

Ryan Weege Achjian

Building a Database for Evaluating Smart Contract Vulnerabilities Detection Tools

São Paulo, SP

2023

Ryan Weege Achjian

Building a Database for Evaluating Smart Contract Vulnerabilities Detection Tools

Trabalho de conclusão de curso apresentado ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro.

Universidade de São Paulo – USP

Escola Politécnica

Departamento de Engenharia de Computação e Sistemas Digitais (PCS)

Supervisor: Prof. Dr. Marcos Antônio Simplicio Junior

São Paulo, SP

2023

Gerar a ficha catalográfica em <https://www.poli.usp.br/bibliotecas/servicos/catalogacao-na-publicacao>
Salvar o pdf e incluir na monografia

Acknowledgements

I would like to thank Professor Marcus Antônio Simplício Junior, for the guidance and patience along the development of this work, as well as all the team of backdoors detection and blockchain development of the Laboratório de Redes de Computadores (LARC) of the Universidade de São Paulo, for their support, constructive comments and recommendations of recent works, trends and related topics.

Abstract

With the emergence of cryptocurrencies, transacting in a blockchain environment, such as the Ethereum network, has become a common practice. As the volume of transactions grows, new threats in the so-called smart contracts emerge continuously. Seeking to avoid issues in the transactions, several detection tools have already been developed or are in the process thereof. The intent of this work is to address the detection tool validation process. To the best knowledge of the authors, so far there is no unified test database for detection tools validation. The final result of this study will be the implementation of a database that is as representative of the real world vulnerabilities present in the Ethereum mainnet as possible.

Keywords: Blockchain, Security, Vulnerability, Solidity, Ethereum, Smart Contract

Resumo

Com o aparecimento das cripto-moedas, transações em ambientes de *blockchain*, como a rede Ethereum, se tornaram uma prática comum. Conforme o volume de transações cresce, novas ameaças nos chamados contratos inteligentes emergem continuamente. Procurando evitar problemas nas transações, uma série de ferramentas de detecção foram desenvolvidas ao longo do tempo. Do conhecimento do autor, ainda não existe uma base para validação de ferramentas de detecção. O resultado final deste estudo visa implementar uma base de dados que seja o mais representativa das vulnerabilidades presentes no mundo real na rede Ethereum possível.

Palavras-chave: Blockchain, Segurança, Vulnerabilidade, Solidity, Ethereum, Contrato Inteligente

List of Figures

Figure 1 – Results obtained in the [6] survey	18
Figure 2 – Results obtained in the [26] survey	19
Figure 3 – Selection of SC Applications to be Implemented	22
Figure 4 – Inclusion Framework for external Data Sets SCs	23
Figure 5 – Workflow	24
Figure 6 – The three types of vulnerabilities, as specified in [12]	30
Figure 7 – Example of call to the unknown SC	31
Figure 8 – Example of gas costly pattern SC	32
Figure 9 – Example of gasless send SC	33
Figure 10 – Example of Hash Collision Sign Verification SC	34
Figure 11 – Example of Hash Collision Signing SC	35
Figure 12 – Example of Mishandled Exception SC	35
Figure 13 – Example of Overflow SC	36
Figure 14 – Example of Reentrancy SC	37
Figure 15 – Example of Self-destruct Unprotected Call SC	39
Figure 16 – Example of Unprepared for Self-destruct SC	40
Figure 17 – Example tx.origin SC	41
Figure 18 – Example tx.origin attack SC	41
Figure 19 – Example Weak Rand SC	42
Figure 20 – SCs Implementation Workflow	46
Figure 21 – JSON file configuration	49
Figure 22 – Bash Code that Implements the Automation Module	50
Figure 23 – Example of Automatic Usage of the Database Using SmartCheck	51
Figure 24 – Blockchain Hype in 2019	55
Figure 25 – Proposed Workflow for Vulnerability Detection Tool Validation	57
Figure 26 – Example of the RemixIDE developer interface	62
Figure 27 – RemixIDE Ethereum Execution Environments	63
Figure 28 – RemixIDE Supported Compiler Versions (Incomplete) Sample	64
Figure 29 – Example the Deployment of a SC in the Shanghai Test Environment	65
Figure 30 – RemixIDE Connected to a Locally Generated Ganache Testnet	65
Figure 31 – Example the Deployment of a SC in the Local Ganache Testnet	66
Figure 32 – Example of Locally Created Testnet Accounts	67
Figure 33 – Example of Blocks Generated by Interacting with the Ganache Testnet	68
Figure 34 – Example of Block Content in the Ganache Testnet	68
Figure 35 – Example of Transaction Content in the Ganache Testnet	69

