary = _beneficiary
    self.deadline = block.timestamp + _timelimit
    self.timelimit = _timelimit
    self.goal = _goal

@public
@payable
def participate():
    assert block.timestamp < self.deadline, "deadline not met (yet)"
    nfi: int128 = self.nextFunderIndex
    self.funders[nfi] = Funder({sender: msg.sender, value: msg.value})
    self.nextFunderIndex = nfi + 1

@public
def finalize():
    assert block.timestamp >= self.deadline, "deadline not met (yet)"
    assert self.balance >= self.goal, "invalid balance"
    selfdestruct(self.beneficiary)

@public
def refund():
    assert block.timestamp >= self.deadline and self.balance < self.goal
    ind: int128 = self.refundIndex
    for i in range(ind, ind + 30):
        if i >= self.nextFunderIndex:
            self.refundIndex = self.nextFunderIndex
            return
        send(self.funders[i].sender, self.funders[i].value)
        clear(self.funders[i])
    self.refundIndex = ind + 30
```

Listing 2.5: Example of a contract in Vyper. [5]

As presented in Listing 2.5, the struct Funder with the variables sender and value is defined on line 1. Vyper is a scripting language and therefore does not have classes or objects. Instead, it includes structs, similar to C language.

Following the example, there are several state variables defined (lines 4 - 10). The state variables, like most languages, can be a value type or a reference type, and the syntax is very similar to Solidity.

There are four functions in the example:

- The __init__ function (line 13) that initialises the contract.
- The function participate (line 21) annotated with the @payable decorator that receives funds and add them to the crowdfund.
- The finaliser function (line 28) that ends the crowdfunding and removes the contract from the network (line 31).
- Finally the refund, that sends an Ether stored in the contract to the designated addresses (line 41).

A decorator in Vyper can be one of the following [46]:

- @public - Can only be called externally.
- @private - Can only be called within the current contract.
- @constant - Does not alter the contract state.
- @payable - The contract is open to receive Ether.
- @nonreentrant(<key>) - Function can only be called once, both externally and internally. Used to prevent reentrancy attacks.

In Vyper, events must be declared before any global declarations and function and can include up to three indexed variables. The syntax for executing an event is shown in Listing 2.6.

```
# Financial events the contract logs
Transfer: event({_from: indexed(address), _to: indexed(address), _value: uint256(currency_value)})
Buy: event({_buyer: indexed(address), _buy_order: uint256(currency_value)})
Sell: event({_seller: indexed(address), _sell_order: uint256(currency_value)})
Pay: event({_vendor: indexed(address), _amount: wei_value})

# Initiate the variables for the company and its own shares.
company: public(address)
totalShares: public(uint256(currency_value))
price: public(uint256 (wei / currency_value))