List of Tables

Table 1 – Number of each Vulnerability in The Database by Source (Part 1)	. . .	47
Table 2 – Number of each Vulnerability in The Database by Source (Part 1)	. . .	47
Table 3 – Number of each Vulnerability in The Database by Source (Part 1)	. . .	48
Table 4 – Number of each Vulnerability in The Database by Source (Part 1)	. . .	48

List of abbreviations and acronyms

SC Smart Contract

Contents

1	INTRODUCTION	11
1.1	Objectives	11
1.2	Justification	12
1.3	Work Organization	13
1.3.1	Introduction	13
1.3.2	Conceptual Aspects	13
1.3.3	Methodology	13
1.3.4	Requisites Specification	14
1.3.5	Development	14
1.3.6	Final considerations	14
2	CONCEPTUAL ASPECTS	15
2.1	State of the Art	16
3	METHOD	21
4	REQUISITES SPECIFICATIONS	24
4.1	Database Characteristics Specifications	24
4.2	Developed Smart Contract Specifications	25
4.3	Externally Sourced Smart Contracts Specifications	26
5	DEVELOPMENT	27
5.1	Technologies	27
5.2	Database Construction	29
5.2.1	Lead Example	43
5.2.2	Variations Implementations	43
5.2.3	IA Assisted Implementation	45
5.2.4	Addition of Externally Implemented Smart Contracts	46
5.2.5	Database Smart Contracts Labeling	47
5.3	Automation Module Implementation	49
5.4	Tests and Evaluation	51
6	FINAL CONSIDERATIONS	54
6.1	Conclusions	54
6.2	Contributions	56
6.3	Future Works	56

Bibliography	58
ANNEX	61
ANNEX A – REMIXIDE INTERFACE AND CAPABILITIES	62
ANNEX B – GANACHE INTERFACE AND CAPABILITIES	67

1 Introduction

Starting in 2008 with Bitcoin, digital currencies have become a reality for many businesses and common people around the world. With the decentralized network provided by blockchain technologies anyone is empowered to transact in this digital world with ease. One way to do that is through *smart contracts* [14]. Those are simply pieces of code, like any other software, that seek to implement more complex financial functionalities besides simple money transfer without the need of a trusted intermediary. There are many programming languages that can be used to implement smart contracts, but the most common language used in this environment is Solidity, which is an object oriented programming language created by the Ethereum Foundation for their network.

With a growing number of everyday users, the safety of the smart contracts who power these financial transactions has been a growing concern for users, companies and the scientific community focused in information security. As the number of daily transactions through smart contracts, as well as the volume of funds on them increases, malicious pieces of code or bugs introduced during their development could mean a huge loss for the parties involved. This has already happened before as can be seen by the examples presented in the justification section.

1.1 Objectives

The main objective of this project is to provide the means to perform the validation of any future vulnerability detection tool. The focus in terms of validation is the implementation of a database comprised of several examples of SC vulnerabilities together with highly accurate labels for each vulnerability present in the SCs. The examples should be short but representative of real life usages of SCs. That should enable the database to measure a vulnerability detection tool detection capability in a simple environment, that is, in examples without many dependencies and complex logic that often have low quality labels. Such other examples of databases have already been presented in the scientific literature.

A second objective for this project it will be to propose a validation method that is more reliable than the methods in use today. In the context of this work and the problems presented with detection tools validation "more reliable" means a validation framework that can assert with more certainty the capabilities of the tool, that does not depend of the given time the validation takes place (unlike the validations who extract a smart contract database from the Ethereum mainnet, as was explained previously) and, therefore, is able to compare the performance of the different vulnerability detection tools.

1.2 Justification

As previously presented, as the volume of transactions that take place in various blockchain networks grow, notoriously Bitcoin and Ethereum networks, it is necessary to be aware of the vulnerabilities that can be present in the smart contracts that make this transactions possible. Focusing on the Ethereum network, numerous attacks involving solidity code vulnerabilities have already taken place and resulted in voluminous losses in funds for the parties involved. Some notorious examples are the following:

In 2016 a reentrancy attack in a crowdfunding SC that received the nickname of "The DAO" stole 3.6 million ethers, which, at the time, amounted to about US\$50 million. This was one of the first large proportions attack in Ethereum and the first to use the reentrancy technique. In 2022 hackers used a bridge (which is a way to connect different blockchain networks together and make transfers of funds between them possible) vulnerability to steal a sum that amounted more than US\$320 million. This attack has since received the name of Wormhole Attack. Recently, SC vulnerabilities that were not noticed during the development of SC already deployed in the Ethereum mainnet have been exploited as ransoms. The attackers identify a vulnerability in a complex SC powered DApp and demand the owners a ransom in order to disclosure the vulnerability, under the treat of selling the vulnerability knowledge in the dark web or exploited the vulnerability.

With the examples presented and other attacks that happened in the ethereum network over the years, the community have growing concerns with the vulnerabilities that may be present in the smart contracts operating in the network and being developed. Following this trend, many studies have already presented different kinds of detection tools with the aim of finding possible vulnerabilities in a solidity code. Many of this initiatives have supposedly achieved positive results and enabled smart contracts to be analyzed during and after their development phase. The possibility of utilizing these tools have improved the safety of transacting in the Ethereum network.

The development of vulnerabilities detection tools have been gaining steam since 2019. However, it have been noticed the absence of a robust and verifiable framework for validating the detection tools that have been as much as those that are being developed. Most of the validation of the detection tools have been based in using big unlabeled databases extracted from the Ethereum mainnet itself based on an arbitrary start date (for example, all the smart contracts deployed after January 1st of 2015). This database is then scanned for vulnerabilities with the newly developed detection tool and the smart contracts that are pointed as vulnerable are analyzed by undisclosed specialists who determine if the supposed vulnerability pointed by the tool is, in fact, a vulnerability.

In the present situation it is extremely difficult to assert that the state of the art vulnerability detection tools have a satisfactory detection capability. Besides that,

without an accurate validation framework it is almost impossible to determine if the newly developed tools are in fact contributing with the improvement of the state of the art capabilities. This statement is also corroborated in the scientific literature, as is presented in chapter 2.

1.3 Work Organization

This work is organized in the following chapters:

1.3.1 Introduction

The 1 chapter presents an introduction to the detailed development of the henceforth work. It is comprised of a thorough contextualization of this work and where in the scientific development it fits as well as a justification of why the development and the results presented are important to the development of the current state of the art and to the scientific and developer communities.

1.3.2 Conceptual Aspects

The 2 chapter, the theoretical aspects that underlined all the development of this work will be presented. Those aspects include technical definitions and conventions that are of utmost importance to understand the development that will be presented. Besides that, this chapter presented the current state of the art regarding the current vulnerability detection tools validation capabilities. This is achieved by the presentation of a comprehensive review of the scientific literature that was undertaken prior and during the development of this work.

1.3.3 Methodology

The 3 chapter presets the methods that were utilized during the development of this work in order to obtain a logic conclusions. Besides that, the methodology also explains how the author conducted the work in terms of the choices that presented themselves during the development of this work in order to achieve coherent results. Lastly, the method presented makes it possible to anyone with the basic knowledge of programming language, blockchain and SCs to be able to reproduce the results of this work. It is important to notice, nevertheless, that the reproduction of this work would hardly result in an identical database as the one presented in this work. However, every database that were to be constructed following the methodology and steps presented would ensure that this other database has the same characteristics as the one presented in this work, as well as permitting the developer to reach the same conclusions.

1.3.4 Requisites Specification

The 4 chapter presents the the guidelines that oriented the development of this work. Those guidelines include the aspects that were deemed indispensable to the developed database in order to consider it a worthy contribution to the scientific understanding of the are were this work is situated. That is, the criteria considered to consider this works development success. This specifications include quantitative and qualitative parameters for the developed database as well as the automation module.

1.3.5 Development

The 5 chapter begins by providing an overview of the technologies that were selected for the development of this work. Those technologies are explained and their selection is justified. Besides that, this chapter aims to provide a detailed explanation of all the steps that were taken during the development of this work. This explanation includes a thorough description of each of those steps as well as a justification of why it was taken. Furthermore, all the projects decisions are explained in this chapter.

Chapter 5 also presents an overview of the characteristics of the constructed database, such as directories, file contents, vulnerabilities examples and explanations and output examples. In addition to that, the interaction with the database is explained and exemplified. This ensures that the reader can make sense of what have been developed and can readily use the database to its full potential, as well as modify it as he or she deems relevant.

1.3.6 Final considerations

The 6 aims to summarize which goals was achieved under the development of this work as well as formalize and specify the proposal of which the constructed database takes part. Besides that, the main challenges encountered by the author during the development are also presented and explained. Lastly, the scientific contributions achieved by this work are highlighted and a list of future works that are made possible after the development of this work is presented.

2 Conceptual Aspects

To the development of this work a great number of technical concepts needed to be taken into account.

One of them is the definition of review coverage, that is, how to determine that the literature review undertook was enough to cover the most important vulnerabilities already encountered and compiled.