# Store a ledger of stockholder holdings.
holdings: map(address, uint256(currency_value))
```

Listing 2.6: Example of an event's declaration in Vyper. [5]

## 2.4 Code Smells

A code smell is a hint which indicates that program code or design could have symptoms of a problem. The concrete definition of a code smell varies from project to project and has multiple points of view. The quality of a program can be quantified by the number of occurrences of code smells [47]. On the opposite, clean code is a practice that aims to write readable, functional, small, simple code, easy to understand and maintain. It must be easily accessible to others, with a clear intention, unambiguous, right abstractions, proper names for units, methods, classes, forms, data modules. There are some common signs of code smells:

- Bloaters: Classes, methods or parameters that have grown to proportions difficult to manage.
- Object-Orientation Abusers: The use of incorrect application of switch statements, temporary fields or inheritance.
- Change Preventers: When a piece of code needs to be changed, and it is necessary to make the restructuring of a high amount of places in the code.
- Dispensables: Code that is not being used or not necessary for the problem.
- Couplers: Use of excessive coupling or excessive delegation between classes.

Code smells are related to security issues and design flaws because they could increase the risk of bugs or failures in the system [48]. One of the techniques used to reduce the code smell in the system is refactoring the code or even the design. Refactoring is the process of changing a system in such a way that it does not alter the behaviour of the code and improves its internal structure [49].

## 2.5 Coding Standards

Coding standards are collections of coding rules, guidelines, and best practices to write cleaner and safer code. A coding standard attempts to ensure that all the developers of a given project, organisation, or even community, are following specific guidelines, and the code can be easily understood, maintaining its consistency [50]. Using a coding standard can reduce the risk of having errors in code and increase its readability and maintainability. The most used programming languages have adopted or exemplified coding standards. The most commonly used are:

- Documentation: writing the explanation or annotation of source code in the format of comments in the source code or document.
- Indentation: convention used for the source code structure stating the space between the beginning of a line and the code.
- Naming: there are two naming conventions for functions, variables and other properties, CamelCase where the first letter of each word is capitalised except for the first word and UnderScore used between words.
- Avoided deep nesting structure: The nested structures are difficult to read and understand.

- Don't Repeat Yourself (DRY) principle: aimed at reducing repetition of information and code.
- Keep It Simple, Stupid (KISS) principle: To keep the code simple and transparent, making it easy to understand.

## 2.6 Summary

Ethereum Smart Contracts are elements of the Ethereum network that can operate in numerous contexts and have multiple coding languages supported by an interacting development community, namely Solidity and Vyper.

Solidity (2.3.1) and Vyper (2.3.2) are very distinct languages. The development of smart contracts in these languages is not simple. Ethereum languages are new and highly experimental, and new challenges need to be taking into account [51]. The software and network behind Ethereum is continually changing and upgrading, but smart contracts are different and cannot upgrade, due to their immutable.

While Solidity has a language very similar to an object-oriented language, Vyper focuses on the development of smart contracts with increased security and focus on protecting developers' code from exploits. It was designed after a number of various stages of Solidity development and learned from its mistakes. A comparative table between Solidity and Vyper regarding released versions and contracts deployed can be seen in table 2.1.

Table 2.1: Comparison between Solidity and Viper

| Language | First Release (Date - Version) | Current Version |
|---|---|---|
| Solidity | 21 Aug 2015 - v0.1.2 | v0.6.3 |
| Vyper | 23 Mar 2018 - v0.0.4 | v0.1.0-beta.16 |

As stated in the previous table 2.1, Solidity is in a later stage of development with a much higher number of contracts deployed on the blockchain.

# Chapter 3

# State of the Art

This chapter presents an exhaustive study of the state of the art of technologies and relevant information regarding the Ethereum development in Solidity and Vyper.

## 3.1 Methodology

This Section describes the research plan for the literature review to address the practises in Solidity and Vyper, through an analysis of scientific and academic publications/studies and tools developed by the Ethereum community.

There are two possible literature review approaches identified for the research plan:

- Systematic literature review (SLR): "A systematic review is a technique that answers a defined research question by collecting and summarising all the available evidence regarding a specific area that meets the eligibility criteria" [52].
- Traditional or Narrative literature review (NLR): Narrative reviews take a less formal approach than systematic reviews, do not require the presentation of rigorous aspects characteristic of a systematic review [53].
- Meta-analysis Review: "Technique that statistically combines the results of quantitative studies to provide a more precise effect of the results" [54].

The approach adopted in this research is the meta-analysis review. With this type of research, it is possible to gather multiple data studies and combine them with more recent development in smart contracts. The selection of the data studies is chosen based on multiple criteria relevant to identify possible vulnerabilities and patterns following the best practices and development approaches. The gathered data is synthesised following specific criteria, analysed and evaluated against their usage in smart contracts development. A meta-analysis consists of the following phases:

- Define the research question and literature review (Section 3.2)
- Extract and Analyse Data (Section 3.3)
- Present results (Section 3.4)

## 3.2 Research Questions

Writing a research question is the first step in conducting a meta-analysis review. Following the main question of this dissertation provided in Section 1.3), the following questions were identified:

- [Knowledge] What are the most common vulnerabilities and code smells detected and reported in smart contracts?

  In this first question, the purpose is to find precise taxonomies of vulnerabilities, bugs, and code smells with its definitions, examples and implications on the development of smart contracts.

- [Availability] What is the state of tools used in the development of Ethereum smart contract?

  Identify and categorise available and usable tools that support the development of smart contracts.

- [Quality] Which design patterns and anti-patterns are documented and exemplified in smart contracts?

  Obtain common illustrated design patterns and respective anti-patterns that successfully identify the problems and solutions for the patterns.

- [Effectiveness] Which of the vulnerabilities, bugs, and code smells can be detected by tools?

  For this question, the interest is to determine which tools can detect vulnerabilities in vulnerable smart contracts.

## 3.3 Data Collection Analysis

The research was conducted on several online databases, such as Google Scholar, IEEE, ResearchGate, ACM DL, Solidity/Vyper documentation, and "grey" literature (eg., blogs, books, dissertations and video conferences).

The process of selecting the appropriate studies is defined by following Inclusion/Exclusion criteria along with the search strings. The defined I/E criteria are described in Table 3.1.

Table 3.1: Inclusion and Exclusion criteria

| Inclusion | Exclusion |
|---|---|
| Reports referencing tools, design patterns, code smells, and practices related to Ethereum Solidity or Vyper | Reports not related to Ethereum |
| Published after 2015 to June 2020 | Before 2015 |
| Published or unpublished but publicly available | Private or Unavailable |

To conclude the selection of the appropriate studies, the search string is simply defined to obtain the most important data following the research questions and I/E criteria:

*ethereum AND smart AND contract? AND (solidity OR vyper)*

As a result of the selection and screening of studies following the selected search string and I/E criteria, a total of 103 documents were identified from online databases and 20 documents identified from other sources.

The analysis of data collected is completed with a quality assessment of the finds. This assessment breaks down the relative information of the findings focusing on answering the research questions defined (Section 3.2).

The main documents found in the data collection analysis process are presented in table 3.2.

Table 3.2: Data collection analysis documents

| Identifier | Date | Summary |
|---|---|---|
| 1 | 2016 | Luu et al. [55] Investigated the security of running smart contracts in Ethereum and proposed ways to enhance the operational semantics of Ethereum to make contracts less vulnerable by building a symbolic execution tool called Oyente to find potential security bugs. |
| 2 | 2017 | Dika [17] proposed a categorised taxonomy of known security issues by inspecting analysis tools (Oyente, SmartCheck, Remix and Securify) and proposing a taxonomy with the categorisation of that tools. |
| 3 | 2017 | Atzei et al. [56] surveyed attacks on Ethereum smart contracts and defined a taxonomy for Ethereum smart contract vulnerabilities. |
| 4 | 2018 | Chen et al. [57] researched on under-optimised smart contracts and identified 24 anti-patterns from the execution traces of real smart contracts. They developed a tool called GasReducer, to automatically detect anti-patterns from the bytecode of smart contracts and replace them with efficient code through bytecode-to-bytecode optimisation. |
| 5 | 2018 | Parizi et al. [58]conducted an empirical study of security analysis tools against vulnerability detection in Ethereum smart contracts. The result show that SmartCheck is the most effective tool while Mythril is the most accurate one. |
| 6 | 2018 | Wöhrer and Zdun [59] analysed security patterns for Solidity, by gathering data from different sources and applied Grounded Theory techniques to extract and identify the patterns. In total, they identified six design patterns that lack execution control once a contract is deployed, resulting from the distributed execution environment provided by Ethereum. |
| 7 | 2018 | Antonopoulos et al. [3] referenced some security best practices, anti-patterns and various vulnerabilities encountered by smart contract developers. |
| 8 | 2018 | Mense and Flatscher[60] compiled a taxonomy of vulnerabilities and compared 7 analysis tools against all of the vulnerabilities in the taxonomy. The study showed that SmartCheck, Securify and Mythril identified the higher number of vulnerabilities and that Remix and Security focus only on severe vulnerabilities. |
| 9 | 2018 | Liu et al. [61] summarised smart contract design patterns based on existing smart contracts and classified them into the Design Patterns categories: Creational Patterns, Structural Patterns, Inter-Behavioral Patterns,and Intra-Behavioral Patterns |
| 10 | 2018 | Wöhrer and Zdun [62] conducted an analysis based on a Multi-vocal Literature Research and identified 18 design patterns, grouped into five categories (Action and Control Patterns, Authorisation, Lifecycle, Maintenance and Security) and described them in detail with exemplary code for illustration. |

| | | |
|---|---|---|
| 11 | 2019 | Coblenz et al. [16] analysed different blockchain publications and considers that a new generation of blockchain software development tools should focus on users' needs, seek to facilitate safe development, by detecting relevant classes of severe bugs at compile-time, and be blockchain-agnostic. |
| 12 | 2019 | Grundy et al. [48] conducted an empirical study on Stack Exchange posts [63] related with smart contracts code smells. As a result, 20 types of code smells in smart contracts were defined and categorised in security, architecture, and usability. |
| 13 | 2019 | Murray and Anisi [64] surveyed the state of the art of formal verification of smart contracts and identified several methods and approaches for performing formal verification. They also consider that, at the time of the publication, there were no established standard or best practice for the development of smart contracts. |
| 14 | 2019 | Demir et al. [65] reviewed 28 security smells in deployed smart contracts and categorised them by their context: in the smart contract's execution environment, design and coding. |
| 15 | 2019 | Di Angelo and Salzer [66] surveyed the various tools for analysing Ethereum smart contracts. They concluded that there are three types of tools: academic tools, tools developed by companies, and community tools in open repositories. They surveyed 27 considered tools regarding availability, maturity level, methods employed, and detection of security issues, regardless of their provenance and by installing and testing them. From this survey they highlighted 5 tools: FSolidM, KEVM, Securify, MAIAN and Mythril. |
| 16 | 2019 | Sierra [67] extended the verification of smart contracts by developing a formal verifier for smart contracts written in Vyper. The verifier introduces a new specification construct to give guarantees about the execution of a contract even in the presence of reentrancy problems, the correct handling of funds, and the absence of security problems. |
| 17 | 2019 | Gupta [68] analysed Ethereum smart contracts from a security perspective, by studying multiple security vulnerabilities and issues, and developing a taxonomy with the results. Then, analysed the security tools with the taxonomy to test their effectiveness against the vulnerabilities. |
| 18 | 2020 | Durieux et al. [69] created a framework to analyse and compare multiple analysis tools, called SmartBugs. The tool compared 9 analysis tools against a dataset of Vulnerable Smart Contracts. The results show that the tools with most detected vulnerabilities per category defined are Mythril, Slither, Oyente and Osiris. |
| 19 | 2020 | Kaleem et al. [70] analysed Vyper and presented a survey to assess if common vulnerabilities in Solidity can be translated to Vyper development. They analysed the vulnerabilities and categorised them in 5 groups. |

| | | |
|---|---|---|
| 20 | 2020 | Gao et al. [71] proposed and implemented an automated approach, named SMARTEMBED, to provide clone detection, bug detection and contract validation on Solidity smart contracts. They tested their tool in 22,000 smart contracts collected from the Ethereum blockchain and showed that their tool can effectively identify a clone ratio around 90% and more than 1000 clone-related bugs. |
| 21 | 2020 | Praitheeshan et al. [18] investigated 16 security vulnerabilities in smart contracts and categorised them in terms of static analysis, dynamic analysis, and formal verification. By correlating the vulnerabilities and 19 software security issues compared the three analysis methods in terms of their performance, coverage of finding vulnerabilities and accuracy. |

## 3.4 Results

As result of the data collection obtained from the collected documents, the following categories were identified:

- Security vulnerabilities - A classification of commonly identified vulnerabilities in the Ethereum network with the development of smart contracts.
- Design Patterns - A list of design and programming patterns for smart contract programming.
- Tools - A classification of analysis, development and testing tools in smart contract development.

### 3.4.1 Security Vulnerabilities

Based on the proposed EIP-1470, the most common weakness of smart contracts have been identified since October 2018, in a Smart Contract Weakness Classification (SWC) web page [72] publicly available and opened for contribution. The overall classification for security vulnerabilities were extracted from the SWC Registry and compared with other sources [73] [65] [74] [60]. The vulnerabilities that were present in other sources and didn't matched the vulnerabilities of the SWC registry were added to the classification. The complete classification of the security vulnerabilities with respective description and recommendation is present in Appendix A. The classification contains the following registries:

- SWC 100 - Function Default Visibility
- SWC 101 - Integer Overflow and Underflow
- SWC 102 - Outdated Compiler Version
- SWC 103 - Floating Pragma
- SWC 104 - Unchecked Call Return Value
- SWC 105 - Unprotected Ether Withdrawal
- SWC 106 - Unprotected SELFDESTRUCT Instruction
- SWC 107 - Reentrancy
- SWC 108 - State Variable Default Visibility

- SWC 109 - Uninitialised Storage Pointer
- SWC 110 - Assert Violation
- SWC 111 - Use of Deprecated Solidity Functions
- SWC 112 - Delegatecall to Untrusted Callee
- SWC 113 - DoS with Failed Call
- SWC 114 - Transaction Order Dependence
- SWC 115 - Authorisation through tx.origin
- SWC 116 - Block values as a proxy for time
- SWC 117 - Signature Malleability
- SWC 118 - Incorrect Constructor Name
- SWC 119 - Shadowing State Variables
- SWC 120 - Weak Sources of Randomness from Chain Attributes
- SWC 121 - Missing Protection against Signature Replay Attacks
- SWC 123 - Requirement Violation
- SWC 124 - Write to Arbitrary Storage Location
- SWC 125 - Incorrect Inheritance Order
- SWC 126 - Insufficient Gas Griefing
- SWC 127 - Arbitrary Jump with Function Type Variable
- SWC 128 - DoS With Block Gas Limit
- SWC 129 - Typographical Error
- SWC 130 - Right-To-Left-Override control character (U+202E)
- SWC 131 - Presence of unused variables
- SWC 132 - Unexpected Ether balance
- SWC 133 - Hash Collisions With Multiple Variable Length Arguments
- SWC 134 - Message call with hardcoded gas amount
- SWC 135 - Code With No Effects
- SWC 136 - Unencrypted Private Data On-Chain
- Array length manipulation
- Complex Fallback
- Freezing ether
- Function order
- Gassless send
- Mark callable contracts
- Payable fallback
- Reason string
- Unchecked Division
- Quotes
- Uninitialised State
- Visibility Modifier Order

### 3.4.2 Design Patterns

The design patterns in Ethereum are mainly identified as a way to prevent the occurrence of vulnerabilities in the network. In this research, the design patterns were found in three research publications [59][75][61]:

- Access Restriction
- Automatic Deprecation
- Balance Limit
- Commit and Reveal (Hash Secret)
- Contract Register
- Contract Relay

- Checks-Effects-Interaction
- Contract Facade
- Contract Factory
- Contract Composer
- Contract Mediator
- Contract Observer
- Data Segregation
- Emergency Stop (Circuit breaker)
- Mortal
- Mutex
- Multi-signature (Multiple authorization)
- Oracle (Data Provider)
- Ownership
- Pull Payment
- Rate Limit
- Satellite (Contract Decorator)
- State Machine
- Speed Bump

### 3.4.3 Tools

The SWEBOK guide [76] classifies software engineering knowledge areas in a set of categories, including:

- Software construction tools - frameworks that allow the creation of smart contracts in the network through a combination of coding, debugging and deployment.
- Software testing tools - validate and verify the correctness of the contract.
- Software quality tools - evaluate the state of the contract.

These categories were used to classify the tools available in the development of smart contracts and respective sources (Table 3.3):

Table 3.3: Classification of Ethereum tools

| **Tools** | **Category** | **Source** |
|---|---|---|
| Buidler | Construction | https://github.com/nomiclabs/buidler |
| Dapp | Construction | https://github.com/dapphub/dapptools/tree/master/src/dapp |
| Embark | Construction | https://github.com/embarklabs/embark |
| Etherlime | Construction | https://github.com/LimeChain/etherlime |
| OpenZeppelin | Construction | https://openzeppelin.com/ |
| Parasol | Construction | https://github.com/Lamarkaz/parasol |
| Remix IDE | Construction | https://github.com/ethereum/remix |
| Truffle | Construction | https://github.com/trufflesuite/truffle/ |
| adelaide | Quality | https://github.com/sec-bit/adelaide |
| ContractGuard | Quality | https://contract.guardstrike.com/ |
| contractLarva | Quality | https://github.com/gordonpace/contractLarva |
| EasyFlow | Quality | https://github.com/Jianbo-Gao/EasyFlow |
| E-EVM | Quality | https://github.com/pisocrob/E-EVM |
| Erays | Quality | https://github.com/teamnsrg/erays |
| EtherTrust | Quality | https://github.com/SecPriv/EtherTrust |
| Ethlint | Quality | https://github.com/duaraghav8/Ethlint |
| FSolidM | Quality | https://github.com/FSolidM/smart-contracts |
| GigaHorse | Quality | https://zenodo.org/record/2578692#.Xrm0tGhKhPZ |
| HoneyBadger | Quality | https://github.com/christoftorres/HoneyBadger |
| KEVM | Quality | https://github.com/kframework/evm-semantics |

| | | |
|---|---|---|
| KVyper | Quality | https://github.com/kframework/vyper-semantics |
| MadMax | Quality | https://github.com/nevillegrech/MadMax |
| MAIAN | Quality | https://github.com/MAIAN-tool/MAIAN |
| Manticore | Quality | https://github.com/trailofbits/manticore |
| Mythril | Quality | https://github.com/ConsenSys/mythril |
| MythX | Quality | https://mythx.io/ |
| Octopus | Quality | https://github.com/pventuzelo/octopus |
| Osiris | Quality | https://github.com/christoftorres/Osiris |
| Oyente | Quality | https://github.com/melonproject/oyente |
| Porosity | Quality | https://github.com/comaeio/porosity |
| Rattle | Quality | https://github.com/crytic/rattle |
| Remix Analyzer | Quality | https://github.com/ethereum/remix/tree/master/remix-analyzer |
| Securify | Quality | https://github.com/eth-sri/securify2 |
| Slither | Quality | https://github.com/crytic/slither |
| SmartCheck | Quality | https://github.com/smartdec/smartcheck |
| SmartEmbed | Quality | https://github.com/beyondacm/SmartEmbed |
| Solgraph | Quality | https://github.com/raineorshine/solgraph |
| Solhint | Quality | https://github.com/protofire/solhint |
| Solidity Visual Auditor | Quality | https://github.com/ConsenSys/vscode-solidity-auditor |
| Solitor | Quality | https://github.com/LarsStegeman/EthereumRuntimeMonitoring |
| SolMet | Quality | https://github.com/chicxurug/SolMet-Solidity-parser |
| teEther | Quality | https://github.com/nescio007/teether |
| Vandal | Quality | https://github.com/usyd-blockchain/vandal |
| VeriMan | Quality | https://github.com/VeraBE/VeriMan |
| VeriSol | Quality | https://github.com/microsoft/verisol |
| vscode-Vyper | Quality | https://github.com/tintinweb/vscode-vyper |
| Brownie | Testing | https://github.com/eth-brownie/brownie |
| Echidna | Testing | https://github.com/crytic/echidna |
| espresso | Testing | https://github.com/hillstreetlabs/espresso |
| Ethereum Tester | Testing | https://github.com/ethereum/eth-tester |
| ethereum-graph-debugger | Testing | https://github.com/fergarrui/ethereum-graph-debugger |
| hevm | Testing | https://github.com/dapphub/dapptools/tree/master/src/hevm |
| solidity-coverage | Testing | https://github.com/sc-forks/solidity-coverage |
| sol-profiler | Testing | https://github.com/Aniket-Engg/sol-profiler |
| Tenderly | Testing | https://tenderly.co/ |
| Waffle | Testing | https://github.com/EthWorks/Waffle |

### 3.4.4 Effectiveness of Security Analysis Tools

In some studies, security analysis tools are evaluated based on the number of vulnerabilities identified in smart contracts and some of these tools also provided the list of identifiable vulnerabilities.

#### Reported by Tools

Table 3.4 shows the effectiveness of security analysis of each tool in detecting the vulnerabilities identified in Section 3.4.1.

Table 3.4: Effectiveness of Security Analysis Tools

| **Vuln\Tool** | **Remix** | **Slither** | **Oyente** | **Mythril** | **Securify** | **SmartCheck** | **Solhint** | **ContractGuard** |
|---|---|---|---|---|---|---|---|---|
| **Array length manipulation** | x | | | | | x | | |
| **Complex Fallback** | | | | | | | x | x |
| **Constant function state** | | | | x | | | | x |
| **Freezing ether** | | x | | | x | x | | x |
| **Function order** | | | | | | | x | x |
| **Gasless Send** | | x | | | | x | | x |
| **Mark callable contracts** | | | | | | | x | x |
| **Payable fallback** | | | | | | | x | x |
| **Reason string** | | | | | | | x | x |
| **Single quotes** | | | | | | | x | x |
| **SWC-100** | | x | | | x | | x | x |
| **SWC-101** | | | x | x | | | | x |
| **SWC-102** | | x | | | x | | x | |
| **SWC-103** | | x | | | | x | | |
| **SWC-104** | x | | | x | x | x | x | |
| **SWC-105** | | | | x | x | | | |
| **SWC-106** | x | x | | x | x | | | |
| **SWC-107** | x | x | x | x | x | | x | x |
| **SWC-109** | | x | | | x | | | x |
| **SWC-110** | x | | | x | | x | | |
| **SWC-111** | | x | | x | | x | x | x |
| **SWC-112** | | x | | | x | | | x |
| **SWC-113** | x | | x | | x | x | x | x |
| **SWC-114** | | | x | | x | | | |
| **SWC-115** | x | x | | | x | | x | x |
| **SWC-116** | x | x | x | x | x | x | x | x |
| **SWC-118** | | x | | | | x | x | |
| **SWC-119** | | x | | | x | | | x |
| **SWC-120** | x | | | x | | x | x | x |
| **SWC-123** | x | | | | | x | | |
| **SWC-124** | | x | | x | x | | | |
| **SWC-127** | x | x | | | x | | x | x |
| **SWC-128** | x | | | | | x | | x |
| **SWC-130** | | x | | | x | | | x |
| **SWC-131** | | x | | | x | | x | |
| **SWC-132** | | x | | | | x | | x |
| **Unchecked Division** | x | x | | | | | | |
| **Initialised State** | | x | | | x | | | x |
| **Initialised State** | | | | | | | x | x |
| **Total** | 13 | 22 | 4 | 12 | 19 | 14 | 20 | 27 |

According to Table 3.4, the most effective tools detecting vulnerabilities are ContractGuard and Slither, followed by Securify and Solhint. Securify and Slither seem to verify most of the SWC vulnerabilities while ContractGuard verifies most of the others. The least effective tool is Oyente with only 4 identifiable vulnerabilities from SWC registry.

### Reported by studies

The security analysis tools reported by the studies analyses are identified in Table 3.5 with each study correspondent to the identifier in Table 3.2.

Table 3.5: Effectiveness of Security Analysis Tools reported by studies

| Vuln\Tool | MAIAN | Manticore | Mythril | Osiris | Oyente | Porosity | Remix | Securify | SmartCheck | Solgraph | Vandal | ZEUS | Ethir | Gasper | HoneyBadger | Slither | F* | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Freezing ether** | 15,21 | | | | | | | 15,8 | 15,8 | | | | | | | | | 3 |
| **Other** | | 15 | 15 | | | | 15 | 15 | 15 | 15 | 15 | | | | 17 | 18 | | 9 |
| **Payable fallback** | | | | | | | | | 8 | | | | | | | | | 1 |
| **SWC-100** | | | | | | | | | 17 | | | | | | | | | 1 |
| **SWC-101** | | 18 | 17,18 | 18 | 18,21 | | | | 18 | | | 21 | | | | | | 6 |
| **SWC-102** | | | | | | | | | | | | | | | | 17 | | 1 |
| **SWC-104** | 15 | 15,18 | 15,17,18 | | | | 15,17 | 15,18 | 15,17,18 | | | | | | | 17,18 | | 7 |
| **SWC-105** | | | 8,17 | | | | 8 | 8,17 | | | | | | | | 17 | | 4 |
| **SWC-106** | 15,21 | 15 | 17 | | | | 15,17 | | | | 15,21 | 21 | | | | 17 | | 7 |
| **SWC-107** | | 15,18 | 2,8,15,18 | 18 | 2,8,15,17,18,21 | 15 | 2,8,15,17 | 2,8,15,17,18,21 | 2,8,15,18 | | 15,21 | 21 | 21 | | | 17,18 | 2,8 | 13 |
| **SWC-108** | | | | | | | 15 | | 15 | 15 | | | | | | | | 3 |
| **SWC-109** | | | | | | | | | | | | | | | | 17 | | 1 |
| **SWC-110** | | | 17 | | | | | | | | | | | | | | | 1 |
| **SWC-111** | | | 15,17 | | | | 15 | | 15,17 | | | | | | | 17 | | 4 |
| **SWC-112** | | | 8,17 | | | | | 15 | 15 | | | | | | | 17 | | 4 |
| **SWC-113** | | | 2,15 | | 2,8,21 | | 2,8,15 | 2,8,15,21 | 2,8,15 | 15 | 15,21 | 21 | 21 | | | 17 | 2,8 | 11 |
| **SWC-114** | | | 8,15,18 | | 2,8,15,21 | | | 2,8,15,17,18,21 | 2 | | | 21 | 21 | | | | | 6 |
| **SWC-115** | | 15,18 | 2,8,15,17,18 | | 8 | | 2,8,15,17 | 2,8 | 2,8,15,18 | | 15,21 | | | | | 17,18 | | 8 |
| **SWC-116** | | 15,18 | 8,15 | | 2,8,15,21 | | 2,8,15,17 | | 2,8,15,18 | | | 21 | 21 | | | 18 | | 8 |
| **SWC-118** | | | 2 | | 2 | | | 8 | 2 | | | | | | | | | 4 |
| **SWC-119** | | | | | | | | | | | | | | | | 17 | | 1 |
| **SWC-120** | | 15 | 8,15 | | 8 | | 2,15,17 | 8 | 8 | | | | | | | | | 6 |
| **SWC-123** | | | | | | | | 21 | | | | | | | | | | 1 |
| **SWC-124** | | | | | | | | 21 | | | | | | | | | | 1 |
| **SWC-127** | | | 8 | | | | 17 | 17 | 17 | | | | | | | 17 | | 5 |
| **SWC-128** | | | | | 8 | | 2,8,15 | | 2,8,15,17 | | | | | 8,21 | | | 2 | 5 |
| **SWC-132** | 21 | | | | | | | | 17 | | 21 | 21 | | | | 17 | | 5 |
| **Unchecked Division** | | 15 | 8,15 | 15 | | | | | 8, 15 | | | | | | | | | 4 |
| **Total** | 4 | 9 | 17 | 3 | 9 | 1 | 13 | 14 | 20 | 3 | 6 | 7 | 4 | 1 | 1 | 15 | 3 | |

Table 3.5 shows that most of the tools are unsuccessful in identifying the majority of vulnerabilities. The most successful tool is SmartCheck with 20 vulnerabilities identified. Porosity, Gasper and HoneyBadger only identified 1 vulnerability. Mythril, SmartCheck and Slither identified the most SWC vulnerabilities, followed by Remix and Securify.

## 3.5 Summary

This Section aimed to compile all the information available on Ethereum development.

In this chapter, the methodology chosen to perform the literature review was the SLR. The literature review was initially performed defining four research questions, I/E criteria and a search string to screen the data gathered from multiple databases.

As result of this research, three categories were identified, i.e. security vulnerabilities, design patterns and development/analysis tools.

There are several analysis tools implemented to test and assure the correctness of smart contracts. Some of these tools are publicly available, while others are purely academic or experimental. The analysis tools were grouped and compared according to the vulnerabilities claimed by each tool and studies that analysed the tools against a set of smart contracts.

Up until now, only one document [70] focused on the security of Vyper by analysing vulnerabilities in the language. Practitioners, researchers and even security analysis tools use different standards and classifications for the identification of vulnerabilities and security issues. The most official classification is the SWC Registry, provided by the Ethereum community that can be used by any member of the community.

# Chapter 4

# Value Analysis

Value Analysis can be defined as a process of systematic, formal, and organised review that is applied to compare the function of the product, solution, or service required to meet the functional specification and performance criteria required by the customer or consumer [77]. Understanding the key features a solution provides is vital to discuss and understand how the product can improve its value for the targeting market.

This chapter presents the value analysis process proposed by Rich et. al [77], composed by the following phases: Orientation, Functional Analysis, Develop Alternatives, Analysis and Evaluation, and Implementation.

## 4.1 Orientation

The orientation process consists of gathering all the resources required to identify the products or solutions available that can transmit value or opportunities in the market [77]. One of the techniques that can be used in this process is Fuzzy Front End (FFE), also called Front End of Innovation (FEI). The fuzzy front end model, as shown in Figure 4.1, is the first part of the innovation process, that follows two phases, New Concept Development (NCD) and Commercialisation.

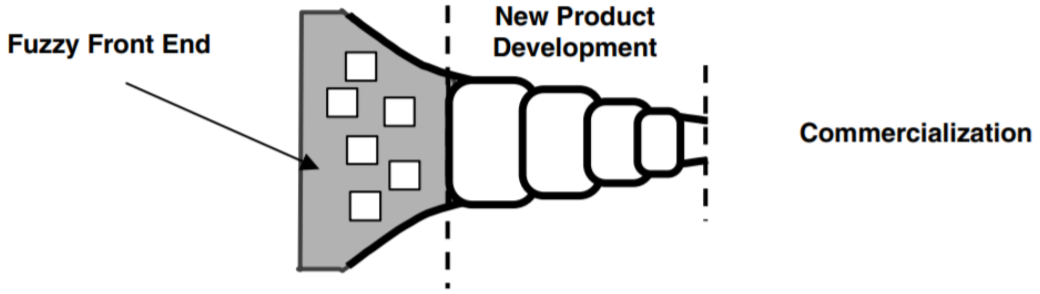


Figure 4.1: The entire innovation process. From [78].

The FFE model can be unstructured and unpredictable to follow. Therefore Koen et a.l [78] improved the understanding of the glsffe by describing it using terms that mean the same thing to everyone using the NCD model represented in Figure 4.2.

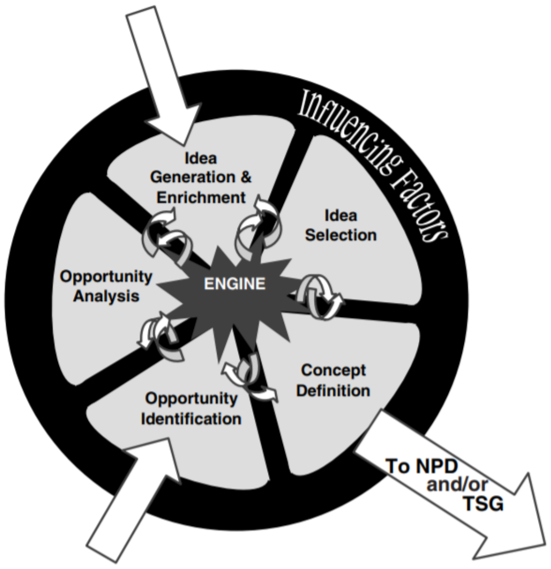


Figure 4.2: The new concept development (NCD). From [78].

The NCD model composes of three parts [78]:

- The engine or bull's-eye portion is composed of the business strategy, culture, and leadership of the organisation that influences the five key elements of the FFE.
- The five key activity elements that represent the inner spoken area of the business.
- The Influencing Factors consist of uncontrollable external factors like, organisational capabilities, the outside world, and the enabling sciences, that might influence the entire innovation process.

### 4.1.1 Business Strategy

For Koen et. al. [78], "the entire innovation process needs to be aligned with business strategy to ensure a pipeline of new products and processes with value to the corporation."

To complement the original context of chapter 1, the strategy of Ethereum is explained. Ethereum's open-source platform strategy wants to enable anyone to "write code that controls digital value, runs exactly as programmed, and is accessible anywhere in the world" [79]. With this strategy statement in mind, it is possible to affirm that Ethereum user's data and ether can be shared without compromising its security. Users should be able to write applications (Dapps) without any concern about the functionality, declaring that Ethereum wants to build a network that has the key attributes: privacy, security, high accessibility, usability, and reliable functionality.

### 4.1.2 Opportunity Identification

Both Bitcoin and Ethereum have been increasing their popularity in the last years, due to their success in asserting as a good currency alternative to the traditional currencies. The following Figure 4.3 shows the trend of online research using the Bitcoin and Ethereum concepts.

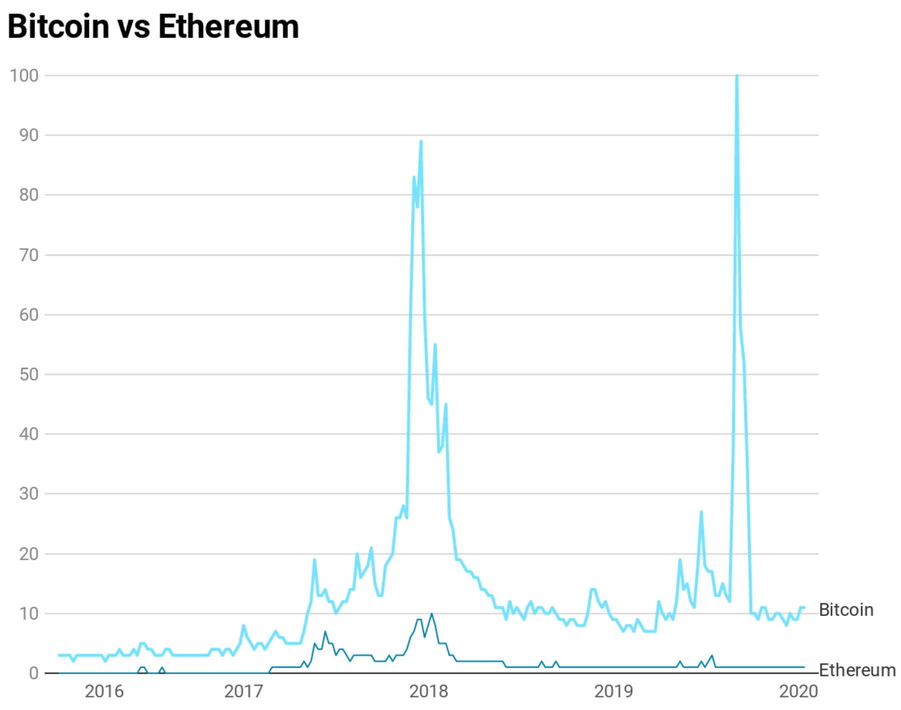


Figure 4.3: Bitcoin vs Ethereum search results [80].

Even though the graph shows a more interest search in Bitcoin, the interest in Ethereum and smart contracts is growing significantly. According to Ethereum web page [79], "thousands of developers all over the world are building applications on Ethereum, and inventing new kinds of applications." At the date of the document, an approximate total of 2,760 DApps are registered in Ethereum [81], and the numbers have been increasing since 2015, as shown in Figure 4.4.

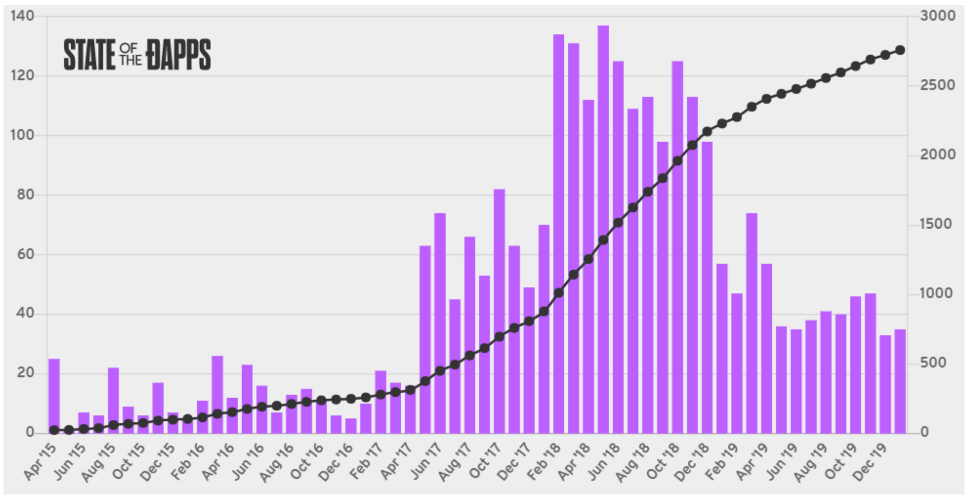


Figure 4.4: Graphic of Ethereum new and total DApps per month. From [81]

The number of Blockchain-related and Ethereum jobs has also increased (Figure 4.5) with a total of almost 4,000 developers in January 2019. Despite the verified downturns, full-time developers increased 13% from June 2018 to June 2019 and are consolidating jobs in the marked. LinkedIn's 2018 U.S Emerging Jobs Report [82] also shows that blockchain developers are the most emerging job roles, having grown 33 times in 2018 over the previous year.

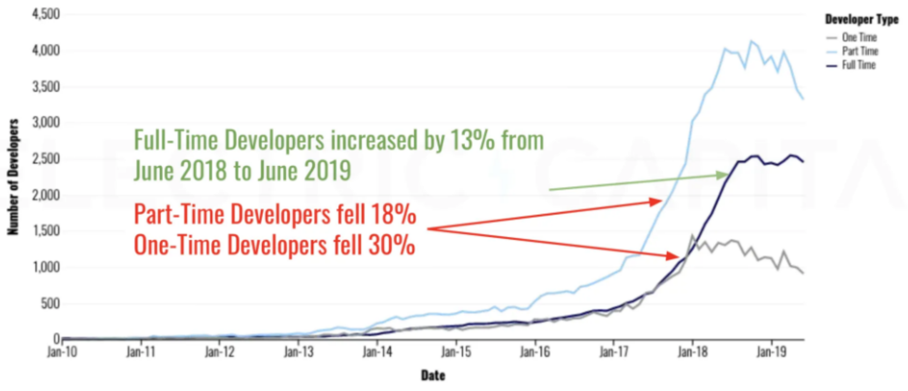


Figure 4.5: Graphic of increased Blockchain developer jobs. From [83]

Other evidence identified, shows that several attacks in the Ethereum ecosystem successfully stole ether coins, including the famous DAO [84], Parity, and MyEtherWallet attacks. The first one, as introduced in Section 2.2.6, stole 3.6 million Ether, forcing Ethereum to a hard fork to return DAO Funds. The second attack, caused by a Parity multisig wallet library contract, when a user supposedly accidentally triggered a killer function [85], stealing 280 million ether. The third one, the attack exploited the unreliable BGP messages, the sensitive DNS and URL redirection and stole approximately 17 million ether [86].

### 4.1.3 Opportunity Analysis

The opportunity identified in the last Section 4.1.2, raises a trust concern in new and aspiring smart contract developers. Being a highly experimental platform that is continuously changing brings uncertainty to the platform and raises many ongoing issues and challenges.

The above incidents, in Section 4.1.2, show that even considered experienced developers have issues in developing smart contracts, and the learning curve to achieve a proficient knowledge and expertise can be very steep and hard to accomplish.

One of the challenges that have to be addressed is the education of Ethereum community, so that a person without a vast knowledge or expertise in smart contract development or even in blockchain in general, can make transactions or make applications (DApps) without being stolen or hacked.

Considering both languages, Solidity and Vyper, the first being a more stable and robust language and latter a yet experimental language but intending to improve security, there is the necessity to protect smart contracts in both languages against the evidenced problems and attacks to Ethereum.

### 4.1.4 Idea Generation and Selection

From gathering the data of Ethereum resources, demonstrated in Section 3.3, and opportunity identified, the following deductions can be assessed:

- Security tools are not sufficient for a good smart contract development. There is a need to use other tools to develop, deploy, and test smart contracts.
- Guidelines and resources to learn about the skills and best practices in any language are necessary for a clean and code smell-free code.
- There isn't enough evidence supporting the idea that Vyper has major security vulnerabilities. The assessment of the Vyper's security could lead to a change in smart contract development. New tools for Vyper like linters, formatters, and even frameworks for formal verification tools could be considered.

It is necessary to analyse the smart contracts tools with prototypes that allow demonstrating the efficiency, accuracy, and reliability of the tools explored, against both Solidity and Vyper and make it possible to select the most suitable for the development of smart contracts.

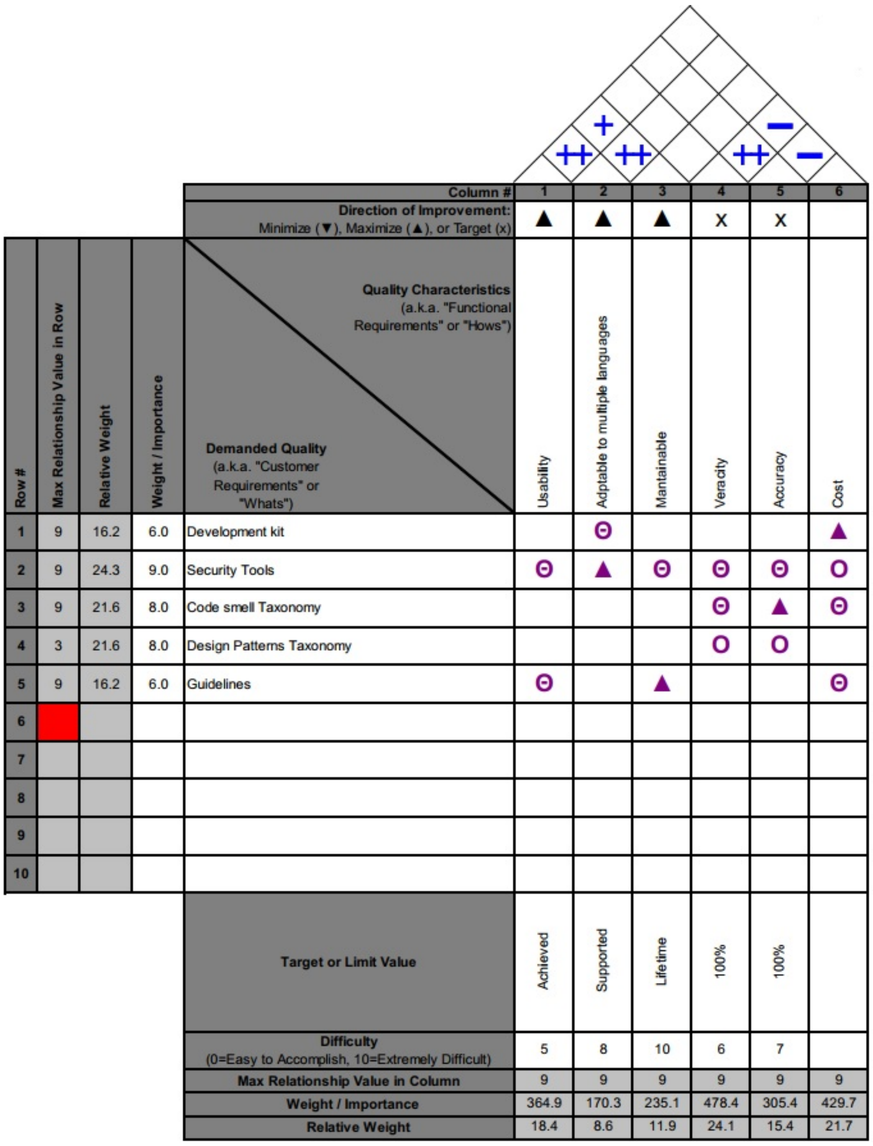
### 4.1.5 Influencing Factors

There are a few key factors of the blockchain ecosystem and Ethereum that could influence the approach of this document:

- Centralisation of Ethereum - Currently, the two most commonly used clients for running nodes (Ethereum programs that maintain the blockchain) are Geth and Parity. However, if any of these clients is abandoned, and no new clients emerge, "Ethereum will become more centralised and vulnerable to attacks that target a single type of client" [87].
- Regulations - Although Ethereum interest is increasing, several countries are prohibiting or banning Ethereum from being sold or used. Since no official institutions are working as an intermediary or backing the contract, if any account is stolen, there are no legal rights that can help or compensate. It is up to the community to judge and decide the course of action.
- Serenity - The last stage of the improvement of the Ethereum network, Serenity, is in an ongoing process with some crucial changes being made. These changes could influence the entire paradigm of the network and therefore change the development approaches.

## 4.2 Functional Analysis

The stage of Functional Analysis is the process of identifying the most critical functions of a product or service [77]. In order to serve these requirements, a Quality Function Deployment (QFD) has been proposed, Figure 4.6. The QFD is a product developing technique performed to translate customer requirements to product specification [88].



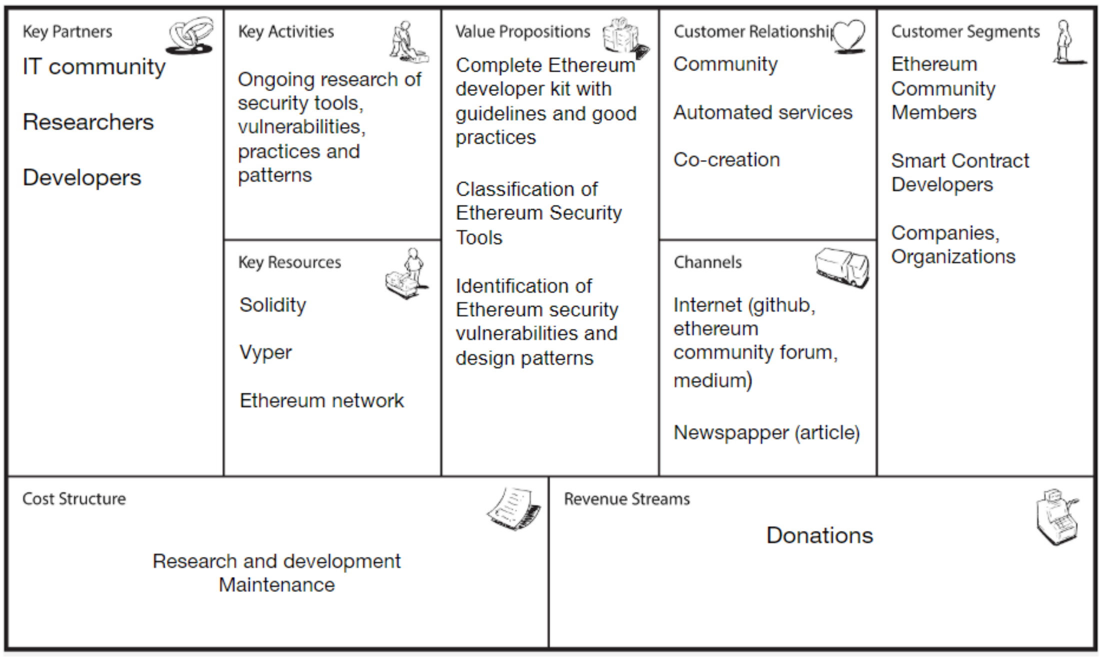
| | | | | Column # | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Direction of Improvement: Minimize (▼), Maximize (▲), or Target (x) | ▲ | ▲ | ▲ | x | x | |
| Row # | Max Relationship Value in Row | Relative Weight | Weight / Importance | Quality Characteristics (a.k.a. "Functional Requirements" or "Hows") / Demanded Quality (a.k.a. "Customer Requirements" or "Whats") | Usability | Adptable to multiple languages | Mantainable | Veracity | Accuracy | Cost |
| 1 | 9 | 16.2 | 6.0 | Development kit | | Θ | | | | ▲ |
| 2 | 9 | 24.3 | 9.0 | Security Tools | Θ | ▲ | Θ | Θ | Θ | O |
| 3 | 9 | 21.6 | 8.0 | Code smell Taxonomy | | | | Θ | ▲ | Θ |
| 4 | 3 | 21.6 | 8.0 | Design Patterns Taxonomy | | | | O | O | |
| 5 | 9 | 16.2 | 6.0 | Guidelines | Θ | | ▲ | | | Θ |
| 6 | 🟥 | | | | | | | | | |
| 7 | | | | | | | | | | |
| 8 | | | | | | | | | | |
| 9 | | | | | | | | | | |
| 10 | | | | | | | | | | |
| | | | | Target or Limit Value | Achieved | Supported | Lifetime | 100% | 100% | |
| | | | | Difficulty (0=Easy to Accomplish, 10=Extremely Difficult) | 5 | 8 | 10 | 6 | 7 | |
| | | | | Max Relationship Value in Column | 9 | 9 | 9 | 9 | 9 | 9 |
| | | | | Weight / Importance | 364.9 | 170.3 | 235.1 | 478.4 | 305.4 | 429.7 |
| | | | | Relative Weight | 18.4 | 8.6 | 11.9 | 24.1 | 15.4 | 21.7 |

Figure 4.6: Quality Function Deployment

Analysing Figure 1 is possible to identify the most important quality characteristics of the customer's requirements being the veracity, the cost to production, followed by usability and accuracy. Stating that the solution should validate if the requirements gathered can be used by any developer or aspiring Ethereum developer, providing simple access to guidelines and best practices and, develop contracts with a correct indication of the finding vulnerabilities during the process.

## 4.3 Developing Alternatives, Analysis and Implementation

Analysing and evaluating the cost and worth of the product allows assessing its potential value[77]. To describe the value for the customer's project involved, the Business Model Canvas was created, Figure 4.7.



Key Partners
IT community
Researchers
Developers

Key Activities
Ongoing research of security tools, vulnerabilities, practices and patterns

Key Resources
Solidity
Vyper
Ethereum network

Value Propositions
Complete Ethereum developer kit with guidelines and good practices
Classification of Ethereum Security Tools
Identification of Ethereum security vulnerabilities and design patterns

Customer Relationshi
Community
Automated services
Co-creation

Channels
Internet (github, ethereum community forum, medium)
Newspapper (article)

Customer Segments
Ethereum Community Members
Smart Contract Developers
Companies, Organizations

Cost Structure
Research and development
Maintenance

Revenue Streams
Donations

Figure 4.7: Business Model Canvas

The fulfilment and evaluating of the value proposition referred in 4.7, can be accomplished with an approach to research the demanded qualities in the Ethereum development process based on the multiple alternatives and tools to evaluate objectively according to the functional requirements.

A Design Science Research (DSR), proposed by Hevner et al. [89], is used to ensure the alignment of this approach. A DSR is a cycle to analyse the use and performance of designed artefacts intended to solve the identified problems. The DSR used is defined in three main phases "problem identification", "solution design" and "evaluation" [90]:

- Problem identification: Ensure the relevance of the problem (Chapter 5) using the gathered tools to identify the value of the problem (Chapter 4) and unsolved state of the art (Chapter 3).
- Solution Design: Define a viable solution (Chapter 7), taking into account existing artefacts viable to the solution (Chapter 6).
- Evaluation: The evaluation demonstrate the utility, quality and efficacy of the design solution, using a practical application (Chapter 8) and summarising the results of the solution (Chapter 9).

# Chapter 5

# Problem statement

## 5.1 Objectives

The security of smart contracts is a major issue in Ethereum languages. The developers in the network should be able to write smart contracts knowing these issues and validating their code against them. Prior studies have already focused on these issues [56][60][74] and analysed the frameworks and tools [17][66][18][69] used to test the contracts vulnerabilities. However, these studies did not focus on the integration of these tools and frameworks in the smart contracts development process [91]. This implies that the development process of smart contracts is not adequate and there is a "clear gap between the research regarding developers' perception of smart contract issues and technologies" [92]. The knowledge of new developers is limited to what they can find and understand through research, and to resources available on Ethereum languages [93].

The main goal of this dissertation is to provide new uses for the Ethereum development practices by analysing the tools available in Ethereum, considering the two main languages, Solidity and Vyper. The final solution should demonstrate the safety of executing smart contracts providing a better perception of issues and technologies to developers.

The objectives are summarised as:

- Explore solutions to identify vulnerabilities and guidelines that provide a readable resolution of these vulnerabilities and potential improvements.
- Address the smart contract vulnerabilities and technologies, proposing a solution to ease the development of smart contracts using good practices and guidelines.

## 5.2 Main Points of Focus

The resources to substantiate the knowledge of Ethereum development practices are already introduced in Chapter 3. However, the improvement of these practices requires many following points of focus:

- Development Process

  The improvement of a development process can have a resemblance to a streamline of steps to improve a business process. The steps are defined as **Mapping** - Identify the key areas to improve; **Analyse the process** - Analyse which of the areas developers get more issues and what areas can create obstacles to slow or delay the process;

**Redesign** - Gather the information of the mapping stage and design a solution to eliminate the issues found in the areas; **Implement** - Find the right resources available to implement the redesign and perform the changes in the respective areas; **Reflect, Review and potentially, Optimise** - Monitor the execution of the new process to identify what changes made don't work and what areas can be improved.

- Responsibility

  Changing the development practices of any software should not change the scope of responsibilities of the end-user. The user should always be able to perform the usual coding software without having to perform extra required steps, like having to create a specific test case or having to follow specific coding syntax. This steps should always be optional and configurable to the user's needs.

- Automation

  The ability to move manual processes to automated processes can reduce the risk of developer errors and keep track of the development process. An automated solution can focus on the most important tasks and improve the quality of the development.

## 5.3 Approach

The approach of this dissertation focus on comparing techniques used to improve the development of smart contracts in Ethereum languages and define a feasible solution that serves as a reference to the Ethereum community awareness of code smells, security vulnerabilities, tools available, and design patterns found. The resulting solution would also provide the user with continuous integrated development and vulnerability detection framework on Ethereum smart contracts. The solution is developed in two main Chapters, each of them answering one of the questions in Section 5.1:

- Ethereum Development Research (Chapter 6)
- Solution Design and Implementation (Chapter 7)

# Chapter 6

# Ethereum Development Research

This chapter presents the research plan design and execution to validate and identify the possible practices using the tools available in the evolving state of smart contracts development.

## 6.1 Methodology

This section provides a detailed explanation of the analysis and evaluation method performed based on the categories identified in Section 3.4. The methodology used is described in Figure 6.1.

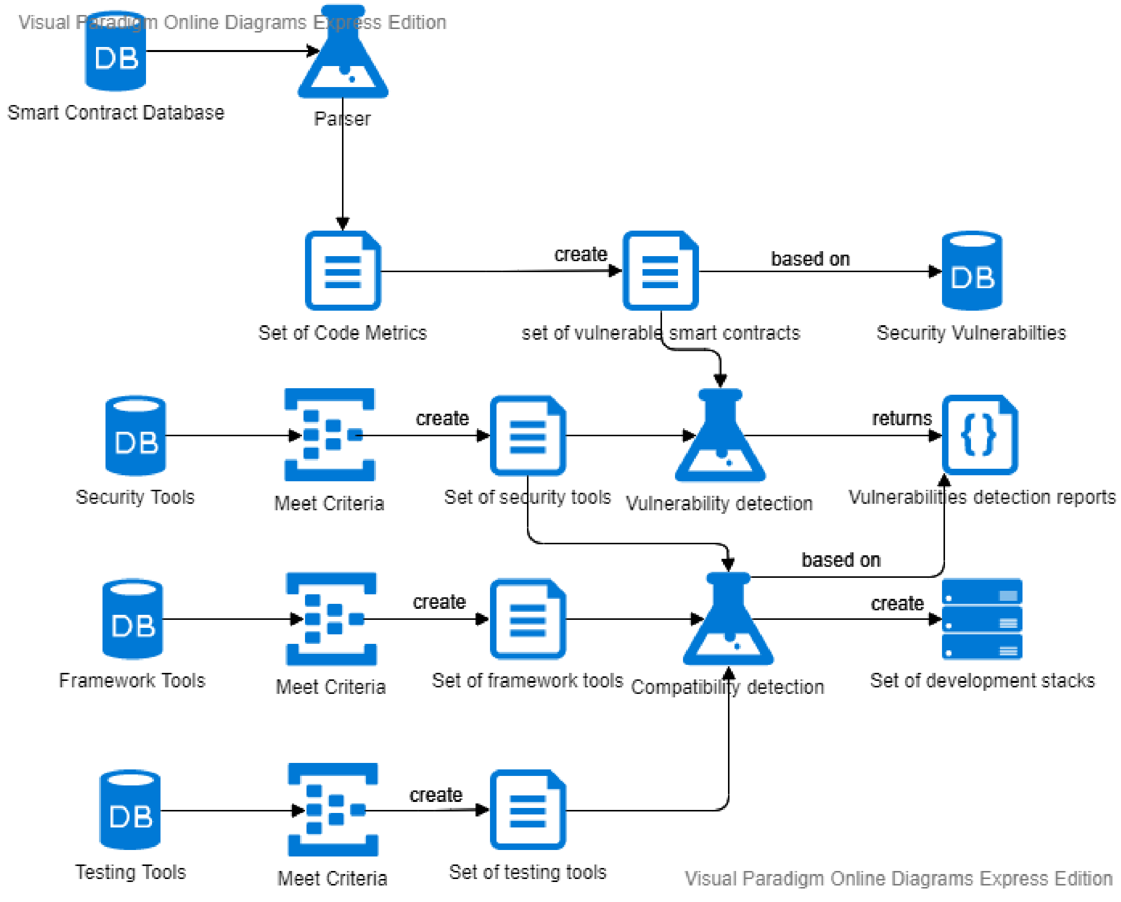


Figure 6.1: Ethereum Development research methodology

As shown in Figure 6.1, the research flow starts with the parsing of a collection of smart contracts. The parser analyses the code of these contracts and generates a list of code metrics obtained from the contracts. Then, based on the security vulnerabilities, a new set of vulnerable contracts is created, with each contract containing a single vulnerability. The process continues with a selection of security tools that meet the required criteria and detect the results of each vulnerable contracts in the selected tools. The frameworks and testing tools are also selected based on required criteria. Finally, a compatibility detection between the tools and frameworks is performed to create a list of development stacks, that contains a list of languages, frameworks and tools used for the development of smart contracts. The results of this analysis are compared with responses obtained in Section 3.4.4 and conclusions drawn.

## 6.2 Smart Contract Data Collection

The smart contracts were retrieved from Etherscan [94], an explorer that allows obtaining a list of verified smart contracts published in the network. A verified contract is a contract compiled and analysed to make sure that the respective code matches the deployed contract in the blockchain. The extraction was conducted with a list of smart contracts verified and available until February 2020.

The extraction process was performed with a Python script, available in github [95], to scan an excel file provided by Etherscan and obtain the source code of each contract. The excel file is a compilation of the addresses of smart contracts available in the Ethereum Mainnet and Testnets.

The script reads the address from the excel and downloads the source and ABI from Etherscan Application Programming Interface (API) to a folder containing the source code files and ABI code. The version and language of each contract is accounted for later verification. As a result of the extraction, 7911 contract codes were downloaded, from a total of 7914 addresses, where 7796 contracts were developed with Solidity and 115 with Vyper. The number of contracts downloaded from the Mainnet and TestNets is identified in the following Table 6.1.

Table 6.1: Number of contracts downloaded from Mainnet and TestNets

| URL | Number |
|---|---|
| api-goerli.etherscan.io | 112 |
| api-kovan.etherscan.io | 559 |
| api-rinkeby.etherscan.io | 1155 |
| api-ropsten.etherscan.io | 2393 |
| api.etherscan.io | 3692 |

From the total number of contracts downloaded from Etherscan, 3692 were retrieved from the Mainnet. Almost two-thirds of these numbers are deployed and extracted from the Ropsten TestNet.

The statistic of the number of Solidity and Vyper contracts per compiler version is identified in Figure 6.2 and Figure 6.3, respectively.

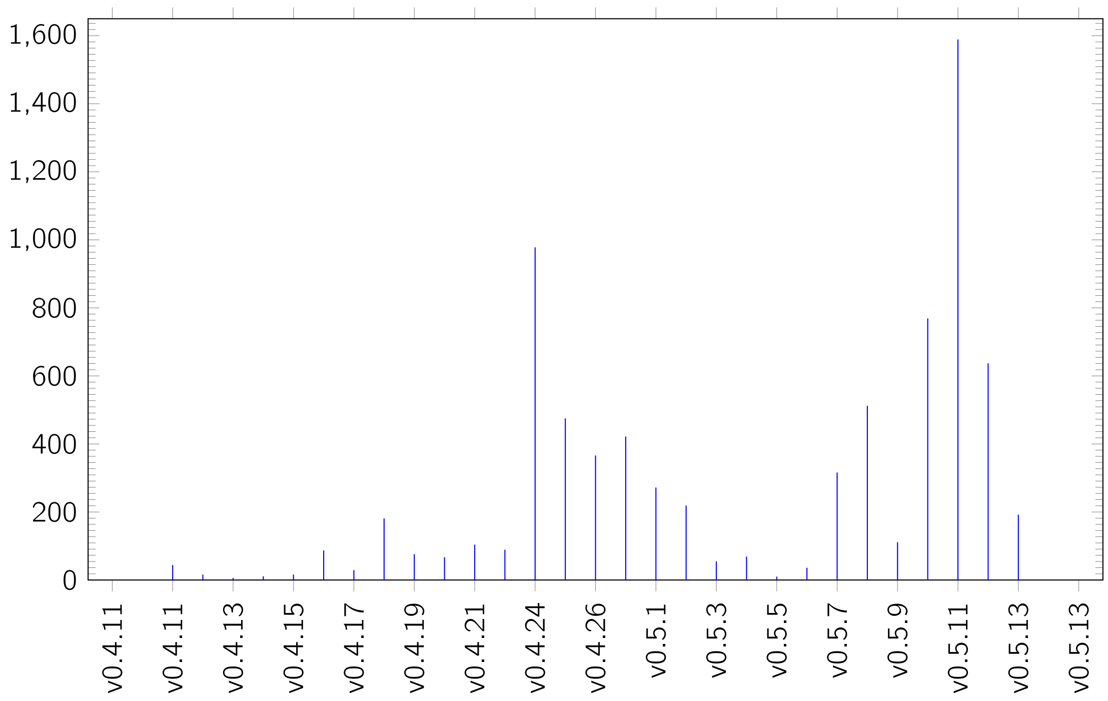


Figure 6.2: Histogram of the number of verified Solidity contracts per compiler version.

From the histogram in Figure 6.2, the most used Solidity compiler versions are v0.4.24 and v0.5.11. No results were observed for the version 0.4.22 and versions before v0.4.11.

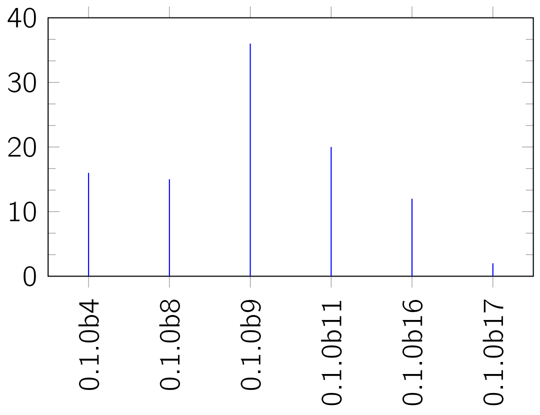


Figure 6.3: Histogram of the number of verified Vyper contracts per compiler version.

Regarding Vyper, the histogram in Figure 6.3, shows a single normal distribution with the most used version 0.1.0b9 containing almost 40 contracts.

Following the process flow, after gathering the code for each address, the parsing of the obtained smart contracts is performed. The parsing is divided in two script written in Python, one for Solidity [96] and another one for Vyper [97]. The parsers transforms the contracts into an AST tree, using the Solidity parser [96] and Vyper built-in parser. The parsed trees were serialised to a contract-level and function-level, capturing the structural information of the contract names, functions and duplicated contracts. The Solidity parser version 0.0.7 used could not process 482 contracts of the total 7796 and Vyper version 0.1.0b17 could not parse 80 out of a total of 115.

The analyses identified a total of 5370 solidity contract addresses using contract names with 50440 of those used more than once, remaining 3130 unique contract names in the collection. In Solidity, 4339 duplicated contracts were identified in a total of 7314 contracts available.

Table 6.2 references the top 30 contract names written in Solidity ordered by total of occurrences.

Table 6.2: Top 30 most used contract names written in Solidity contracts

| Contract Name | Total |
|---|---|
| IERC165 | 359 |
| ERC165 | 370 |
| BasicToken | 388 |
| ERC20Mintable | 415 |
| Token | 459 |
| MinterRole | 490 |
| Context | 521 |
| ERC20Basic | 531 |
| StandardToken | 566 |
| Address | 584 |
| ERC20Detailed | 645 |
| Pausable | 668 |
| ApproveAndCallFallBack | 757 |
| Roles | 909 |
| Owned | 942 |
| ERC20Interface | 1005 |
| IERC20 | 1708 |
| Ownable | 2048 |
| ERC20 | 2260 |
| SafeMath | 4483 |

From the analysed Table 6.2, *SafeMath* is the most referenced contract and presents a library that performs arithmetic operations with overflow verification. All *ERC20*-like contracts represent a variation of the *ERC20* token implementation. The third most used contract name, the *Ownable* contract, is a pattern that provides a basic authorisation control for the owners of the contract.

As Vyper contracts are identified in a single file and don't have specific names, the study focused on the functions naming. The top 25 contract functions written in Vyper are identified in Table 6.3.

Table 6.3: Top 25 most used contract functions written in Vyper contracts

| | |
|---|---|
| onERC20Received(address,uint256) | 6 |
| getValuation(uint256) | 6 |
| tokenByIndex(uint256) | 8 |
| _init_(string,string,uint256,uint256) | 8 |
| mint(address,uint256) | 8 |
| tokenOfOwnerByIndex(address,uint256) | 8 |
| supportsInterface(bytes32) | 9 |
| safeTransferFrom(address,address,uint256) | 9 |
| safeTransferFrom(address,address,uint256,bytes) | 9 |
| isApprovedForAll(address,address) | 9 |
| ownerOf(uint256) | 9 |
| getApproved(uint256) | 9 |
| setApprovalForAll(address,bool) | 9 |
| burnFrom(address,uint256) | 11 |
| _init_() | 11 |
| burn(uint256) | 12 |
| transfer(address,uint256) | 13 |
| allowance(address,address) | 13 |
| decimals() | 14 |
| totalSupply() | 21 |
| symbol() | 21 |
| approve(address,uint256) | 22 |
| transferFrom(address,address,uint256) | 22 |
| balanceOf(address) | 22 |
| name() | 26 |

Regarding the result of Vyper functions, Vyper compiler assumes that all public variables are described as a function and from the 35 contracts identified, three of them are public variables: name, symbol and decimals. The high number of similar function names suggest that some contracts are identical, and perform the same operations. Analysing the duplicability of functions 23 contracts were identified with the same functions.

## 6.3 Tools and Frameworks

### 6.3.1 Analysis

In this section, the available analysis tools are examined and compared regarding the following aspects:

- Level of execution (source code and byte code)
- Use of a command line interface to execute the tool
- Display of information in a User Interface
- Integration with an Integrated Development Environment (IDE) throughout extensions and plugins
- Maintained by the community or company

- Available to use without costs
- Detect known vulnerabilities

Table 6.4: Comparison of smart contract analysis tools

| **Tool** | **Type** | **Level** | **CLI** | **UI** | **IDE Extension** | **Maintained** | **Available** | **Detect Vulnerabilities** |
|---|---|---|---|---|---|---|---|---|
| ContractGuard | Analysis | source code | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ |
| contractLarva | Analysis | source code | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ |
| EtherTrust | Analysis | bytecode | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| EasyFlow | Analysis | bytecode | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| Manticore | Analysis | both | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| E-EVM | Analysis | bytecode | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| Mythril | Analysis | both | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Erays | Analysis | bytecode | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| MythX | Analysis | both | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ |
| GigaHorse | Analysis | bytecode | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| Octopus | Analysis | bytecode | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ |
| HoneyBadger | Analysis | both | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| KVyper | Analysis | source code | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| MadMax | Analysis | bytecode | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| MAIAN | Analysis | both | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ |
| Oyente | Analysis | bytecode | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| Rattle | Analysis | bytecode | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ |
| Securify | Analysis | source code | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| Slither | Analysis | source code | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| SmartCheck | Analysis | source code | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| Osiris | Analysis | both | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| VeriSol | Analysis | source code | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ |
| Solgraph | Analysis | source code | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| teEther | Analysis | bytecode | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| Vandal | Analysis | bytecode | ✓ | ✗ | ✗ | ✗ | ✓ | ✓ |
| VeriMan | Analysis | source code | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| adelaide | Analysis | source code | ✓ | ✗ | ✓ | ✗ | ✗ | ✓ |
| Porosity | Analysis | source code | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ |
| SmartEmbed | Analysis | source code | ✗ | ✓ | ✗ | ✗ | ✓ | ✓ |
| Ethlint | Linter | source code | ✓ | ✗ | ✓ | ✗ | ✓ | ✓ |
| Solhint | Linter | source code | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| Solidity Visual Auditor | Linter | source code | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| vscode-Vyper | Linter | source code | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| Remix Analyzer | Analysis | source code | ✗ | ✓ | ✗ | ✓ | ✓ | ✓ |
| KEVM | Analysis | bytecode | ✗ | ✗ | ✗ | ✓ | ✓ | ✗ |

Table 6.4 represents the comparison of the analysis tools collected. Then, the tools are chosen based on the following criteria: maintenance, to verify if the tool is updated accordingly to Ethereum changes; availability, to choose open-source or free tools; and vulnerability detection, to select tools that can verify issues in smart contracts. As a result, 8 of the analysed tools meet the criteria: ContractGuard, Manticore, Mythril, Securify, Slither, SmartCheck, Solhint, Remix Analyzer. Despite the ability of MythX to detect vulnerabilities and one of the most used analysis tools for smart contracts, it requires a monthly fee to analyse multiple contracts, thus not meeting the criteria defined.

To analyse the selected tools against the set of vulnerable smart contracts defined in Section 6.2, a modified version of an automated tool called SmartBugs[98] was selected. SmartBugs automates the execution of predefined Command Line Interface (CLI) analysis tools, based on docker images, against a set of smart contracts. This tool was modified [99] to allow new versions of the analysis tools and different Solidity versions, using the solc-select script [100].

The communication flow of the modified tool is defined in Figure 6.4.

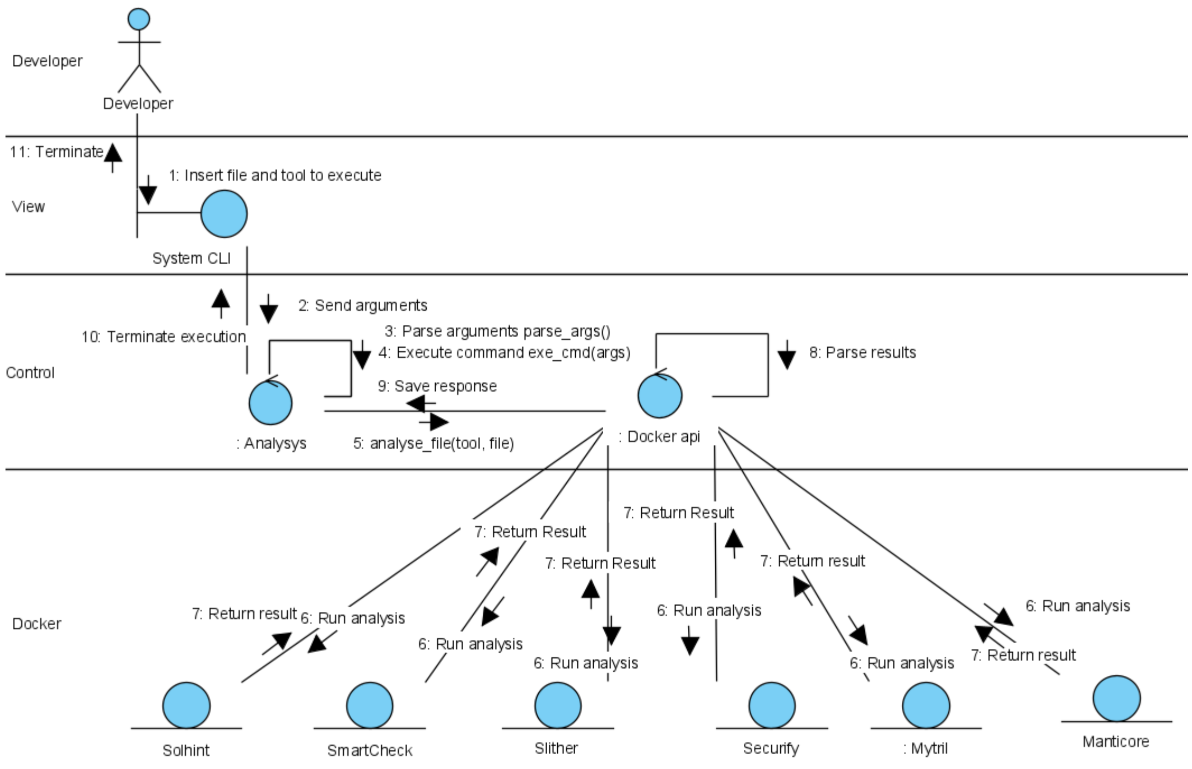


Figure 6.4: Security analysis tool communication flow

After each analysis, the results were manually checked and classified accordingly to the respective vulnerability. The result of the execution of this tool in complement with a manual analysis of ContractGuard and Remix Analyzer is displayed in Table 6.5. Every execution of the analysis tool against a vulnerable contract is classified in one of the following aspects:

- V - The tool found the vulnerability identified in the contract
- O - The tool found other vulnerabilities
- E - The tool could not parse the contract
- N - The tool didn't find vulnerabilities

Table 6.5: Comparison of the analysis tools against listed vulnerable smart contracts

| **Vulnerabilities** | **ContractGuard** | **Manticore** | **Mythril** | **Securify** | **Slither** | **SmartCheck** | **Solhint** | **Remix Analyzer** | **Compiler Fixed Version** |
|---|---|---|---|---|---|---|---|---|---|
| Array length manipulation | O | O | O | O | O | V | O | O | 0.6.0 |
| Complex Fallback | V | O | N | V | V | O | V | V | - |
| Constant function state | V | N | N | V | V | O | N | V | 0.5.0 |
| Freezing ether | V | O | N | V | O | N | O | N | - |
| Function order | V | N | N | O | O | O | V | N | - |
| Gasless send | V | E | V | V | V | V | V | V | - |
| Mark callable contracts | V | N | N | O | O | O | V | O | - |
| Payable fallback | V | N | N | O | N | O | V | N | - |
| Reason string | V | O | N | O | O | O | V | O | - |
| Single quotes | V | O | N | O | O | O | V | O | - |
| SWC-100 Function Default Visibility | V | E | N | E | V | O | V | O | 0.5.0 |
| SWC-101 Integer Overflow and Underflow | V | V | V | O | O | O | N | N | - |
| SWC-102 Outdated Compiler Version | N | E | E | O | E | N | N | N | - |
| SWC-103 Floating Pragma | N | N | N | O | V | V | O | N | - |
| SWC-104 Unchecked Call Return Value | O | E | V | E | V | V | O | V | 0.5.0 |
| SWC-105 Unprotected Ether Withdrawal | O | V | V | V | O | O | N | O | - |
| SWC-106 Unprotected SELFDESTRUCT Instruction | N | V | V | V | V | O | O | V | - |
| SWC-107 Reentrancy | O | E | V | V | O | O | O | V | - |
| SWC-108 State Variable Default Visibility | V | E | N | E | O | V | V | O | - |
| SWC-109 Uninitialized Storage Pointer | V | E | O | E | E | O | V | O | 0.5.0 |
| SWC-110 Assert Violation | N | O | V | N | N | N | N | V | - |
| SWC-111 Use of Deprecated Solidity Functions | V | E | O | E | V | V | V | V | - |
| SWC-112 Delegatecall to Untrusted Callee | V | E | V | V | V | O | O | V | - |
| SWC-113 DoS with Failed Call | V | O | V | V | V | O | V | V | - |
| SWC-114 Transaction Order Dependence | O | O | O | V | O | O | O | O | - |
| SWC-115 Authorization through tx.origin | V | V | V | V | V | V | V | V | - |
| SWC-116 Block values as a proxy for time | O | V | V | O | O | O | O | O | - |
| SWC-117 Signature Malleability | O | E | V | E | O | O | N | O | - |
| SWC-118 Incorrect Constructor Name | O | E | N | E | O | O | O | O | 0.5.0 |
| SWC-119 Shadowing State Variables | O | N | N | E | V | O | O | N | 0.5.0 |
| SWC-120 Weak Sources of Randomness from Chain Attributes | V | N | O | O | O | O | V | V | - |
| SWC-123 Requirement Violation | O | O | N | O | V | O | O | O | - |
| SWC-124 Write to Arbitrary Storage Location | O | E | V | V | O | O | O | O | - |
| SWC-125 Incorrect Inheritance Order | O | E | O | E | E | O | O | O | - |
| SWC-126 Insufficient Gas Griefing | O | N | O | O | O | O | O | O | - |
| SWC-127 Arbitrary Jump with Function Type Variable | V | N | V | N | V | V | V | V | - |
| SWC-128 DoS With Block Gas Limit | O | N | N | O | O | O | O | V | - |
| SWC-129 Typographical Error | O | N | N | E | E | O | O | N | 0.5.0 |
| SWC-130 Right-To-Left-Override control character (U+202E) | O | O | N | V | V | O | O | O | - |
| SWC-131 Presence of unused variables | V | N | N | O | O | O | V | N | - |
| SWC-132 Unexpected Ether balance | V | V | N | V | V | V | N | O | - |
| SWC-133 Hash Collisions With Multiple Variable Length Arguments | N | N | N | O | N | O | O | O | |
| SWC-134 Message call with hardcoded gas amount | E | N | O | E | N | O | O | O | - |
| SWC-135 Code With No Effects | O | N | N | O | O | O | O | O | - |
| SWC-136 Unencrypted Private Data On-Chain | O | N | O | V | V | V | O | O | - |
| Unchecked Division | N | N | N | N | N | O | N | V | - |
| Unitialized State | V | N | N | V | V | N | N | N | - |
| Visibility Modifier Order | V | N | N | O | O | O | V | O | - |
| Total - V - Found specific vulnerability and other vulnerabilities | 23 | 6 | 14 | 16 | 18 | 10 | 17 | 15 | |
| Total - O - Found other vulnerabilities | 18 | 10 | 9 | 18 | 21 | 34 | 22 | 23 | |
| Total - N - No errors found | 6 | 19 | 24 | 3 | 5 | 4 | 9 | 10 | |
| Total - E - Could not parse | 1 | 13 | 1 | 11 | 4 | 0 | 0 | 0 | |

As can be seen in the Table 6.5:

- Only Remix Analyzer verifies unchecked divisions and SWC-128.
- Seven of the vulnerabilities identified are fixed by the Solidity compiler after v0.5.0 and after v0.6.0 to array length manipulation vulnerability.
- Eight specific vulnerabilities were not verified by the analysis tools: SWC-102, SWC-118, SWC-125, SWC-129, SWC-133, SWC-134 and SWC-135.
- The tools Solhint, SmartCheck and Remix Analyzer didn't provide any compilation error to verify the contract.
- Exception disorder is verified by almost all tools except Manticore and Solhint.

## 6.3.2 Development

Development frameworks are compared in Table 6.6. For better development of smart contracts, it is required that the framework can execute in Solidity and Vyper and is constantly maintained to support new Ethereum versions and languages. Remix, Embark, Truffle, Etherlime and Buidler meet these requirements.

Table 6.6: Comparison of frameworks for developing smart contracts

| **Tool** | **Solidity** | **Vyper** | **Add extensions** | **Mantained** |
|---|---|---|---|---|
| Remix | ✓ | ✓ | ✓ | ✓ |
| Dapp | ✓ | ✗ | ✗ | ✓ |
| Embark | ✓ | ✓ | ✓ | ✓ |
| Etherlime | ✓ | ✓ | ✗ | ✓ |
| OpenZeppelin | ✓ | ✗ | ✗ | ✓ |
| Parasol | ✓ | ✗ | ✗ | ✗ |
| Truffle | ✓ | ✓ | ✓ | ✓ |
| Buidler | ✓ | ✓ | ✓ | ✓ |

## 6.3.3 Testing

Regarding the requirements for testing tools, it is required that the tool:

- Execute in Solidity and Vyper languages.
- Execute using the CLI.
- Is maintained.
- Is free.

The comparison of the testing tools is described in Table 6.7:

Table 6.7: Comparison of testing tools for smart contracts

| **Tool** | **Level** | **Solidity** | **Vyper** | **CLI** | **UI** | **Maintained** | **Paid** |
|---|---|---|---|---|---|---|---|
| ethereum-graph-debugger | source code | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ |
| espresso | source code | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Echidna | source code | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ |
| Ethereum Tester | bytecode | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ |
| hevm | both | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ |
| solidity-coverage | source code | ✓ | ✗ | ✓ | ✗ | ✓ | ✗ |
| sol-profiler | source code | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Tenderly | bytecode | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| Waffle | source code | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ |
| Brownie | source code | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ |

Base on the comparison of the testing tools, the following meet the requirements: Echidna, hevm, solidity coverage, Waffle and Brownie.

## 6.4 Compatible Development Stacks

A development stack is a set of tools and frameworks that are used in the development of a software product. In smart contracts development, the development stack consists of a set of components that create a functional smart contract or Dapp.

Respecting all the previous analysed tools, CLI analysis tools can be aggregated to analyse a smart contract to verify a greater number of vulnerabilities. In this case, Mythril, Securify, Slither and Solhint were combined as MSSS tools in order to verify a higher number of vulnerabilities. Table 6.8 compares the screened set of tools with each other. Frameworks are not compared against each other since their usage is completely independent of each other.

Table 6.8: Comparison of tools with other tools

| **Tools** | **ContractGuard** | **MSSS Tools** | **Remix Analyzer** | **Echidna** | **HEVM** | **Solidity Coverage** | **Waffle** | **Remix** | **Embark** | **Etherlime** | **Truffle** | **Buidler** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ContractGuard | - | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| MSSS Tools | - | - | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Remix Analyzer | - | - | - | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |
| Echidna | - | - | - | - | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| HEVM | - |  | - | - | - | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| Solidity Coverage | - | - | - | - | - | - | ✓ | ✗ | ✗ | ✓ | ✓ | ✓ |
| Waffle | - | - | - | - | - | - | - | ✓ | ✓ | ✓ | ✓ | ✓ |

From the Table 6.8 the following aspects can be found:

- ContractGuard is not compatible with any tool or framework.
- MSSS tools can be used in any Framework and are compatible with other testing and analysis tools except Remix analyzer.
- Remix Analyzer can work with other testing and analysis tools but only support their framework Remix.
- Etherlime does not support testing tools besides Solidity Coverage.
- The frameworks that supports more tools are Truffle and Buidler.

For each framework, the following development stacks are identified:

- Remix + Remix Analyzer + MSSS tools + Echidna + Waffle
- Embark + MSSS tools + Echidna + Waffle
- Etherlime + MSSS tools + Solidity Coverage + werenWaffle
- Truffle + MSSS tools + Solidity Coverage + Echidna + Waffle
- Buidler + MSSS tools + Solidity Coverage + Echidna + Waffle

## 6.5 Comparison to State of the Art

As established in Chapter 3, many security analysis tools, have some but not complete effectiveness in detecting the full list of vulnerabilities identified. From the security tools analysed in this study, only ContractGuard and Solhint are not analysed in previous studies.

Regarding the reported vulnerabilities from each tool, only Oyente was not analysed in this study. A comparison of the reported vulnerabilities by each tool to the vulnerability detected in this study is demonstrated in Table 6.9. The vulnerabilities identified in this study and reported by the respective tool are marked with ✓. If the study identified the vulnerability but is not reported by the tool is marked with O, otherwise marked with ✗. The vulnerabilities fixed in any compiler version were removed from the comparison.

Table 6.9: Comparison of vulnerabilities detected by each tool and study

| **Vuln\Tool** | **Remix Analyzer** | **Slither** | **Mythril** | **Securify** | **SmartCheck** | **Solhint** | **ContractGuard** |
|---|---|---|---|---|---|---|---|
| **Complex Fallback** | O | O | | O | | ✓ | ✓ |
| **Freezing ether** | | | | ✓ | ✗ | | ✓ |
| **Function order** | | | | | | ✓ | ✓ |
| **Gasslend send** | O | ✓ | O | O | ✓ | O | ✓ |
| **Mark callable contracts** | | | | | | ✓ | ✓ |
| **Payable fallback** | | | | | | ✓ | ✓ |
| **Reason string** | | | | | | ✓ | ✓ |
| **Single quotes** | | | | | | ✓ | ✓ |
| **SWC-101 Integer Overflow and Underflow** | | | ✓ | | | | ✓ |
| **SWC-102 Outdated Compiler Version** | | ✗ | | ✗ | | ✗ | |
| **SWC-103 Floating Pragma** | | ✓ | | | ✓ | | |
| **SWC-105 Unprotected Ether Withdrawal** | | | ✓ | ✓ | | | |
| **SWC-106 Unprotected SELFDESTRUCT Instruction** | ✗ | ✗ | ✓ | ✓ | | | |
| **SWC-107 Reentrancy** | ✓ | ✗ | ✓ | ✓ | | ✗ | ✗ |
| **SWC-108 State Variable Default Visibility** | | | | | O | O | O |
| **SWC-110 Assert Violation** | ✓ | | ✓ | | ✗ | | |
| **SWC-111 Use of Deprecated Solidity Functions** | O | ✓ | ✗ | | ✓ | ✓ | ✓ |
| **SWC-112 Delegatecall to Untrusted Callee** | O | ✓ | ✓ | ✓ | | | ✓ |
| **SWC-113 DoS with Failed Call** | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| **SWC-114 Transaction Order Dependence** | | | | ✓ | | | |
| **SWC-115 Authorization through tx.origin** | ✓ | ✓ | O | ✓ | O | ✓ | ✓ |
| **SWC-116 Block values as a proxy for time** | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| **SWC-117 Signature Malleability** | | | O | | | | |
| **SWC-120 Weak Sources of Randomness from Chain Attributes** | ✓ | | ✗ | | ✗ | ✓ | ✓ |
| **SWC-123 Requirement Violation** | ✗ | O | | | | | |
| **SWC-124 Write to Arbitrary Storage Location** | | ✗ | ✓ | ✓ | | | |
| **SWC-125 Incorrect Inheritance Order** | | | | | | | |
| **SWC-126 Insufficient Gas Griefing** | | | | | | | |
| **SWC-127 Arbitrary Jump with Function Type Variable** | ✓ | ✓ | O | ✗ | O | ✓ | ✓ |
| **SWC-128 DoS With Block Gas Limit** | ✓ | | | | ✗ | | ✗ |
| **SWC-130 Right-To-Left-Override control character** | ✓ | | ✓ | | | ✗ | |
| **SWC-131 Presence of unused variables** | | ✗ | | ✗ | | ✓ | ✓ |
| **SWC-132 Unexpected Ether balance** | | ✓ | | O | ✓ | | ✓ |
| **SWC-133 Hash Collisions With Multiple Variable Length Arguments** | | | | | | | |
| **SWC-134 Message call with hardcoded gas amount** | | | | | | | |
| **SWC-135 Code With No Effects** | | | | | | | |
| **SWC-136 Unencrypted Private Data On-Chain** | | ✓ | | ✓ | ✓ | | |
| **Unchecked Division** | ✓ | ✗ | | | | | |
| **Unitialized State** | | ✓ | | ✓ | | | ✓ |
| **Visibility Modifier Order** | | | | | | ✓ | ✓ |

As observed in table 6.9, the vulnerabilities that were least detected in the study and stated as detected by tools were SWC-116 and SWC-120. Slither and SmartCheck reported the most unexpected results. ContractGuard and Solhint are reported as the most accurate security tools.

## 6.6 Summary

The results of the research show that some of the security tools can detect more vulnerabilities than others. However, all of them can be used to validate smart contracts.

ContractGuard was the most effective and accurate security analysis tool, however it is not compatible with any other tool or framework. Mythril, Securify, Slither and Solhint combined can detect the higher number of vulnerabilities and are compatible with all other tools and frameworks, except Remix analyzer.

Other evidence indicates that tools are not prepared to deal with various types of approaches to the same vulnerability and, as a result, lack proper verification of vulnerabilities in smart contracts. The results show that if there are other errors in the code, or the code is not set in the structure required by the tool, it may vary the outcome of the analysis.

As a general conclusion, this research indicates that the use of security analysis tools in the development process of smart contracts would have significant benefits regarding the security verification and design of smart contracts. The use of security analysis guidelines would also improve the knowledge of developers about the implementation of design patterns and, as an effect, potentially avoid vulnerabilities.

# Chapter 7

# Framework Design and Implementation

This chapter defines the design and implementation of the solution proposed, gathering the required artefacts that constitute the structure and process of the solution.

## 7.1 Requirements Gathering

The objectives of this dissertation are defined to help the smart contract developers, so the main actors of this solution are the developers themselves, that mainly interact with the system and perform the functional operations. Researchers are also important stakeholders to the solution, that could have an interest in checking the vulnerabilities of smart contracts.

To identify the requirements of the solution, Figure 7.1 represents the use case diagram with the use cases that a smart contract developer or researcher can perform.

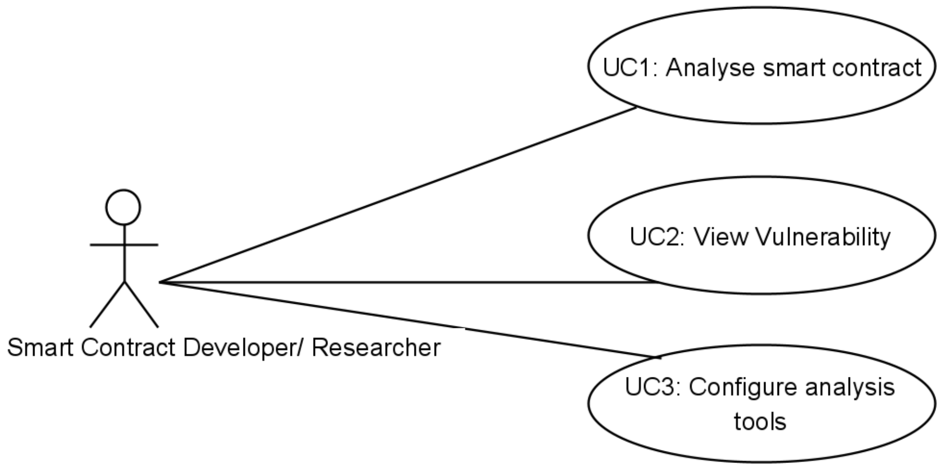

UC1: Analyse smart contract
UC2: View Vulnerability
Smart Contract Developer/ Researcher
UC3: Configure analysis tools

Figure 7.1: Use case diagram

The requirements behind the solution can be divided into functional and non-functional requirements. The functional requirements are illustrated in Table 7.1.

Table 7.1: Functional requirements

| Number | Description |
|---|---|
| 1 | The user should be able to analyse the security of a smart contract, written in Solidity or Vyper. |
| 2 | The system should integrate multiple security analysis tools. |
| 3 | The system should provide the user a configurable view to select the tools enabled to perform the security analysis. |
| 4 | The system should display information of the finding vulnerabilities in the respective code location. |
| 5 | The system should allow to install the integrated security analysis tools. |

Non-functional requirements can be described as the system quality attributes and behaviour of the solution. The following non-functional requirements were identified:

- The system should not outperform the sum of the isolated execution of all analysis tools.
- The system should provide failure-handling mechanisms to the analysis execution and tools installations.
- The interaction with the user must be simple, intuitive, and adaptive to smart contracts.
- The system should be compatible with a code editor that allows multiple development frameworks, analysis and testing tools and support multiple versions of Solidity and Vyper.
- The system should be exposed their functionalities from publicly available protocols.

## 7.2 Design alternatives

To create a suitable design for the development of the solution is necessary to consider the findings of the previous Chapter 6, declaring that the most effective security tool against the set of analysed smart contracts is ContractGuard. However, due to being only accessible on a web page [73] cannot be grouped and integrated with other development frameworks or CLI applications in the development process. Therefore, the most suitable identified tools are the combined analysis tools, Mythril, Securify, Slither and Solhint. With these tools, there are two possible approaches for the design:

The approach (1) showed in Figure 7.2 with the security analysis tools are used as distinct and separate extensions composed by a single server that handles the communication with the security analysis tools and the development framework through a single API.

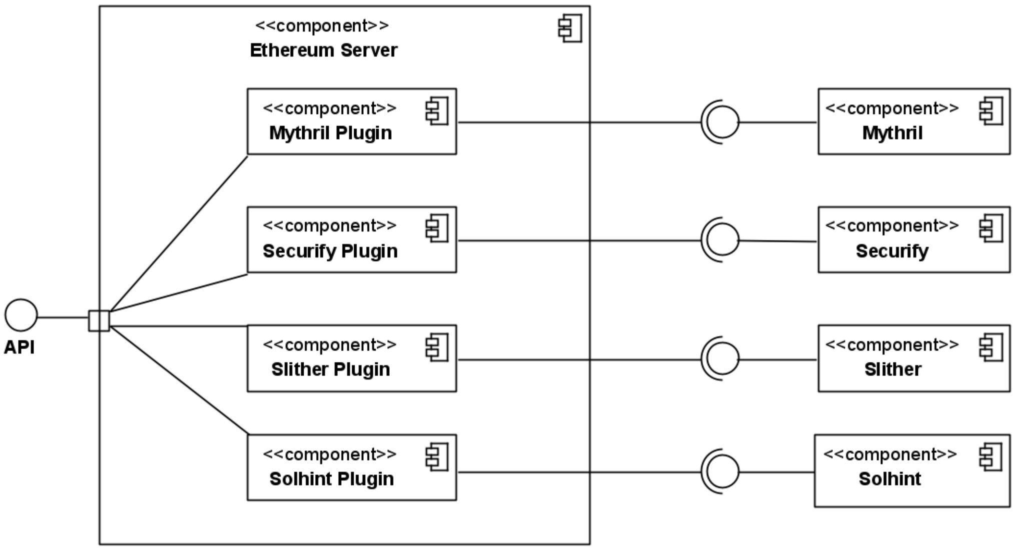


Figure 7.2: High-level design approach (1)

And the approach (2) showed in Figure 7.3 with a single server that exposes two API's through the Language Server Protocol (LSP) [101] and CLI protocols. The server would handle all the requests and determine which tools are needed for the execution. The CLI would enable the user to analyse the smart contract without other dependencies, and the LSP would allow the solution to connect with a code editor or IDE.

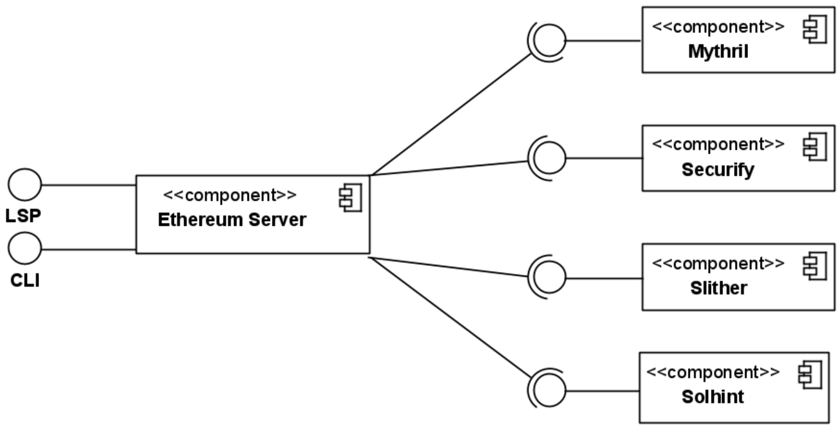


Figure 7.3: High-level design approach (2)

It is worth mention again that the approaches presented above were taken as a starting point to the development of the prototyped solution while using the research presented in Chapter 6. If a chosen security tool proves to be inadequate, or if one alternative tool is more suitable to the security of smart contracts, then, the solution should also have that alternative.

In the first approach, the Ethereum Server would have to run the tools in a parallel order to decrease the time to perform the analysis. However, it would not be possible to wait for all the tools to finish because the security tools are not interdependent, and a single call to the API can only execute an available plugin. In the second approach, the server would handle the parallel execution of each security tool and wait for all the tools to finish to retrieve the results. Based on this interpretation, the second approach is the optimal choice to develop the solution.

## 7.3 Design Approach

This section presents the design to capture the architecture and implementation of the selected approach.

### 7.3.1 Logical View

The high-level overview of the solution architecture is illustrated in Figure 7.4.

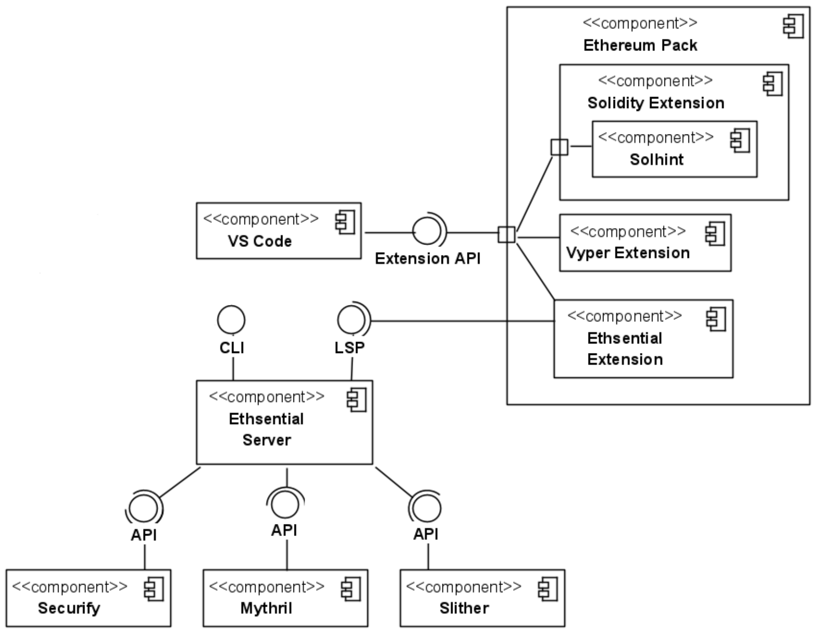


Figure 7.4: Framework logical view

As defined in Section 7.2, Figure 7.4 presents the solution architecture centred on a server application called Ethsential that exposes the two CLI and LSP protocols. The first directly used by the developer and the second used in Visual Studio Code (VS Code) extension functionality.

The solution is integrated with the VS Code editor to demonstrate the extendability of the solution to multiple development platforms. Through its popularity and features, VS Code provides an extension API [102] supporting numerous extensions to the Ethereum development and Solidity and Vyper languages. The Ethereum Extension pack (Ethereum pack) is a VS Code extension package that combines and installs all required plugins to smart contract development. The VS Code extensions used in the Ethereum pack are:

- EthSential Extension - Handle the logic required to execute the analysis of the smart contract and sends the requests to Ethsential Server through LSP protocol.

- Solidity Extension - Handle the analyses of smart contract's code written in Solidity with a Solhint integration.
- Vyper Extension - The language support for Vyper smart contracts.

The respective analysis tools (Mythril, Securify and Slither) are also a part of the architecture for the solution, integrated into EthSential Server. The integration with the analysis tools is described in the detailed design.

### 7.3.2 Detailed Design

After defining the main architecture, the detailed components Ethsential server and VS Code extensions are specified. Figure 7.5 presents the EthSential Server detailed design defined.

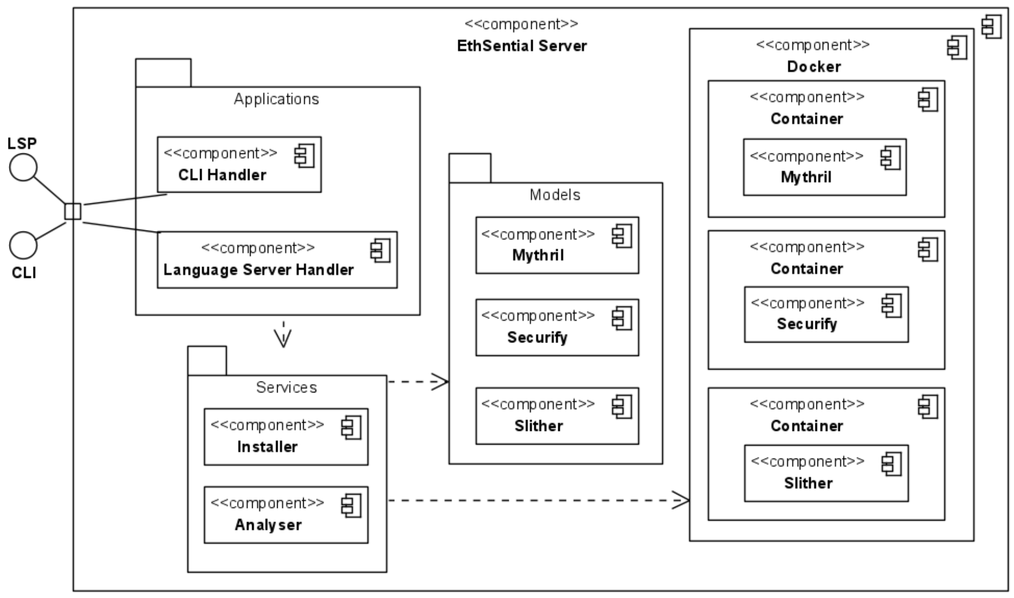


Figure 7.5: EthSential Server detailed design

As illustrated in Figure 7.5, the EthSential Server contains three layers:

- Application layer - programmatically exposes the command line interface or language server based on the functionalities required by the user and routes the commands to the respective handlers.
- Model layer - The definition of the business entities (Mythril, Securify and Slither) with the individual command and parser functionalities.
- Service layer - The components that handle the logical functionality of the application. Contains two main components, that allows the installation and analysis of the required tools following the user' specifications and the entities model commands.

Regarding the Docker component, it is designed to isolate the application environment of each security tool from the physical machine with a significant increase in security. A containerised tool helps to minimise the problems of any dependency or package necessary to its execution. Each security tool (Mythril, Securify and Slither) contains an image published in Docker Hub [103] that is loaded into the user machine during the execution of the installer service and is initialised in a container when analysing a contract (using the analyser service).

## 7.4 Implementation

In this section is presented the main implementation and coding decisions made along the development of the solution.

### 7.4.1 Development Workflow

The Ethsential Server is implemented using the version control Git [104] and is hosted on the platform Github [105].

The Git workflow used for the development of the application is the centralised workflow [106] with a single master branch only available to the repository owner. However, the community can add new contributions to the project using a feature branch workflow [107].

The development process is performed using the Continuous Integration practices under the Github actions, where the unit tests are run in each push or pull request to the master branch [108]. Only if the tests pass the integration can be concluded.

The project is published in the PyPi repository [109] and the VS Code extension in the Visual Studio Marketplace [110]. A release to these repositories is triggered when a Github Release is manually created that triggers a deployment Github action [111].

### 7.4.2 Ethsential Server

Ethsential is built with Python and relies on the Docker API [112] and pygls [113] to properly execute as a language server. The Docker API only supports its integration using one of the Go Software Development Kit (SDK), Python SDK or HTTP client. A language server can be implemented in any language; however, it can easily be used using one of the SDK's available (Go, C#, Java, PHP, Python, etc.) [101]. In this case, to integrate Docker API and pygls the Python SDK's were chosen.

The user has two options to initialise the Ethsential application:

- As a LSP server connected via stdin/stout or TCP:

  ```
  $ ethsent  #stdin/stout
  $ ethsent --tcp #tcp
  ```

- As a CLI application with the included file(s) and tools:

  ```
  $ ethsent --cli --file file.sol --tool all
  ```

The commands available are grouped by actions that represent the options to initialise the application via CLI or LSP. The commands are defined in Table 7.2.

Table 7.2: Ethsential command overview

| Group | Command | Alias | Description |
|---|---|---|---|
| 1 | tcp | | Use TCP server. |
| 1 | --port | -p | Bind tcp to port (default=2087). |
| 1 | --host | | Bind tcp to port (default=127.0.0.1). |
| 2 | cli | | Use cli tool. |
| 2 | --file | -f | Select file(s) or directories to be analysed. |
| 2 | --tool | -t | Select tool(s) (default=all). tools=[all, mythril, securify, slither] |
| 2 | --outputPath | -op | The output full path, relative to the current workspace. (default=result/) |
| 2 | install | i, isntall, add | Install tools via cli. |

After initialising the application as an LSP server, its features are available to execute (Listing 7.1). The server has three features, the CMD_ANALYSE (line 6) when a request is sent to analyse a contract, the feature CMD_INSTALL (line 10) if the user chooses to install the tools and a mandatory feature to register the configuration changes (line 14).

```
 1
 2  class EthSencialLS(LanguageServer):
 3      CMD_ANALYSE = 'analyse'
 4      CMD_INSTALL = 'install'
 5
 6  @ETHSENTIAL.feature(EthSencialLS.CMD_ANALYSE)
 7  async def doAnalysisFeat(ls: EthSencialLS, params):
 8       #...
 9
10  @ETHSENTIAL.feature(EthSencialLS.CMD_INSTALL)
11  async def doInstallFeat(ls: EthSencialLS, *args):
12      #....
13
14  @ETHSENTIAL.feature(WORKSPACE_DID_CHANGE_CONFIGURATION)
15  async def didChangeWorkspace(ls: EthSencialLS, *args):
16      #...
```

Listing 7.1: EthSential supported features

Each tool (Mythril, Securify and Slither) contains a business entity defined by inheriting a class called Tool (Listing 7.2). This pattern allows any derived tool to acquire all the properties and behaviours of a Tool.

```
class Tool(ABC):
    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def parse(self, str):
        pass

    @property
    def image(self):
        raise NotImplementedError

```

```
14      @property
15      def command(self):
16          raise NotImplementedError
17
18      @property
19      def lang_supported(self):
20          raise NotImplementedError
```

Listing 7.2: Abstract Tool class

A tool is defined by the command (line 15), image (line 11), and language support properties (line 19) and the parse method (line 7) called during the analysis process.

The creation of the required tools is defined by a tool factory class (Listing 7.3). This class chooses the tools to create (line 5) based on the selected options available (line 3). If the selected option is not available will prompt an error (line 15).

```
1   class ToolFactory():
2
3       TOOLS_CHOICES = ['all', 'mythril', 'securify', 'slither']
4
5       def createTool(tool_type):
6           if tool_type == "mythril":
7               return [Mythril()]
8           if tool_type == "securify":
9               return [Securify()]
10          if tool_type == "slither":
11              return [Slither()]
12          if tool_type == "all":
13              return [Mythril(), Securify(), Slither()]
14          raise ValueError(format)
15      createTool = staticmethod(createTool)
```

Listing 7.3: ToolFactory class

An analysis of a contract, either triggered by the LSP or CLI, is analysed (Listing 7.4) using the file path, language and selected tools (line 16). Before performing the analysis in the containerised security tool, the application mounts the file path to a specific volume and extracts the contract language version from the file (lines 17-18). The analysis of the contract is run in a parallel execution for each selected tool using the multiprocessing Python pool (lines 19-23). In the analysis execution function (lines 1-14), the tool command property is formatted to the executing file and language version (line 4). The execution in the docker container (line 6) uses the tool image, command and mounted volume to start. The result of the container execution is then parsed accordingly to the tool parsing functionality (lines 8-9).

```
1   def start_container(filePath, lang_version, volume_bindings, tool):
2       container = None
3       try:
4           cmd = tool.command.format(contract='/analysis/' + os.path.basename
        (filePath).replace(
5                   '\\', '/'), version=lang_version)
6           container = client.containers.run(tool.image,cmd,detach=True,
        volumes=volume_bindings)
7           container.wait(timeout=(30 * 80))
8           output = container.logs().decode('utf8').strip()
```

```
        return tool.parse(output)
    except Exception as exception:
        return {"sucess": False, "exception": exception}
    finally:
        stop_container(container)
        remove_container(container)

def analyse_file(filePath, lang, tools):
    volume_bindings = mount_volumes(os.path.dirname(filePath))
    lang_version = get_lang_version(filePath, lang)
    pool = multiprocessing.Pool()
    results = []
    results = [pool.apply(start_container, args=(
        filePath, lang_version, volume_bindings, tool)) for tool in tools]
    pool.close()
    return results
```

Listing 7.4: Contract analysis method

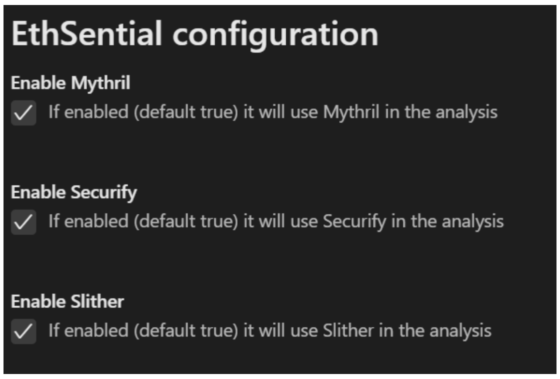
### 7.4.3 Ethsential Extension

As defined in Section 7.3, VS Code allows creating custom extensions through a dedicated API. A custom extension relies on two main files, the *extension.ts* file, to handle all the extension logic and, the *package.json* file [114], to register all functionalities and activation events of the extension. In Ethsential extension (Listing 7.5) there are two main functionalities written as command points of the extension, the analysis command, to perform the security analysis of a file and installation command, to install all the tools available. The extension is activated when a Solidity or Vyper file is opened (lines 4-5) or one of the provided commands is executed (lines 6-7).

```
1  {
2    ...
3    "activationEvents": [
4      "onLanguage:solidity",
5      "onLanguage:vyper"
6      "onCommand:ethsential.analyse",
7      "onCommand:ethsential.install"
8    ],
9    "contributes": {
10     "commands": [
11       {
12         "command": "ethsential.analyse",
13         "title": "EthSential: Analyse File"
14       },
15       {
16         "command": "ethsential.install",
17         "title": "EthSential: Install Security Analysis Tools"
18       }
19     ]
20   ...
21 }
```

Listing 7.5: Register extension events, commands and configuration

The configuration of the active security tools is also a feature required in the solution. This feature works as a configuration property for each tool, displayed as a checkbox, in the user preference settings under the EthSential configuration as illustrated in Figure 7.6.



EthSential configuration

**Enable Mythril**
✓ If enabled (default true) it will use Mythril in the analysis

**Enable Securify**
✓ If enabled (default true) it will use Securify in the analysis

**Enable Slither**
✓ If enabled (default true) it will use Slither in the analysis

Figure 7.6: EthSential Tools configurations

After the extension is activated, a subscription to the language server is started (line 13 in Listing 7.6) and the EthSential commands are registered for execution (lines 14-15).

```
 1  export function activate(context: ExtensionContext) {
 2    async function analyse(editor: TextEditor) { ...  }
 3
 4    async function install(){ ... }
 5
 6    //...
 7
 8    context.subscriptions.push(
 9      client.start(),
10      commands.registerTextEditorCommand('ethsential.analyse', analyse),
11      commands.registerCommand('ethsential.install', install)
12    );
13  }
```

Listing 7.6: Extension activation function

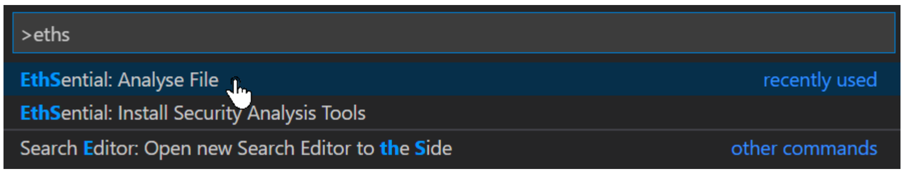
When a command, like ethsential.analyse (line 14 in Listing 7.6), is triggered, the respective registered function is executed by sending an asynchronous request directed to the server (e.g. Listing 7.7). The *sendRequest* method (line 1) represents the connection to the server. It requires a parameter called *RequestType* that defines two interfaces, the *ActiveAnalysis-Params* and the *ToolCommandOutput*, describing the parameters and response obtained from the language server. The parameters are defined in lines 4-6 using the document *URI*, language and tools to run.

```
1  let result: ToolCommandOutput[] = await client.sendRequest( new
     RequestType<ActiveAnalysisParams, ToolCommandOutput[], void, void>( '
     analyse'),
2        {
3          textDocument: { uri },
4          lang: editor.document.languageId,
5          tools,
6        }
```

```
        );
```

Listing 7.7: Analysis request to the Language Server

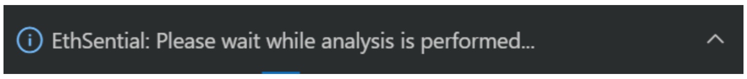
Figure 7.7 shows the selection of available EthSential operations in the command palette of the VS Code. In this case, the solidity file is already opened in the VS Code and the analysis of the file is selected.



```
>eths
EthSential: Analyse File                                          recently used
EthSential: Install Security Analysis Tools
Search Editor: Open new Search Editor to the Side                other commands
```

Figure 7.7: VS Code command selection to analyse file

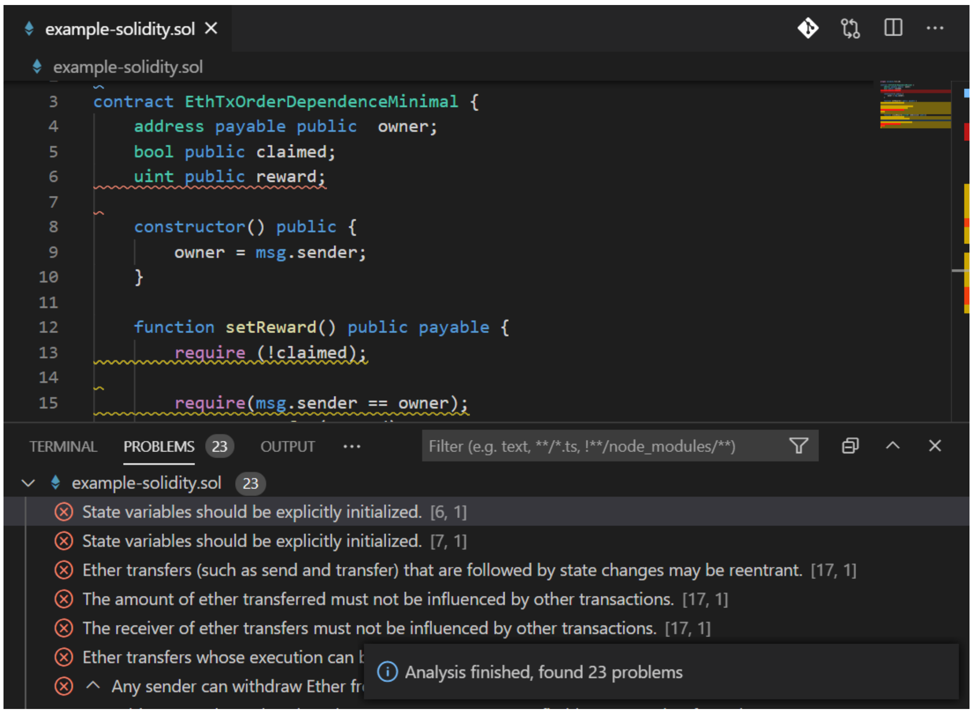
After the analysis option is selected, a window with the information of the analysis progress is shown (Figure 7.8). This progress window only concludes when the analysis is completed.



EthSential: Please wait while analysis is performed...

Figure 7.8: VS Code analysis in progress information

Once the analysis is completed, the normalised result of the language server analysis is presented to the user, as demonstrated in Figure 7.9. The analysis is presented with an informational window containing the total vulnerabilities found. Every single vulnerability detected is present as a code diagnostic in the source code as a native VS Code information/warnings/error and shown in the problems section with the vulnerability details.



example-solidity.sol
example-solidity.sol

```
contract EthTxOrderDependenceMinimal {
    address payable public  owner;
    bool public claimed;
    uint public reward;

    constructor() public {
        owner = msg.sender;
    }

    function setReward() public payable {
        require (!claimed);

        require(msg.sender == owner);
```

TERMINAL PROBLEMS 23 OUTPUT ...
Filter (e.g. text, **/*.ts, !**/node_modules/**)
example-solidity.sol 23

```
State variables should be explicitly initialized. [6, 1]
State variables should be explicitly initialized. [7, 1]
Ether transfers (such as send and transfer) that are followed by state changes may be reentrant. [17, 1]
The amount of ether transferred must not be influenced by other transactions. [17, 1]
The receiver of ether transfers must not be influenced by other transactions. [17, 1]
Ether transfers whose execution can b
Any sender can withdraw Ether fr
```

Analysis finished, found 23 problems

Figure 7.9: VS Code analysis completed information

This mapping from the analysis is performed using a specific class called *DiagnosticProvider*, as represented in Listing 7.8. This class is instantiated when the extension is activated and called after an analysis is called to the language server. When is instantiated the constructor is called (line 6) and the *fileDiagnosticMap* and *diagnosticCollection* are initialised, representing respectively the diagnostics obtain in each file and the diagnostics to show to the user. The method *refreshDiagnostics* handles the mapping of the outcome of the analysis into the *diagnosticCollection* and calls the private methods of the class.

```
1  export class DiagnosticProvider {
2    private fileDiagnosticMap: Map<Uri, Diagnostic[]>;
3    private diagnosticCollection: DiagnosticCollection;
4    numberOfProblems = 0;
5
6    constructor(collectionName: string) {
7      this.fileDiagnosticMap = new Map<Uri, Diagnostic[]>();
8      this.diagnosticCollection = languages.createDiagnosticCollection(
     collectionName);
9    }
10
11   public refreshDiagnostics(document: TextDocument, outputResults:
     ToolCommandOutput[]) { ... }
12
13   private gatherDiagnostics(issues: ToolCommandIssue[], document:
     TextDocument
14   ) { ... }
15
16   private createDiagnosticResult( textRange: Range, issue:
     ToolCommandIssue, severity: DiagnosticSeverity) { ... }
17
18   private getTextRange(document: TextDocument, lines: Number[]) { ... }
19
20   private getDiagnosticSeverity(severity: string) { ... }
21
22   private clearDiagnostic(diagnosticCollection: DiagnosticCollection) {
     ... }
23 }
```

Listing 7.8: DiagnosticProvider representative class

The EthSential operations to install the security analysis tools, uses the enabled tools of the user preference settings and sends an installation command to the language server. The installation begins by showing a progress information window, in the same way of the analysis operation, and finally, when the installations are completed, display the respective information.

## 7.4.4 Ethereum Pack

The Ethereum Pack consists of a VS Code extensions pack that bundles the Ethsential extension with the Solidity and Vyper languages support. These extensions are defined in the *package.json* under the *extensionsPack* (Listing 7.9).

```
  "extensionPack": [ "JuanBlanco.solidity",
   ↪ "tintinweb.vscode-vyper", "1140251.ethsential"]
```

Listing 7.9: Ethereum Pack extensions

### 7.4.5 Development Testing Setup

Following the specification of the Software Development Life Cycle (SDLC) [115], any development process must be accompanied by the testing process. There are various types of testing, which can be done to find bugs and errors within software or even functional flaws in the specification of the implemented requirements.

In this solution, the Ethsential extension and the language server were developed accompanied by the VS Code debugging functionality and the specific unit software testing.

#### Debug Ethsential Language Server and CLI

The Ethsential language server can be debugged by running the VS Code debug mode (Listing 7.10). The debugging itself is a standard Python debugging that interprets the installed Python version (line 8) and executes the server module folder (line 5) with the TCP argument (line 6).

```
1  {
2        "name": "Launch Server",
3        "type": "python",
4        "request": "launch",
5        "module": "ethsential",
6        "args": ["tcp"],
7        "justMyCode": false,
8        "pythonPath": "${command:python.interpreterPath}",
9        "cwd": "${workspaceFolder}",
10       "env": {
11         "PYTHONPATH": "${workspaceFolder}"
12       }
13 }
```

Listing 7.10: Command to start Ethsential application for TCP connection inside VSCode

The Ethsential CLI application is debugged using the same standard Python debugging and adding CLI action. The command specifies the file to execute and the argument to execute all tools (e.g. *cli –file server/examples/example-solidity.sol -t all*).

#### Debug Ethsential VS Code Extension

The Ethsential VS Code extension is started in debug mode by running the Typescript compiler in watch mode. The commands to start and compile the extension ready to deploy are specified in the *package.json* file [114]. As seen in Listing 7.11, the launch in debug mode is started by running a VS Code extension host (line 2) with the compiled source code in a workspace folder (lines 7-10).

```
1  {
2        "type": "extensionHost",
3        "request": "launch",
4        "name": "Launch Client",
      "runtimeExecutable": "${execPath}",
      "args": ["--extensionDevelopmentPath=${workspaceRoot}"],
      "outFiles": ["${workspaceRoot}/vscode-client/out/**/*.js"],
      "preLaunchTask": {
        "type": "npm",
        "script": "watch"
      },
      "env": {
        "VSCODE_DEBUG_MODE": "true"
      }
}
```

Listing 7.11: Command to start Ethsential extension inside VSCode

### Unit Testing

Unit testing is the first testing method used in this project. It focuses on testing individual units of source code and is an important component to detect code flaws missed during the development process.

In this project, the unit tests are performed for the main component of the solution, the EthSential language server. The component is tested using the Behaviour Driven Development (BDD) following the Arrange-Act-Assert (AAA) pattern. This pattern states that each section (act, arrange and assert) is responsible for a specific part of the test: the arrange is responsible for the code required for the test setup (inputs); the act is responsible for the invocation of the methods being tested; And finally, the assert, responsible for the verification of the outcome of the methods called.

The unit tests in the language server were developed using the built-in Python *unittest* framework. An example of these unit tests is provided in Listing 7.12 and represents a test case of the *ToolFactory* class. The objective of this class is to test the possible outcomes of the method *createTool*: retrieve one tool (lines 3-6), all tools (lines 8-11) or even throw an error if the tool name is not supported (lines 13-18). Like this listing, all other unit tests developed within the scope of the language server follow the same approach to test all the possible outcomes of a method or class.

```
1  class ToolFactoryTest(unittest.TestCase):
2      def test_create_mythril_pass(self):
3          tool_name = 'mythril'
4          tools = ToolFactory.createTool(tool_name)
5          self.assertIsInstance(tools[0], Mythril)
6      def test_create_all_pass(self):
7          tool_name = 'all'
8          tools = ToolFactory.createTool(tool_name)
9          self.assertEqual(len(tools), 3)
10     def test_create_tool_fail(self):
11         try:
12             tool_name = 'tool'
13             _ = ToolFactory.createTool(tool_name)
14         except Exception as identifier:
15             self.assertIsInstance(identifier, ValueError)
```

Listing 7.12: Tool Factory test class

# Chapter 8

# Evaluation

This chapter aims to evaluate the solutions implemented by defining the evaluation indicators, their hypotheses, the methods used and respective results.

## 8.1 Indicators

To perform the evaluation of the solution is mandatory to define the evaluation indicators. "An indicator is a documentable or measurable piece of information regarding some aspect of the program in question" [116]. In order to do so, the indicators must be defined based on the requirements and objectives of the solution. The main indicators defined are:

- Compliance with functional requirements - Any software should be compliant and performed accordingly to the functional requirements and use cases defined.
- Performance - Verify the performance of the solution in terms of response times compared to the response times of each tool.
- User satisfaction and usability - Evaluate the usability and experience of smart contract developers when using the solution.

## 8.2 Hypothesis Specification

The definition of the hypothesis is an important requirement to measure the defined indicators and assess the achievement of the proposed objectives.

Each indicator contains an hypothesis of two opposing statements [117]. The Null hypothesis testing ($H_0$) is the formal argument while the second statement, called the alternative hypothesis ($H_1$) is the rejection of $H_0$. The following Table 8.1 describes the hypothesis for each indicator.

Table 8.1: Hypothesis Specification

| Indicator | $H_0$ | $H_1$ |
|---|---|---|
| Compliance with functional requirement | The solution does not have the requirements required | The solution fulfils the requirements required |
| Performance | The solution is less performant than the sum of the execution of each tool | The solution is more performant than the sum of the execution of each tool |
| User satisfaction and usability | The satisfaction of smart contract developers is less then 80% | The satisfaction of smart contract developers is greater than or equal to 80% |

The defined hypotheses for the compliance with functional requirements is defined based on the requirements to obtain a fast and integrated solution. Since the integrated solution would need to be adapted to different tools, is necessary to evaluate if the solution would not outperform the sum of executions of each tool. Regarding the satisfaction score of 80% for the user satisfaction, this score is based on the Customer Satisfaction Score (CSAT) [118], that states that an user to be satisfied requires at least an 80% score.

## 8.3 Evaluation Methods

This section presents the evaluation methods of the identified indicators. There are three methods used to perform the evaluation: software testing, performance evaluation and, satisfaction and usability questionnaires. Table 8.2 shows the allocation of these methods in each indicator.

Table 8.2: Allocation of the evaluation methods

| Indicator | Method |
|---|---|
| Compliance with functional requirement | System and Acceptance Testing |
| Performance | Performance Testing |
| User satisfaction and usability | Satisfaction and Usability Questionnaire |

### 8.3.1 System and Acceptance Testing

As already mentioned in Section 7.4.5, the development of an application should always be connected and paired with both the manual and the automatic application tests. The commonly used stages of software testing are: unit testing (section 7.4.5), integration testing, system testing and acceptance testing [119]. In this evaluation, the stages of software testing performed were the system and acceptance testing.

### 8.3.2 Satisfaction and Usability Questionnaire

The satisfaction and usability survey aims to assess the user' satisfaction and agreement with the developed solution, in order to verify that it meets their needs.

The questionnaire should have a compact structure with clear questions, avoiding ambiguity, and uncertain answers. The answers should have up to three different response scales: Dichotomous scales, using two choices answers; Rating scales, using the Likert scale by defining answers in 5 levels (Strongly disagree, Disagree, Neither agree nor disagree, Agree and Strongly agree); and Semantic differential scales, providing a paired choice of clearly opposite statements.

Considering that usability is one of the critical requirements and indicators, the questionnaire should focus on evaluating the interaction of the targeted users (experienced and inexperienced developers) on using the tools with specified guidelines. The opinion of these users has an huge importance to understand the feasibility of this solution. Addressing the experience of smart contract developers can be a difficult task, so the guidelines provided should be explicit, allowing experienced and inexperienced developers to test the solution and provide their feedback.

### 8.3.3 Testing Evaluation Based on Performance

Creating a new framework that is involved in the development process happens with the intention to improve it by removing or reducing the manual tasks that the user has to perform. Reducing the manual tasks allows for a faster and more efficient experience for the user. According to Jakob Nielsen [120], the time limit for a user to shift their attention from the application to another task is about 10 seconds. Any time above that will not provide a good performance of the application, therefore it is necessary to test the EthSential framework in this matter.

Since EthSential itself is integrated in the development process and possesses other security analysis tools to analyse smart contracts, the overall response times must be analysed. So, to understand how the solution will perform on Ethereum, tests must be applied to verify the response times of each tool against the response times of the EthSential framework.

## 8.4 Results Evaluation

In this section, the indicators are evaluated against the defined hypotheses, demonstrating the results obtained from each evaluation method performed.

### 8.4.1 System Testing

System testing is the stage of the software testing where the system is tested as a whole, gathering a clear representation of the requirements and testing all possible results. Table 8.3 shows a system test case performed to analyse a smart contract in VS Code. According to the expected and actual result obtained, the system is capable of analysing a file using VS Code integration. The complete list of system tests is available in Appendix B.

Table 8.3: System Test Case to Analyse Smart Contract from VS Code

| | |
|---|---|
| **ID** | 12 |
| **Description** | Analyse smart contract from VS Code |
| **Steps** | 1. Open VS Code<br>2. Open example-solidity.sol<br>3. Run command - EthSential: Analyse File |
| **Expected Result** | 1. Show message - Analysis finished<br>2. Show diagnostics |
| **Passed/Failed** | 1. Passed<br>2. Passed |

### 8.4.2 Acceptance Testing

Acceptance tests are intended to determine if the delivered system satisfies the acceptance criteria defined by the customer's requirements. Table 8.4 shows an acceptance test regarding the analysis of a smart contract with specific tools selected and installed. To determine if an acceptance test is valid, a set of criteria are identified in which they must fully comply with the system requirements after executing the test case.

Table 8.4: Acceptance Test to Analyse a Smart Contract with Specific Tools Installed

| | |
|---|---|
| **Scenario** | Analysis a smart contract with specific tools installed |
| **Assumptions** | 1. VS Code installed with extension<br>2. Docker running |
| **Criteria** | 1. Mythril is not installed<br>2. Show message - Installation completed finished<br>3. Show message - Analysis finished<br>4. Show code |
| **Test Case** | 1. Open VS Code<br>2. Open example-solidity.sol<br>3. Open the user preference settings<br>4. Deselect Mythril<br>5. Run command - EthSential: Install Security Analysis Tools<br>6. Run command - EthSential: Analyse File |
| **Result** | All criteria were met |

### 8.4.3 Performance Testing

The performance is evaluated with a test to determine the execution time of smart contract evaluations in the EthSential framework, through the command line. The test was performed five times using the 40 vulnerabilities obtained in table 6.9 on a computer with an Intel® Core$^{TM}$ i5-8265U .60GHz 1.80GHz CPU and an average WiFi download speed of 556.16 Mbps.

The results of all instances of the test are shown in Appendix C, with the mean and median of all the results present in Table 8.5.

Table 8.5: Mean and Median of the Execution Time Test Results

| | Slither | | Securify | | Mythril | | All |
|---|---|---|---|---|---|---|---|
| | Before | After | Before | After | Before | After | - |
| Mean | 1.025 | 4.05 | 1.275 | 4.1 | 69.05 | 72.6 | 81.35 |
| Median | 1 | 4 | 1 | 4 | 20 | 21.5 | 24 |

Through the analysis of Table 8.5, it is possible to observe that for Slither and Securify both have a mean and median approximately equal. This evidence indicates that there is not much variation in the results obtained for these tools. However, it also shows that for both tools and even Mythril, despite the higher variation of mean and median results, the EthSential integration requires more time (approximately three seconds more) to evaluate a smart contract than the original tool. Concluding that original tools performed in separate are more effective than the Ethsential framework integration.

So, the final evaluation of Ethsential performance is to verify if the sum of executions of each tool outperforms the execution of EthSential with all tools. In order to verify this case, it is necessary to analyse the tool to check whether the data samples follow a normal distribution. For this case, a Shapiro-Wilk normality test [121] is applied, with the following hypotheses:

- $H_0$: If the P-Value of the Shapiro-Wilk Test is larger than 0.05, it must be assumed that it is a normal distribution.
- $H_1$: If the P-Value of the Shapiro-Wilk Test is smaller than 0.05, it must be assumed that it is not a normal distribution.

The following Table 8.6 shows the resulting P-Value of the Shapiro-Wilk Test.

Table 8.6: Resulting P-Values of the Shapiro-Wilk Tests

| | Slither | | Securify | | Mythril | | All |
|---|---|---|---|---|---|---|---|
| | Before | After | Before | After | Before | After | - |
| P-Value | 8.68E-14 | 8.68E-14 | 7.18E-10 | 9.90E-11 | 1.48E-12 | 1.41E-12 | 1.22E-12 |

The table shows that all p-value values are less than 0.05, so the $H_0$ hypothesis is rejected. Therefore, it is possible to affirm with a 95% confidence level that the analysed samples do not follow a normal distribution. Concluding, to compare the performance of the newly developed EthSential, a 2-sample T-Test [122] cannot be used, and only a non-parametric test can be performed. The Mann-Whitney U test is used for this scenario with the following hypotheses:

- $H_0$: The tool's execution time performance is higher than the tool in EthSential.
- $H_1$: The tool's execution time performance is equal or lower than the tool in EthSential.

As a result of the Mann-Whitney U test [123], the p-value is 0.899, rejecting the alternative hypothesis, therefore the EthSential framework as a higher performance.

The final performance test is performed to analyse if the EthSential framework has an overall response time lower or equal to 10 seconds. So the hypotheses are:

- $H_0$: The execution response times are lower or equal to 10.
- $H_1$: The execution response times are higher than 10.

This test is performed using the Wilcoxon signed rank test [124] and resulting p-value is 2.067e-08, rejecting the $H_0$ hypothesis. This means that the application has a low performance in terms of response times to the user.

### 8.4.4 Satisfaction and Usability Questionnaire

To analyse the satisfaction and usability of the solution, a questionnaire, which can be found in Appendix D, was developed using the Microsoft Forms platform. This means that it was accessible to professionals in the field of software development with interest in the development of smart contracts, and launched to the Ethereum community through forums and network platforms.

The questionnaire had a total of 18 questions divided into two question groups:

- Questions regarding the background of the interviewee in Smart Contracts development in order to verify if the interviewee has some knowledge and experience in developing smart contract
- Questions regarding the solution to validate the usability, using the values of the Likert scale, and satisfaction, using the 10 point likert scale and an open-ended question.

The questionnaire was conducted with 5 participants, in order to assess the level of usability and satisfaction regarding the developed solution. Despite the small sample of interviewees, the analysis of these data was considered to evaluate the usability of the solution.

#### Interviewees Knowledge

To evaluate the interviewees knowledge, the closed-ended questions results were analysed (Table 8.7).

Table 8.7: Answers to questions 1,2,3 and 5

| Question 1 | Question 2 | Question 3 | Question 5 |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |

The internal consistency of the interviewees, that is, how closely they are related to smart contracts, was evaluated with the Cronbach's Alpha [125] test on the closed-ended questions. The reported alpha coefficient of the test is 0.875, meaning that all answered questions are closely related and assuming that the 5 interviewees answered almost the same questions, can be concluded that 3 of the 5 interviewees have knowledge and awareness of security smart contracts.

Regarding the multiple choice question number 4 only two interviewees answered. Figure 8.1 provides the security analysis tools selected in this question. While Mythril and Remix are the most popular tools, Securify and Slither were not chosen even once.

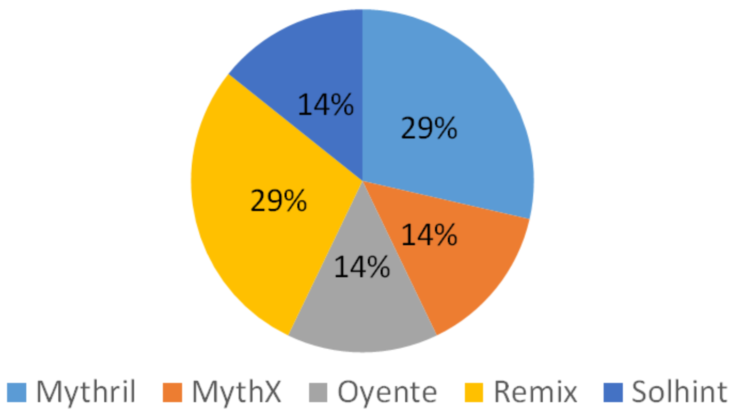


Figure 8.1: Chart of security analysis tools selected

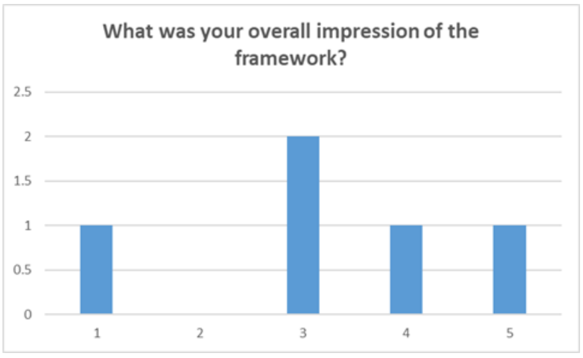
Since only one interviewee answered the open-ended question with Remix and Mythril, there is no possible analysis for this question.

### Solution Evaluation

This section presents the results obtained in the questionnaire regarding the usability of the solution.

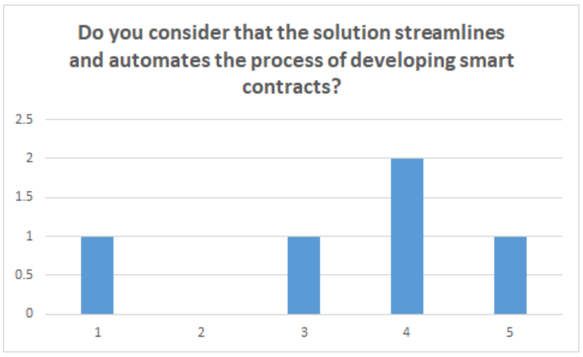
The answers obtained in questions 7 to 13 were classified according to the Likert scale, where the maximum value defined is 5 (Strongly agree) and the minimum value is 1 (Strongly disagree). The results of each question are presented with the respective interpretation.

Figure 8.2: Overall impression of the framework



The overall impression of the interviewees (Figure 8.2) is classified with a positive mean score of 3.2 and a median of 3. Despite the positive mean score, the results show that the solution needs to be analysed for further improvements.

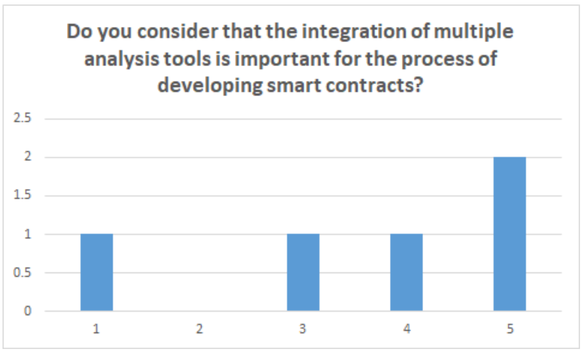
Figure 8.3: Classification of the solution automation



The assessment of the automation and streamlining of the solution (Figure 8.3) obtained a mean score of 3.4 and a median of 4, which represents a positive assessment.

The next question (Figure 8.4) aimed to frame the interviewee' opinions on the integration of security tools.

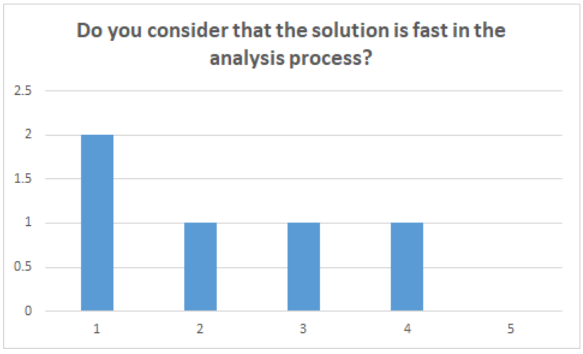
Figure 8.4: Classification of the integration of analysis tools



Analysing the graph of Figure 8.4, the interviewees classified the integration of analysis tools with a mean score of 3.6, being adequate and sufficient to the development process.

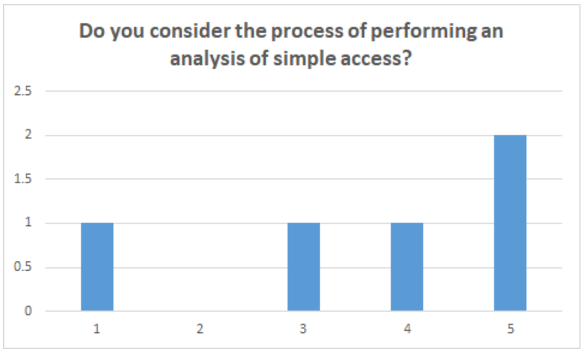
When questioned about the speed of the analysis process, the interviewees answered that the process has a tendency to be slow (Figure 8.5). No interviewee answered with a grade 5, meaning that no interviewee is completely happy with the analysis process speed.

Figure 8.5: Classification of the analysis process speed



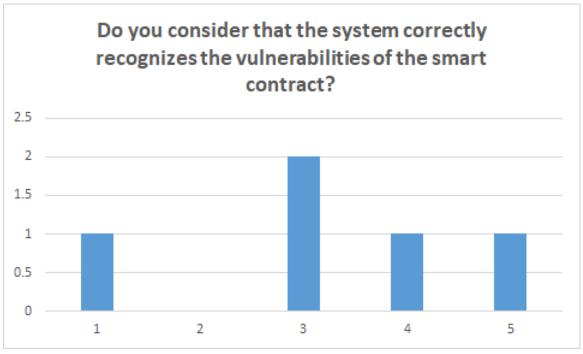
In Figure 8.6, the average interviewee classified the analysis process as simple to use. While only one interviewee provided a grade 1, all the others provided higher grades, positioning the simplicity of the analysis process with a mean of 3.6.

Figure 8.6: Classification of the analysis process simplicity



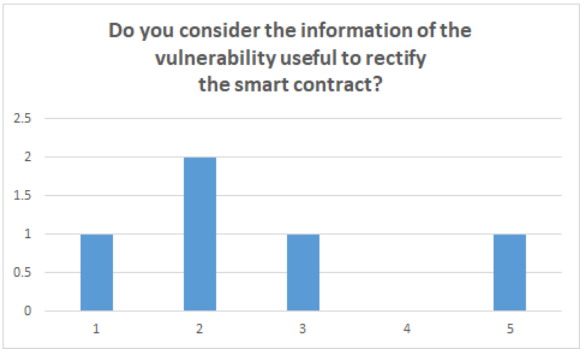
Regarding the vulnerability recognition (Figure 8.7), the majority of the interviewees classified the solution with a grade 3. Results show a mean of 3.2, which translates in a positive score.

Figure 8.7: Classification of the vulnerability recognition



In the final Likert scale question (Figure 8.8), interviewees evaluated the usefulness of the vulnerability information. These results show that the vulnerability information is not sufficiently useful to rectify the smart contract.

Figure 8.8: Classification of the vulnerability information usefulness



Another key analysis is to verify if the interviewees with experience in developing smart contracts have a tendency to give a lower score than interviewees without experience. In this analysis the Likert scale questions were divided into two groups based on the response provided by question 2 (Have you previously developed smart contracts?). The test used to performed this analysis is the Mann-Whitney U test with the following hypotheses:

- $H_0$: The answers of experienced developers are lower or equal to the answers of inexperience developers.
- $H_1$: The answers of experienced developers are higher than the answers of inexperience developers.

The result of the test shows a p-value of 0.999856, hence the $H_0$ is accepted, stating with 5% confidence that experienced developers have a tendency to give a lower score than inexperienced developers.

Regarding question 15, all interviewees unanimously answered that no tool should be added or removed from the framework. Also, none of the interviewees provided an answer to the open-ended questions 14 and 16

The last question analysed is a 10 point Likert scale question with a scale from 1 (not likely at all) to 10 (extremely likely). In this question, the Mann-Whitney U test was applied with the hypotheses:

- $H_0$: Experienced developers responded with a lower or equivalent score than inexperience developers.
- $H_1$: Experienced developers responded with a higher score than inexperience developers.

The Mann-Whitney U test results in a p-value of 0.5, rejecting the $H_1$ hypothesis. This result demonstrates that inexperience developers have a higher tendency to use the framework in the future.

# Chapter 9

# Conclusions

This chapter presents the conclusions drawn from the study on the practices of developing smart contracts and the development of the EthSential prototype, namely the achieved objectives, the limitations identified and finally, the improvements that can be made in the future.

## 9.1 Achieved Objectives

In this Section, the objectives are evaluated based on the corresponding achievements. The main goal of this thesis, as defined in Section 5.1, was to provide new practices for the Ethereum development by analysing the tools available in Ethereum, with the following objectives:

- Explore solutions to identify vulnerabilities and guidelines that provide a readable resolution of these vulnerabilities and potential improvements.
- Address the smart contract vulnerabilities and technologies, proposing a solution to ease the development of smart contracts using good practices and guidelines.

The first objective was achieved by analysing all the information available regarding security analysis tools. The research defined in Chapter 6 allowed to evaluate all the frameworks and, analysis and testing tools available in the market and open source. Of all these tools, the research identified seven that can detect vulnerabilities and provide insights on problems that occurred. By providing these insights, this dissertation has shown that security analysis tools can be integrated in the smart contract development process and can directly improve it by teaching the developers how to avoid the problems encountered and encourage them to use design patterns.

The second objective related to the development of the solution is achieved with the EthSential framework. By providing a framework that can integrate multiple security analysis tools, it allows developers to choose which tools they want to use, knowing that they can identify vulnerabilities and provide some information about present vulnerabilities. The framework integration with the command line and VS Code editor allows a good accessibility and availability of the framework for developers to use in the development of smart contracts.

Regarding the evaluation of the solution performed in Chapter 8, the results show that the solution fulfils all the functional and non-functional requirements. However, some of these requirements show that the solution could be improved regarding the analysis execution time and information provided in each vulnerability. Since experienced developers that filled out the questionnaire answered most of the questions with a lower score than inexperienced developers, the solution did not accomplish the overall satisfaction of 80%.

Concluding, and answering the main question - Is it possible to reduce the vulnerabilities and code smell in a smart contract code by integrating and combining tools in the development process? The answer is yes, there are some testing and security analysis tools available that could be integrated in the development process to increase the security of the smart contract.

## 9.2 Limitations

Despite the development effort to achieve the objectives and the successfully implementation and deployment of EthSential, there are some limitations that influenced the analysis of the tools and results of the application.

One of the biggest limitations of this project was the lack of standards to identify vulnerabilities. Most of the security analysis tools use different notations to identify the same type of vulnerability. This differences made it harder to identify and map the vulnerabilities accordingly to the SWC standards.

The constant evolution of Solidity means that the tools have to be prepared for new versions and adapt accordingly. The majority of the tools analysed were not prepared and adaptable to handle contracts with different versions. To perform the solution was necessary to install the specific Solidity smart contract to perform the analysis and thus an increase in the analysis execution time.

Regarding the EthSential development, there were some limitations identified. The EthSential framework is new and the community of Solidity developers can only be reached by forums or networking platform. This led to a difficult attempt to reach smart contract developers and ask them to analyse the solution and provide their feedback by answering the usability questionnaire in which ultimately results in fewer responses. Also, due to the architecture of the solution using docker as service integrated in the solution to provide the information of the security analysis tools, the integration testing was evaluated but extremely hard to performed, which lead to not being performed and present in this dissertation.

## 9.3 Future Work

The future work intends to overcome some of the identified limitations and the need to improve the developed solution. The future work that can be performed is:

- Reduce the EthSential analysis execution time

  There are some possibilities that can be evaluated in this matter. The security analysis tools could be migrated to a single Kubernetes [126] resource and evaluate its deployment in the cloud and thus only requiring the user to have an internet connection. Another possibility is to create a dedicated server so that all security analysis tools would be installed and managed in the server.

- Unify/Standardise the vulnerabilities found

As identified in the limitations, the security analysis tools can provide different outputs to the same vulnerability. In this case is necessary to analyse the responses of each tool and validate if there are no frequent changes in responses. To perform this change, it would be required to change the parsing functionality of each tool.

- Versioning configuration

  To avoid the constant changes of the smart contract versions, it could be possible that the user has to specify the Solidity version or specific tool versions to analyse the contracts before the analysis occurs. This change would benefit the analysis execution time.

- Specification of the tool behaviour

  New tools are being explored [127] [128] that were not analysed in this dissertation. To add these tools in the solution, it would be required to change the code by adding a new tool and create a new version of the framework. To improve this aspect, it could be possible to transfer this control to the user by allowing to specify the tool behaviour through a single json or xml file that would be imported to the framework during the analysis process. E.g. define a json-schema [129] with a data reading structure that the tool would return and perform the respective mapping for the desired result in the framework.

# Bibliography

[1] *solidity-examples/greeter at master · quantanet/solidity-examples · GitHub*. [Online]. Available: https://github.com/quantanet/solidity-examples/tree/master/greeter (visited on 02/04/2020).

[2] *Contracts — Solidity 0.5.3 documentation*. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.3/contracts.html%7B%5C#%7Dfallback-function (visited on 04/18/2020).

[3] A. M. Antonopoulos and G. Wood, *Mastering Ethereum*. 2018, isbn: 9781491971949. [Online]. Available: https://github.com/ethereumbook/ethereumbook/blob/develop/14consensus.asciidoc.

[4] *EthFiddle - Solidity IDE in the Browser. Powered By Loom Network*. [Online]. Available: https://ethfiddle.com/sCLwlzQGAn (visited on 04/18/2020).

[5] *Vyper by Example — Vyper documentation*. [Online]. Available: https://vyper.readthedocs.io/en/latest/vyper-by-example.html (visited on 02/17/2020).

[6] *Bitcoin - Open source P2P money*. [Online]. Available: https://bitcoin.org/en/ (visited on 02/06/2020).

[7] *blockchain meaning - Explorar - Google Trends*. [Online]. Available: https://trends.google.com/trends/explore?q=blockchain%20meaning&date=today%205-y (visited on 06/25/2020).

[8] P. Tasca and C. Tessone, "A taxonomy of blockchain technologies: Principles of identification and classification", *Ledger*, vol. 4, Feb. 2019. doi: 10.5195/ledger.2019.140.

[9] *Which Governments Are Using Blockchain Right Now?* [Online]. Available: https://consensys.net/blog/enterprise-blockchain/which-governments-are-using-blockchain-right-now/ (visited on 06/25/2020).

[10] *Blockchain 50: Billion Dollar Babies*. [Online]. Available: https://www.forbes.com/sites/michaeldelcastillo/2019/04/16/blockchain-50-billion-dollar-babies (visited on 06/25/2020).

[11] *Known Attacks - Ethereum Smart Contract Best Practices*. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/ (visited on 06/26/2020).

[12] *GitHub - ethereum/solidity: Solidity, the Contract-Oriented Programming Language*. [Online]. Available: https://github.com/ethereum/solidity (visited on 02/06/2020).

[13] *GitHub - vyperlang/vyper: Pythonic Smart Contract Language for the EVM*. [Online]. Available: https://github.com/vyperlang/vyper (visited on 02/06/2020).

[14] I. N. Nikolic, A. Kolluri, P. Saxena, and A. Hobor, "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale", Tech. Rep. arXiv: 1802.06038v2. [Online]. Available: https://github.com/MAIAN-tool/.

[15] W. Zou, D. Lo, S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart Contract Development: Challenges and Opportunities", Tech. Rep., 2019. [Online]. Available: http://neo.org/.

[16] M. Coblenz, J. Sunshine, J. Aldrich, and B. Myers, "Smarter Smart Contract Development Tools", Institute of Electrical and Electronics Engineers (IEEE), Sep. 2019, pp. 48–51. doi: 10.1109/wetseb.2019.00013.

[17] A. Dika, "Ethereum Smart Contracts: Security Vulnerabilities and Security Tools", Tech. Rep., 2017.

[18] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security Analysis Methods on Ethereum Smart Contract Vulnerabilities-A Survey", Tech. Rep., p. 2020. arXiv: 1908.08605v2.

[19] *Vyper Preliminary Security Review | ConsenSys Diligence*. [Online]. Available: https://diligence.consensys.net/blog/2019/10/vyper-preliminary-security-review/ (visited on 02/06/2020).

[20] *Watch Your Language: Our First Vyper Audit - Security Boulevard*. [Online]. Available: https://securityboulevard.com/2019/10/watch-your-language-our-first-vyper-audit/ (visited on 02/21/2020).

[21] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System", Tech. Rep., 2008. [Online]. Available: www.bitcoin.org.

[22] *What Different Types of Blockchains are There? - Dragonchain*. [Online]. Available: https://dragonchain.com/blog/differences-between-public-private-blockchains (visited on 07/29/2020).

[23] *What is Blockchain Technology? | IBM Blockchain | IBM*. [Online]. Available: https://www.ibm.com/blockchain/what-is-blockchain (visited on 07/29/2020).

[24] C. Dwork, A. Goldberg, and M. Naor, "On Memory-Bound Functions for Fighting Spam", Tech. Rep., 2003.

[25] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols", Tech. Rep., 1999.

[26] *White Paper · ethereum/wiki Wiki · GitHub*. [Online]. Available: https://github.com/ethereum/wiki/wiki/white-paper (visited on 01/25/2020).

[27] D. Mohanty, *Ethereum for Architects and Developers*. 2018, isbn: 9781484240748. doi: 10.1007/978-1-4842-4075-5.

[28] *Getting Deep Into EVM: How Ethereum Works Backstage - By*. [Online]. Available: https://hackernoon.com/getting-deep-into-evm-how-ethereum-works-backstage-ac7efa1f0015 (visited on 01/26/2020).

[29] *A Deep Dive into the Ethereum Virtual Machine (EVM) - part 1: Introduction*. [Online]. Available: https://www.mayowatudonu.com/blockchain/deep-dive-into-evm-intro (visited on 01/26/2020).

[30] "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER", Tech. Rep., 2014.

[31] *Account Types, Gas, and Transactions — Ethereum Homestead 0.1 documentation*. [Online]. Available: https://ethereum-homestead.readthedocs.io/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html (visited on 01/27/2020).

[32] *EIP 1: EIP Purpose and Guidelines*. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1 (visited on 01/27/2020).

[33] *Visualizing the Most Important Ethereum Forks to Date*. [Online]. Available: https://www.visualcapitalist.com/mapping-major-ethereum-forks/ (visited on 02/11/2020).

[34] *Ethereum Istanbul Hard Fork Explained 2019 | Cointelegraph*. [Online]. Available: https://magazine.cointelegraph.com/ethereum-hard-fork-istanbul-2019/ (visited on 02/11/2020).

[35] *The Roadmap to Serenity aka Ethereum 2.0 Upgrades*. [Online]. Available: `https://consensys.net/blog/blockchain-explained/the-roadmap-to-serenity-2/` (visited on 02/11/2020).

[36] *Smart Contracts*. [Online]. Available: `http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html` (visited on 02/02/2020).

[37] Imran Bashir, *Mastering Blockchain*, 9. 2013, vol. 53, pp. 1689–1699, isbn: 9788578110796. doi: `10.1017/CBO9781107415324.004`. arXiv: `arXiv:1011.1669v3`.

[38] *LLL Introduction — LLL Compiler Documentation 0.1 documentation*. [Online]. Available: `https://lll-docs.readthedocs.io/en/latest/lll_introduction.html` (visited on 02/03/2020).

[39] *An Introduction to Serpent*. [Online]. Available: `https://www.cs.cmu.edu/%7B~%7Dmusic/serpent/doc/serpent.htm` (visited on 02/03/2020).

[40] *Serpent Compiler Audit – OpenZeppelin blog*. [Online]. Available: `https://blog.openzeppelin.com/serpent-compiler-audit-3095d1257929/` (visited on 02/03/2020).

[41] *GitHub - obscuren/mutan: Compiler & Language definition for the Ethereum project*. [Online]. Available: `https://github.com/obscuren/mutan` (visited on 02/18/2020).

[42] L. Stegeman, "Solitor: Runtime Verication of Smart Contracts On the Ethereum network", Tech. Rep., 2018.

[43] F. Schrans, S. Eisenbach, and S. Drossopoulou, "Writing Safe Smart Contracts in Flint", pp. 218–219, 2018.

[44] *Structure of a Contract — Solidity 0.5.3 documentation*. [Online]. Available: `https://solidity.readthedocs.io/en/v0.5.3/structure-of-a-contract.html` (visited on 02/04/2020).

[45] F. S. Lucern, S. Student, S. R. Niya, and T. Bocek, "Design and Implementation of a Smart Contract Application", Tech. Rep., 2017.

[46] *Vyper — Vyper documentation*. [Online]. Available: `https://vyper.readthedocs.io/en/latest/index.html` (visited on 02/11/2020).

[47] D. Cedrim, L. Sousa, R. Gheyi, and A. Garcia, "Does refactoring improve software structural quality? A longitudinal study of 25 projects", in *ACM International Conference Proceeding Series*, Association for Computing Machinery, Sep. 2016, pp. 73–82, isbn: 9781450342018. doi: `10.1145/2973839.2973848`.

[48] J. Chen, X. Xia, D. Lo, J. Grundy, D. X. Luo, and T. Chen, "Domain Specific Code Smells in Smart Contracts", May 2019. arXiv: `1905.01467`. [Online]. Available: `http://arxiv.org/abs/1905.01467`.

[49] by Martin Fowler, K. Beck, J. Brant, W. Opdyke, and don Roberts, "Refactoring: Improving the Design of Existing Code", Tech. Rep.

[50] *Importance of Code Quality and Coding Standard in Software Development - Multidots*. [Online]. Available: `https://www.multidots.com/importance-of-code-quality-and-coding-standard-in-software-development/` (visited on 02/22/2020).

[51] *General Philosophy - Ethereum Smart Contract Best Practices*. [Online]. Available: `https://consensys.github.io/smart-contract-best-practices/general_philosophy/` (visited on 02/22/2020).

[52] *Systematic reviews and meta-analyses: a step-by-step guide | www.ccace.ed.ac.uk*. [Online]. Available: `https://www.ccace.ed.ac.uk/research/software-resources/systematic-reviews-and-meta-analyses` (visited on 02/23/2020).

[53] N. Jahan, S. Naveed, M. Zeshan, and M. A. Tahir, "How to Conduct a Systematic Review: A Narrative Literature Review", *Cureus*, vol. 8, no. 11, Nov. 2016, issn: 2168-8184. doi: 10.7759/cureus.864.

[54] M. J. Grant and A. Booth, "A typology of reviews: an analysis of 14 review types and associated methodologies", *Health Information & Libraries Journal*, vol. 26, no. 2, pp. 91–108, Jun. 2009, issn: 14711834. doi: 10.1111/j.1471-1842.2009.00848.x. [Online]. Available: http://doi.wiley.com/10.1111/j.1471-1842.2009.00848.x.

[55] L. Luu, D. H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter", in *Proceedings of the ACM Conference on Computer and Communications Security*, vol. 24-28-Octo, Association for Computing Machinery, Oct. 2016, pp. 254–269, isbn: 9781450341394. doi: 10.1145/2976749.2978309.

[56] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts", Tech. Rep., 2017. [Online]. Available: https://coinmarketcap.com/currencies/ethereum.

[57] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts", in *Proceedings - International Conference on Software Engineering*, IEEE Computer Society, May 2018, pp. 81–84, isbn: 9781450356626. doi: 10.1145/3183399.3183420.

[58] R. M. Parizi, A. Dehghantanha, A. Singh, and K.-K. R. Choo, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains", in *CASCON '18 Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, Association for Computing Machinery (ACM), 2018, pp. 103–113. [Online]. Available: https://dl.acm.org/citation.cfm?id=3291303.

[59] M. Wohrer and U. Zdun, "Smart contracts: Security patterns in the ethereum ecosystem and solidity", in *2018 IEEE 1st International Workshop on Blockchain Oriented Software Engineering, IWBOSE 2018 - Proceedings*, vol. 2018-Janua, Institute of Electrical and Electronics Engineers Inc., Mar. 2018, pp. 2–8, isbn: 9781538659861. doi: 10.1109/IWBOSE.2018.8327565.

[60] A. Mense and M. Flatscher, "Security Vulnerabilities in Ethereum Smart Contracts", in *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services - iiWAS2018*, New York, New York, USA: ACM Press, 2018, pp. 375–380, isbn: 9781450364799. doi: 10.1145/3282373.3282419. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3282373.3282419.

[61] Y. Liu, Q. Lu, X. Xu, L. Zhu, and H. Yao, "Applying design patterns in smart contracts: A case study on a blockchain-based traceability application", in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10974 LNCS, Springer Verlag, 2018, pp. 92–106, isbn: 9783319944777. doi: 10.1007/978-3-319-94478-4_7.

[62] M. Wöhrer and U. Zdun, "Design Patterns for Smart Contracts in the Ethereum Ecosystem - IEEE Conference Publication", 2018, pp. 1–8.

[63] *Hot Questions - Stack Exchange*. [Online]. Available: https://stackexchange.com/ (visited on 01/30/2020).

[64] Y. Murray and D. A. Anisi, "Survey of Formal Verification Methods for Smart Contracts on Blockchain", Institute of Electrical and Electronics Engineers (IEEE), Jul. 2019, pp. 1–6, isbn: 9781728115429. doi: 10.1109/ntms.2019.8763832.

[65] M. Demir, M. Alalfi, O. Turetken, and A. Ferworn, "Security Smells in Smart Contracts", Institute of Electrical and Electronics Engineers (IEEE), Oct. 2019, pp. 442–449. doi: 10.1109/qrs-c.2019.00086.

[66] M. di Angelo and G. Salzer, "A Survey of Tools for Analyzing Ethereum Smart Contracts", Institute of Electrical and Electronics Engineers (IEEE), Aug. 2019, pp. 69–78. doi: 10.1109/dappcon.2019.00018.

[67] R. Sierra, M. Eilers, and P. Müller, "Verification of Ethereum Smart Contracts Written in Vyper", PhD thesis, 2019.

[68] "Analysis of Ethereum Smart Contracts-A Security Perspective", Tech. Rep., 2019.

[69] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts", Tech. Rep., 2020. arXiv: 1910.10601v2. [Online]. Available: https://smartbugs.github.io.

[70] M. Kaleem, A. Mavridou, and A. Laszka, "Vyper: A Security Comparison with Solidity Based on Common Vulnerabilities", no. June, 2020. arXiv: 2003.07435. [Online]. Available: http://arxiv.org/abs/2003.07435.

[71] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking Smart Contracts with Structural Code Embedding", Tech. Rep., 2020, p. 1. arXiv: 2001.07125v1.

[72] *Overview · Smart Contract Weakness Classification and Test Cases*. [Online]. Available: https://swcregistry.io/ (visited on 06/28/2020).

[73] *ContractGuard - Testing Platform for Smart Contracts*. [Online]. Available: https://contract.guardstrike.com/#/knowledge (visited on 02/23/2020).

[74] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses", Tech. Rep., 2019. arXiv: 1908.04507v1.

[75] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "ReGuard: Finding reentrancy bugs in smart contracts", *Proceedings - International Conference on Software Engineering*, pp. 65–68, 2018, issn: 02705257. doi: 10.1145/3183440.3183495.

[76] *SWEBOK Guide V3 Topics | IEEE Computer Society*. [Online]. Available: https://www.computer.org/education/bodies-of-knowledge/software-engineering/topics (visited on 06/30/2020).

[77] N. Rich and M. Holweg, "VALUE ANALYSIS, VALUE ENGINEERING", Tech. Rep.

[78] P. A. Koen, G. M. Ajamian, S. Boyce, A. Clamen, E. Fisher, S. Fountoulakis, A. Johnson, P. Puri, and R. Seibert, "FuzzyFrontEnd: Effective Methods, Tools, and Techniques LITERATURE REVIEW AND RATIONALE FOR DEVELOPING THE NCD MODEL", Tech. Rep., 2002.

[79] *Home | Ethereum.org*. [Online]. Available: https://ethereum.org/ (visited on 02/08/2020).

[80] *Bitcoin, Ethereum - Explorar - Google Trends*. [Online]. Available: https://trends.google.pt/trends/explore?date=today%205-y&q=%2Fm%2F05p0rrx,%2Fm%2F0108bn2x (visited on 06/27/2020).

[81] *State of the DApps — DApp Statistics*. [Online]. Available: https://www.stateofthedapps.com/stats (visited on 02/08/2020).

[82] *LinkedIn 2018 Emerging Jobs Report*. [Online]. Available: https://economicgraph.linkedin.com/research/linkedin-2018-emerging-jobs-report (visited on 02/13/2020).

[83] *Ethereum Has 4x More Developers Than Any Other Crypto Ecosystem*. [Online]. Available: https://consensys.net/blog/blockchain-development/ethereum-has-4x-more-developers-than-any-other-crypto-ecosystem/ (visited on 02/13/2020).

[84] *GitHub - slockit/DAO at v1.0*. [Online]. Available: https://github.com/slockit/DAO/tree/v1.0 (visited on 02/08/2020).

[85] *anyone can kill your contract · Issue #6995 · openethereum/openethereum*. [Online]. Available: https://github.com/paritytech/parity-ethereum/issues/6995 (visited on 02/09/2020).

[86] *BGP leaks and cryptocurrencies*. [Online]. Available: https://blog.cloudflare.com/bgp-leaks-and-crypto-currencies/ (visited on 02/08/2020).

[87] *Ethereum's Parity Client Loses Sync During Attack | Crypto Briefing*. [Online]. Available: https://cryptobriefing.com/ethereums-parity-client-loses-sync-during-attack/ (visited on 02/09/2020).

[88] Y. Akao, *Quality function deployment: Integrating customer requirements into product design*. Taylor & Francis, 2004, isbn: 9781563273131. [Online]. Available: https://books.google.pt/books?id=NS1Cuw6UQKIC.

[89] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research", *MIS Quarterly: Management Information Systems*, vol. 28, no. 1, pp. 75–105, 2004, issn: 02767783. doi: 10.2307/25148625.

[90] P. Offermann, O. Levina, M. Schönherr, and U. Bub, "Outline of a design science research process", in *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, ser. DESRIST '09, Philadelphia, Pennsylvania: Association for Computing Machinery, 2009, isbn: 9781605584089. doi: 10.1145/1555619.1555629. [Online]. Available: https://doi.org/10.1145/1555619.1555629.

[91] P. Hartel, I. Homoliak, and D. Reijsbergen, "An Empirical Study into the Success of Listed Smart Contracts in Ethereum", Tech. Rep., 2019. arXiv: 1908.11597v1. [Online]. Available: https://etherscan.io/contractsVerified.

[92] A. Ayman, A. Aziz, A. Alipour, and A. Laszka, "Smart Contract Development in Practice: Trends, Issues, and Discussions on Stack Overflow", May 2019. arXiv: 1905.08833. [Online]. Available: http://arxiv.org/abs/1905.08833.

[93] *Amazon.com : learn solidity*. [Online]. Available: https://www.amazon.com/s?k=learn+solidity%7B%5C&%7Dref=nb%7B%5C_%7Dsb%7B%5C_%7Dnoss (visited on 07/03/2020).

[94] *Etherscan - About Us*. [Online]. Available: https://etherscan.io/aboutus (visited on 02/06/2020).

[95] *MasterThesis/etherscan-web-scraping.py at master · 1140251/MasterThesis*. [Online]. Available: https://github.com/1140251/MasterThesis/blob/master/etherscan-web-scraping.py.

[96] *MasterThesis/solidity-parser.py at master · 1140251/MasterThesis*. [Online]. Available: https://github.com/1140251/MasterThesis/blob/master/solidity-parser/solidity-parser.py (visited on 08/23/2020).

[97] *MasterThesis/vyper-parser.py at master · 1140251/MasterThesis*. [Online]. Available: https://github.com/1140251/MasterThesis/blob/master/vyper-parser/vyper-parser.py (visited on 08/23/2020).

[98] *smartbugs/smartbugs: SmartBugs: A Framework to Analyze Solidity Smart Contracts*. [Online]. Available: https://github.com/smartbugs/smartbugs (visited on 07/04/2020).

[99] *MasterThesis/analysis at master · 1140251/MasterThesis*. [Online]. Available: https://github.com/1140251/MasterThesis/tree/master/analysis (visited on 08/23/2020).

[100] *crytic/solc-select: A script to quickly switch between Solidity compiler versions*. [Online]. Available: https://github.com/crytic/solc-select (visited on 07/05/2020).

[101] *Official page for Language Server Protocol*. [Online]. Available: https://microsoft.github.io/language-server-protocol/ (visited on 08/25/2020).

[102] *Extension API | Visual Studio Code Extension API*. [Online]. Available: https://code.visualstudio.com/api (visited on 08/25/2020).

[103] *Docker Hub*. [Online]. Available: https://hub.docker.com/ (visited on 08/25/2020).

[104] *Git*. [Online]. Available: https://git-scm.com/ (visited on 09/20/2020).

[105] *1140251/Ethsential*. [Online]. Available: https://github.com/1140251/Ethsential (visited on 09/19/2020).

[106] *Git Workflow | Atlassian Git Tutorial*. [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows#centralized-workflow (visited on 09/20/2020).

[107] *Git Feature Branch Workflow | Atlassian Git Tutorial*. [Online]. Available: https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow (visited on 09/20/2020).

[108] *Ethsential/ci.yml at master · 1140251/Ethsential*. [Online]. Available: https://github.com/1140251/Ethsential/blob/master/.github/workflows/ci.yml (visited on 09/20/2020).

[109] *ethsential · PyPI*. [Online]. Available: https://pypi.org/project/ethsential/ (visited on 09/20/2020).

[110] *EthSential - Visual Studio Marketplace*. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=1140251.ethsential (visited on 09/20/2020).

[111] *Ethsential/publish.yml at master · 1140251/Ethsential*. [Online]. Available: https://github.com/1140251/Ethsential/blob/master/.github/workflows/publish.yml (visited on 09/20/2020).

[112] *Develop with Docker Engine API | Docker Documentation*. [Online]. Available: https://docs.docker.com/engine/api/ (visited on 08/27/2020).

[113] *Openlawlibrary/pygls: A pythonic generic language server*. [Online]. Available: https://github.com/openlawlibrary/pygls (visited on 08/25/2020).

[114] *Ethsential/publish.yml at master · 1140251/Ethsential*. [Online]. Available: https://github.com/1140251/Ethsential/blob/master/vscode-client/package.json (visited on 09/20/2020).

[115] D. W. W. Royce, "Managing the Development of large Software Systems", Tech. Rep. August, 1970, pp. 1–9.

[116] G. Macdonald, "Goldie MacDonald Centers for Disease Control and Prevention-2013 Criteria for Selection of High-Performing Indicators A Checklist to Inform Monitoring and Evaluation", Tech. Rep.

[117] *How to Set Up a Hypothesis Test: Null versus Alternative*. [Online]. Available: https://www.dummies.com/education/math/statistics/how-to-set-up-a-hypothesis-test-null-versus-alternative/ (visited on 02/22/2020).

[118] *Customer Satisfaction - How to Measure Satisfaction of Customers*. [Online]. Available: https://corporatefinanceinstitute.com/resources/knowledge/other/measuring-customer-satisfaction/ (visited on 10/15/2020).

[119] S. Basak and M. Shazzad Hosain, "Software Testing Process Model from Requirement Analysis to Maintenance", *International Journal of Computer Applications*, vol. 107, no. 11, pp. 14–22, 2014. doi: 10.5120/18795-0147.

[120] J. Nielsen, *Usability engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, isbn: 0125184050.

[121] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)", *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965, issn: 00063444. [Online]. Available: `http://www.jstor.org/stable/2333709`.

[122] STUDENT, "THE PROBABLE ERROR OF A MEAN", *Biometrika*, vol. 6, no. 1, pp. 1–25, Mar. 1908, issn: 0006-3444. doi: `10.1093/biomet/6.1.1`. eprint: `https://academic.oup.com/biomet/article-pdf/6/1/1/605641/6-1-1.pdf`. [Online]. Available: `https://doi.org/10.1093/biomet/6.1.1`.

[123] N. Nachar, "The mann-whitney u: A test for assessing whether two independent samples come from the same distribution", *Tutorials in Quantitative Methods for Psychology*, vol. 4, Mar. 2008. doi: `10.20982/tqmp.04.1.p013`.

[124] F. Wilcoxon, "Individual comparisons by ranking methods", *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945, issn: 00994987. [Online]. Available: `http://www.jstor.org/stable/3001968`.

[125] L. J. Cronbach, "Coefficient alpha and the internal structure of tests", *Psychometrika*, vol. 16, no. 3, pp. 297–334, 1951, issn: 00333123. doi: `10.1007/BF02310555`.

[126] *Kubernetes*. [Online]. Available: `https://kubernetes.io/` (visited on 10/15/2020).

[127] S. Akca, A. Rajan, and C. Peng, "SolAnalyser: A Framework for Analysing and Testing Smart Contracts", *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, vol. 2019-December, pp. 482–489, 2019, issn: 15301362. doi: `10.1109/APSEC48747.2019.00071`.

[128] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "Smartshield: Automatic smart contract protection made easy", in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 23–34.

[129] *JSON Schema | The home of JSON Schema*. [Online]. Available: `http://json-schema.org/` (visited on 10/15/2020).

[130] *openzeppelin-contracts/SafeMath.sol at master · OpenZeppelin/openzeppelin-contracts*. [Online]. Available: `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol` (visited on 07/01/2020).

[131] *openzeppelin-contracts/ReentrancyGuard.sol at master · OpenZeppelin/openzeppelin-contracts*. [Online]. Available: `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol` (visited on 06/29/2020).

[132] *openzeppelin-contracts/ReentrancyGuard.sol at master · OpenZeppelin/openzeppelin-contracts*. [Online]. Available: `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol` (visited on 06/29/2020).

[133] *Other Uses of Commit-Reveal · A Beginner's Guide to Ethereum and Dapp Development*. [Online]. Available: `https://sunnya97.gitbooks.io/a-beginner-s-guide-to-ethereum-and-dapp-developme/writing-smart-contracts/other-uses-of-commit-reveal.html` (visited on 06/30/2020).

[134] *Oracle | solidity-patterns*. [Online]. Available: `https://fravoll.github.io/solidity-patterns/oracle.html` (visited on 06/30/2020).

[135] *randao/randao: RANDAO: A DAO working as RNG of Ethereum*. [Online]. Available: `https://github.com/randao/randao` (visited on 06/30/2020).

[136] *Style Guide — Solidity 0.5.3 documentation*. [Online]. Available: `https://solidity.readthedocs.io/en/v0.5.3/style-guide.html#order-of-functions` (visited on 07/01/2020).

# Appendix A

# Security Vulnerabilities Classification

- SWC 100 - Function Default Visibility

  Problem: Functions without a specific visibility type are public by default. Public functions are opened to any user with access to the contract.

  Recommendation: Specify the function visibility type.

- SWC 101 - Integer Overflow and Underflow

  Problem: An arithmetic operation surpasses the maximum and minimum size for data types.

  Recommendation: Use SafeMath library [130].

- SWC 102 - Outdated Compiler Version

  Problem: Outdated Compiler version can lead to bugs and issues that are fixed in newer versions.

  Recommendation: Use the latest compiler version.

- SWC 103 - Floating Pragma

  Problem: Using the pragma "^" in a compiler version can lead to the usage of outdated code in newer versions.

  Recommendation: Mark a specific compiler version. A contract can have a pragma in specific cases i.e. contracts that belong to libraries or packages used in different contracts versions.

- SWC 104 - Unchecked Call Return Value

  Problem: The address call function returns a tuple with the success of the operation and bytes of the returned data that are unchecked.

  Recommendation: Check the success of the call function.

- SWC 105 - Unprotected Ether Withdrawal

  Problem: A contract with missing function modifiers and incorrect controls of the function can lead to unprotected Ether.

  Recommendation: Add specific controls to allow Ether withdrawal only when required.

- SWC 106 - Unprotected SELFDESTRUCT Instruction

Problem: A SELFDESTRUCT function without proper restrictions can delete the contract from the network.

Recommendation: Remove SELFDESTRUCT functions or allow only specific user to execute the operation.

- SWC 107 - Reentrancy

  Problem: When executing external contracts, a contract can call back into the calling contract before the first invocation of the function is finished.

  Recommendation: Execute internal operations and changes of state before calling the external contract or specific use OpenZeppelin ReentrancyGuard contract [131].

- SWC 108 - State Variable Default Visibility

  Problem: The default visibility for variables is internal. If the state is not specified can have mistaken expectations.

  Recommendation: Always specify variables visibility (public, internal or private).

- SWC 109 - Uninitialised Storage Pointer

  Problem: Local storage variables point to storage locations in the contract. By inappropriately initialise variables can produce unexpected vulnerabilities.

  Recommendation: Specifically mark local variables with the *memory* attribute. Storage variables must be marke with the attribute *storage*.

- SWC 110 - Assert Violation

  Problem: Using falsy assert statements means either the code contains a bug or, is validated improperly.

  Recommendation: Check if the assert is invariant otherwise use the *required* statement.

- SWC 111 - Use of Deprecated Solidity Functions

  Problem: Functions and operators can be changed Solidity and lead to deprecated usage.

  Recommendation: Check and apply replacement functions provided by Solidity [132].

- SWC 112 - Delegatecall to Untrusted Callee

  Problem: The usage of function *delegatecall* imports the target address code to be executed in the context of the current contract allowing the user to to change the targeted address with a different code.

  Recommendation: *delegatecall* functions should be avoided otherwise check if targeted address is desired and trusted.

- SWC 113 - DoS with Failed Call

  Problem: A DoS can occurr when an external call fails.

  Recommendation: Check the result of external calls and avoid using multiple calls in a single transaction and loop conditions.

- SWC 114 - Transaction Order Dependence

  Problem: Transactions in Ethereum are executed based on the amount of gas spent in the transaction. Higher amount transaction will execute first. A race condition can occur when a user submits an transaction with higher gas spent then the previous transactions. The contract will be executed first and other transactions will not performed as expected.

  Recommendation: Use a commit reveal scheme [133] to send the data to the user.

- SWC 115 - Authorisation through *tx.origin*

  Problem: The *tx.origin* represents the sender of the transaction. Using this address for authorisation makes the authorisation always valid.

  Recommendation: Use the *msg.sender* variable.

- SWC 116 - Block values as a proxy for time

  Problem: the variables *block.timestamp*, and *block.number* are imprecise and if used as time values can give incorrect values.

  Recommendation: Avoid using timestamp in constant functions. Use oracles to retrieve data outside the network [134].

- SWC 117 - Signature Malleability

  Problem: cryptographic signatures used in part of the signed message hash can be altered.

  Recommendation: A signature should not be used into a signed message hash.

- SWC 118 - Incorrect Constructor Name

  Problem: Solidity versions prior to 0.4.22 used a constructor as the same name of the contract. If the function is not equal to the contract name is a normal function.

  Recommendation: Upgrade contract to an higher version.

- SWC 119 - Shadowing State Variables

  Problem: Ambiguous state variables with the same name either used in a single contract or inherited from other contracts can provide a faulty result.

  Recommendation: Remove Ambiguous state variables with the same name.

- SWC 120 - Weak Sources of Randomness from Chain Attributes

  Problem: Allied with the SWC-116 using timestamp variables to generate random numbers or other variables *blockhash*, *block.difficulty* can be manipulated by miners.

  Recommendation: Use a commit scheme e.g., *RANDAO* [135] or Oracle to generate numbers.

- SWC 121 - Missing Protection against Signature Replay Attacks

  Problem: The *ecrecover* function only validates the integrity of the message hash, does not validate the signer address or if the signer message is unique. The usage of this function without proper subsequent validations can lead to Signature Replay Attacks.

Recommendation: Use *ECDSA* library to verify the sign message.

- SWC 123 - Requirement Violation

  Problem: Solidity *require()* validates external inputs of a function and an weakness can occur if an improper validation of the returned value violates the requirements.

  Recommendation: Simplify the require logical condition

- SWC 124 - Write to Arbitrary Storage Location

  Problem: Contracts write data in storage or memory. If an attacker can write to an arbitrary storage location authorisations will pass.

  Recommendation: Make assurance that the data structures share the same storage and cant be overwritten.

- SWC 125 - Incorrect Inheritance Order

  Problem: Ambiguous inherit functions are set based on a priority between the base contracts and executes them by order of priority.

  Recommendation: Specify inheritance in the correct order of execution.

- SWC 126 - Insufficient Gas Griefing

  Problem: Contracts that accept data from users and use it in a sub-call on another contract, without proper validations, an attacker can provide less gas then gas needed to execute the sub-call and end the transaction.

  Recommendation: Only allow verified users to execute the sub-call. Require that the user provides gas to execute the whole transaction successfully.

- SWC 127 - Arbitrary Jump with Function Type Variable

  Problem: Uses of assembly instructions, such as *mstore* or assign operator can point a function type variable to any code instruction.

  Recommendation: Avoid using assembly functions. Avoid assigning arbitrary values to function types.

- SWC 128 - DoS With Block Gas Limit

  Problem: When dealing with high and unknown data size structures and the cost of executing the transaction exceeds the block gas limit an Dos can occur.

  Recommendation: Avoid iterations over high and unknown data sized structures.

- SWC 129 - Typographical Error

  Problem: Occur when an unintended defined operation is used.

  Recommendation: Most of the typographical error are fixed in version 0.5.0 and higher. Use the latest compiler version.

- SWC 130 - Right-To-Left-Override control character (U+202E)

  Problem: Use of Right-To-Left-Override unicode character can lead to misunderstand the behaviour of the contract.

  Recommendation: Never use Right-To-Left-Override unicode characters.

- SWC 131 - Presence of unused variables

  Problem: Unused variables can require more gas and indicate a bad code structure.

  Recommendation: Remove unused variables.

- SWC 132 - Unexpected Ether balance

  Problem: Assuming a specific Ether balance can lead to DoS conditions.

  Recommendation: Avoid using specific Ether balance.

- SWC 133 - Hash Collisions With Multiple Variable Length Arguments.

  Problem: Solidity function *abi.encodePacked()* used with multiple variable length parameters can lead to a hash collision.

  Recommendation: Don't allow access to parameters used in *abi.encodePacked()* function or add a fixed size variable as a parameter. In alternative use the function *abi.encode()*.

- SWC 134 - Message call with hardcoded gas amount

  Problem: *transfer()* and *send()* functions used with specific gas amounts can break a contract.

  Recommendation: Avoid using *transfer()* and *send()* functions. Use *call.value().gas()* function and use OpenZeppelin ReentrancyGuard contract [131] or checks-effects-interactions pattern identified in the Design Patterns Taxonomy. In Vyper use *raw_call()* function or/and a *@nonreentrant(<key>)* decorator in the function.

- SWC 135 - Code With No Effects

  Problem: Functions with no behaviour or effects can lead to require more gas then expected.

  Recommendation: Every function should produce an effect. Check the usage of the function and test the contract behaviour.

- SWC 136 - Unencrypted Private Data On-Chain

  Problem: Private type variables can be read from users.

  Recommendation: Protect private type variables with encryption or use Oracles to store the variable outside the network.

- Array length manipulation

  Problem: Changing the array length directly can lead to a storage overlap attack.

  Recommendation: Avoid to change the length of the dynamic array directly.

- Complex Fallback

  Problem: Using a *send()* function with a complex fallback function that can fail will cause an disrupt in the send behaviour.

  Recommendation: Avoid using *send()* function. Use *call.value().gas()* function and reduce the code complexity of fallback functions. In Vyper use *raw_call()* function.

- Exception Disorder

  Problem: Using the send function will not throw an exception when the operation fails.

  Recommendation: Use *call.value().gas()* function. In Vyper use *raw_call()* function.

- Freezing ether

  Problem: Receiving ether from other account but don't send to other account.

  Recommendation: Provide a function to send ether out.

- Function order

  Problem: Writing a contract with disordered function can make the contracts hard to understand and read.

  Recommendation: Write function following Solidity style guide function order [136].

- Gassless send

  Problem: The *send()* function transfers ether to a contract, and can spend all the gas available.

  Recommendation: Use *call.value().gas()* function. In Vyper use *raw_call()* function.

- Mark callable contracts

  Problem: External contracts without the trusted naming of variables or methods can indicate that it may be unsafe to use them.

  Recommendation: marked external contracts trusted or untrusted.

- Payable fallback

  Problem: Fallback functions marked without the payable keyword cannot receive ether.

  Recommendation: Mark callback functions with payable keyword.

- Reason string

  Problem: An *require()* or *revert()* functions can show an error information if the assertion fails, but it is necessary to add an reason string.

  Recommendation: Add reason strings to *require()* or *revert()* functions with at most 500 characters long.

- Unchecked Division

  Problem: Divisions with int type variables unchecked can fail the execution of the contract

  Recommendation: Use SafeMath library [130].

- Quotes

  Problem: Use of double quote strings.

  Recommendation: Use single quote strings.

- Uninitialised State

  Problem: Uninitialised State variables will throw exceptions.

Recommendation: Initialised State variables by assigning values ether by user/transaction input or direct declaration.

- Visibility Modifier Order

  Problem: Modifier functions in order to execute need the visibility order correctly.

  Recommendation: Indicate the visibility of the function before modifier.

# Appendix B

# System Testing

Table B.1: System Test cases

| ID | Description | Steps | Expected Result | Passed/Failed |
|---|---|---|---|---|
| 1 | Install tools | 1. Insert command - ethsent install | Docker images created | Passed |
| 2 | Install tools - Docker Unavailable | 1. Stop Docker 2. Insert command - ethsent install | Show - Docker not found | Passed |
| 3 | Analyse mythril | 1. Insert command - ethsent analyse -t mythril -f example-solidity.sol | Create file with result | Passed |
| 4 | Analyse smart contract with invalid file | 1. Insert command - ethsent analyse -t mythril -f notfound.sol | Show - No such file or directory | Passed |
| 5 | Analyse smart contract with invalid tool | 1. Insert command - ethsent analyse -t notFound -f example-solidity.sol | Show - Invalid choice: 'notFound' | Passed |
| 6 | Analyse smart contract with image not installed | 1. Insert command - ethsent analyse -t mythril -f example-solidity.sol | Install tool and create file with result | Passed |
| 7 | Analyse smart contract with multiple tools | 1. Insert command - ethsent analyse -t all mythril -f example-solidity.sol | Create one file with results for each tool | Passed |

| | | | | |
|---|---|---|---|---|
| 8 | Analyse smart contract with all tools and selected output path | 1. Insert command - ethsent analyse -t all -f example-solidity.sol -op results/all/ | Create one file with results for each tool | Passed |
| 9 | Analyse all tools to selected invalid output path | 1. Insert command - ethsent analyse -t all -f example-solidity.sol -op results/invalid/ | Create directory with one file with results for each tool | Passed |
| 10 | Install tools from VS Code | 1. Open VS Code<br>2. Open example-solidity.sol<br>3. Run command - EthSential: Install Security Analysis Tools | Show message - Installation completed | Passed |
| 11 | Install tools from VS Code - Docker Unavailable | 1. Stop Docker<br>2. Open VS Code<br>3. Open example-solidity.sol<br>4. Run command - EthSential: Install Security Analysis Tools | Show message - Docker is not available | Passed |
| 12 | Analyse smart contract from VS Code | 1. Open VS Code<br>2. Open example-solidity.sol<br>3. Run command - EthSential: Analyse File | Show message - Analysis finished | Passed |
| 13 | Analyse smart contract from VS Code with Docker Unavailable | 1. Stop Docker<br>2. Open VS Code<br>3. Open example-solidity.sol<br>4. Run command - EthSential: Analyse File | Show message - Docker is not available | Passed |

# Appendix C

# Smart Contract Execution Time Test Results

| | Slither | | Securify | | Mythril | | All |
|---|---|---|---|---|---|---|---|
| **Vulnerabilities** | **Before** | **After** | **Before** | **After** | **Before** | **After** | **After** |
| SWC-101 | 1 | 4 | 1 | 4 | 12 | 14 | 17 |
| SWC-102 | 1 | 4 | 1 | 4 | 7 | 9 | 11 |
| SWC-103 | 1 | 4 | 1 | 4 | 8 | 11 | 12 |
| SWC-105 | 1 | 4 | 2 | 4 | 139 | 142 | 156 |
| SWC-106 | 1 | 4 | 1 | 4 | 11 | 14 | 12 |
| SWC-107 | 1 | 4 | 1 | 4 | 193 | 195 | 214 |
| SWC-108 | 1 | 4 | 2 | 4 | 28 | 32 | 35 |
| SWC-110 | 1 | 4 | 1 | 4 | 8 | 10 | 13 |
| SWC-111 | 1 | 4 | 1 | 4 | 9 | 11 | 15 |
| SWC-112 | 1 | 4 | 1 | 4 | 34 | 36 | 35 |
| SWC-113 | 1 | 4 | 2 | 4 | 16 | 20 | 23 |
| SWC-114 | 1 | 4 | 1 | 4 | 23 | 27 | 27 |
| SWC-115 | 1 | 4 | 1 | 4 | 13 | 16 | 18 |
| SWC-116 | 1 | 4 | 2 | 4 | 158 | 160 | 164 |
| SWC-117 | 1 | 4 | 1 | 4 | 90 | 94 | 106 |
| SWC-120 | 1 | 4 | 1 | 4 | 11 | 18 | 20 |
| SWC-123 | 1 | 4 | 1 | 4 | 21 | 24 | 26 |
| SWC-124 | 1 | 4 | 1 | 4 | 26 | 32 | 29 |
| SWC-125 | 2 | 6 | 1 | 6 | 1323 | 1356 | 1525 |
| SWC-126 | 1 | 4 | 1 | 4 | 223 | 225 | 240 |
| SWC-127 | 1 | 4 | 1 | 4 | 12 | 14 | 24 |
| SWC-128 | 1 | 4 | 2 | 4 | 23 | 26 | 24 |
| SWC-130 | 1 | 4 | 2 | 4 | 26 | 28 | 36 |
| SWC-131 | 1 | 4 | 1 | 4 | 9 | 11 | 22 |
| SWC-132 | 1 | 4 | 1 | 4 | 8 | 10 | 21 |
| SWC-133 | 1 | 4 | 2 | 5 | 37 | 40 | 42 |
| SWC-134 | 1 | 4 | 1 | 4 | 45 | 48 | 64 |
| SWC-135 | 1 | 4 | 2 | 4 | 20 | 22 | 37 |
| SWC-136 | 1 | 4 | 2 | 5 | 17 | 19 | 35 |
| Unchecked Division | 1 | 4 | 1 | 4 | 6 | 8 | 9 |
| Unitialized State | 1 | 4 | 2 | 5 | 21 | 23 | 34 |
| Visibility Modifier Order | 1 | 4 | 1 | 4 | 11 | 13 | 24 |
| Complex Fallback | 1 | 4 | 1 | 4 | 37 | 42 | 30 |
| Freezing ether | 1 | 4 | 2 | 4 | 56 | 57 | 62 |
| Function order | 1 | 4 | 1 | 3 | 12 | 16 | 12 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Gasslend send | 1 | 4 | 1 | 4 | 20 | 21 | 23 |
| Mark callable contracts | 1 | 4 | 1 | 4 | 20 | 23 | 14 |
| Payable fallback | 1 | 4 | 1 | 4 | 8 | 10 | 13 |
| Reason string | 1 | 4 | 1 | 4 | 12 | 15 | 16 |
| Single quotes | 1 | 4 | 1 | 4 | 9 | 12 | 14 |

# Appendix D

# Usability Questionnaire

## Usability questionnaire for the Ethsential framework

This usability questionnaire is part of my dissertation project for the Master of Software Engineering at ISEP (Instituto Superior de Engenharia do Porto).
This questionnaire explores the use of an Ethereum smart contract analysis framework, called EthSential.
EthSential is a security analysis framework for Ethereum smart contracts. It bundles security analysis tools to find vulnerabilities in smart contracts code.

Participation in this questionnaire is completely voluntary and confidential and it is expected to take less than 10 minutes of your time. You might withdraw at any time.

The questionnaire requires the participant to install the published EthSential python framework and respective VS Code extension to perform some operations of the guidelines provided in the next page.

Before answering this questionnaire, please follow the requirements:
1. Install Python >=3.6.
2. Install Ethsential using - pip install ethsential - or - pip3 install ethsential
3. Run in the command line: ethsential -h.
4. Check the result displays the ethsential options.
4. Run in command line - ethsential install.
5. Install EthSential VS Code extension from https://marketplace.visualstudio.com/items?itemName=1140251.ethsential (https://marketplace.visualstudio.com/items?itemName=1140251.ethsential) in VS Code.

* Obrigatório

1. Have you ever heard of Smart Contracts? *

◯ Yes

◯ No

2. Have you previously developed smart contracts? *

◯ Yes

◯ No

3. Do you find the security of smart contracts an important issue? *

◯ Yes

◯ No

4. Which of the following tools do you know?

☐ Mythril

☐ Securify

☐ Slither

☐ Solhint

☐ MythX

☐ Oyente

☐ SmartCheck

☐ Remix

5. Have you ever used security analysis tools to evaluate smart contracts? *

◯ Yes

◯ No

6. Which one(s)? *

## EthSential Guidelines

This section requires the user to have the EthSential application and its extension on the VS Code ready to use.
The application only supports Solidity versions from 0.4.1 to 0.6.4 inclusive.
The proposed guidelines to follow are:

1. Open any file with .sol extension in VS Code. There are some examples available in https://github.com/1140251/Ethsential/tree/master/examples/exploits (https://github.com/1140251/Ethsential/tree/master/examples/exploits).
2. With the file opened in VS Code, open the command palette by pressing (Ctrl+Shift+P or Cmd+Shift+P on Mac) and type EthSential.
3. Run the command from the command palette - EthSential: Analyse File.
4. Wait for the results to show in the file.
5. Open the user settings by pressing (Ctrl+, or Cmd+, on Mac) and enter EthSential.
6. Disable Mythril and Securify.
7. Run the steps 2 to 4 one more time.

The tool can also be executed without the VS Code but is not required for this questionnaire.

7. What was your overall impression of the framework? *

☆ ☆ ☆ ☆ ☆

8. Do you consider that the solution streamlines and automates the process of developing smart contracts? *

☆ ☆ ☆ ☆ ☆

9. Do you consider that the integration of multiple analysis tools is important for the process of developing smart contracts? *

☆ ☆ ☆ ☆ ☆

10. Do you consider that the solution is fast in the analysis process? *

☆ ☆ ☆ ☆ ☆

11. Do you consider the process of performing an analysis of simple access? *

☆ ☆ ☆ ☆ ☆

12. Do you consider that the system correctly recognizes the vulnerabilities of the smart contract? *

☆ ☆ ☆ ☆ ☆

13. Do you consider the information of the vulnerability useful to rectify the smart contract? *

☆ ☆ ☆ ☆ ☆

14. What was the best/worst feature of the framework?

15. Would you add/remove tool(s) in the framework? *

○ Yes

○ No

16. Which one(s)?

| |
|---|
| |

17. How likely are you to use this framework in the future? *

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|

Not likely at all — Extremely likely

18. How can we improve the framework?

| |
|---|
| |