To deal with that situation, it was considered that the reviews already presented in the literature were representative, or at least partially representative, of the current state of the academic knowledge of the smart contract vulnerabilities. Therefore, the review was first comprised of different reviews that were analyzed and compiled into a single list of known vulnerabilities. After that, other recent attacks that were still not contemplated in these works were also added to the list.

The other important challenge in terms of definitions is related to vulnerability variation, that is, the different ways to implement the same vulnerability and how to assert that most of them were taken into account during the development. This definition is a real challenge in terms of every computational scientific work that requires a great number of code variations. That is because there is no concrete definition of variation in this scope. Therefore, the project followed the following metrics:

First, the possible implementation variations were separated into two groups: those who interfere directly with the *modus operandi* of the vulnerabilities and those who don't. After that, other vulnerabilities databases in other scopes were analyzed in order to assert how many examples each of them presented. This analysis resulted in the number of 10 different implementations of each vulnerability. Lastly, every group of 10 implementations were separated in a way that resulted in 6 implementations were of the first group (present some kind of *modus operandi* interference) and 4 were of the second group (do not present any kind of *modus operandi* interference).

With a vulnerability *corpus* implementation concepts well defined, the next important aspect to define is the selection of detection tools for test. In order to do that, the main point considered was the detection tool references among the literature and in other development medias, such as citations of the tool in github and other developer communities. This approach considers that, the more a tool is referenced, the more impact it's development had and still have in the community. This is a strong indicator that the tool is widely utilized. Therefore, the characterization of these tools in terms of its performance as defined in this work will be more relevant.

With the definitions presented it is possible to search in the scientific literature for relevant works. This step is of extreme importance and its results are presented below:

2.1 State of the Art

The Ethereum block-chain network is one of the most widely used to this date. The safety of the smart contracts who power the transactions in this network have prompted many groups across the world do work in developing tools to identify possible vulnerabilities in the smart contracts codes. This concerns can be verified by the numerous review works about smart contract vulnerabilities, noticeably in the Ethereum network, and vulnerability detection tools that have been published since 2019 [12] [11]. At first the validation of the first generations of detection tools was done in an *ad-hoc* manner.

The authors of [25], [5], [15], [21] and [20] validated their respective vulnerability detection tools by extracting a large amount of smart contracts directly from the Ethereum mainnet. The database was then inspected by the newly developed detection tool and the results were then analysed. The analysis on the results obtained for each group vary, but can be roughly classified as one of the two alternatives listed below:

1. The extracted database was also inspected by one or more other detection tools already presented and supposedly verified in the scientific literature. The analyses then consisted in comparing the results obtained by this inspection with the one obtained by the newly developed tool, much like as if the legacy tool was a benchmark. The focus in this validation approach was to identify if the new tool could detect those vulnerabilities pointed by the legacy tool and verify among the smart contracts deemed vulnerable by the new tool if they represent a real vulnerability that the legacy tool was unable to detect.
2. The extracted database, or more often the contracts deemed vulnerable by the newly developed detection tool, was then send to specialists in block chain security for a manual verification and a comparison of the new tool inspection with the specialists analysis. Note that, in this approach, the results obtained by the new tool are validated in the same "benchmark like" way as in the previous alternative. The difference between those validation methods is what is considered the benchmark itself.

Following this early validation approaches, a widely cited work recognized by performing a reliable validation test compared with the other contemporary works, was [7]. In the validation process for this vulnerability detection tool the authors noted that most of the smart contracts deployed in the Ethereum mainnet were simple contracts, probably

implemented for test reasons. The authors argued that allowing the validation database to contain multiple simple test smart contracts that are not specially implemented for validation reasons could result in a biased analyses were the newly developed vulnerability detection tool is able to detect various "naive" vulnerabilities that resulted from the test nature of this contracts. This could result in a false assertion that the detection tool have a high accuracy even if it fails to detect more complex vulnerabilities that really represent a threat to real world services or are maliciously introduced into the smart contract code.

In order to avoid this kind of unreliable analyses when performing the validation of vulnerability detection tools, the authors proposed to simple filter the obtained database by the number of transactions. The reason for this approach is based in the fact that test contract have arguably way less transactions then a real world service. All contracts with to few transactions would be excluded from the test database. The result was a 1000 smart contracts database that was arguably free from too simplistic test smart contracts and could provide a better test environment for the newly developed detection tool.

As new vulnerability detection tools in Ethereum smart contracts were developed, some researchers deemed necessary to do review works on these tools to compare the accuracy and detection capabilities of each one of them. On of the most cited of these review works was [6]. In this review the authors pointed that, in order to perform a reliable and fair comparison among vulnerability detection tools, one must test this tools in the same conditions, which includes a same database. The authors then proceeded in their analyses by testing the selected tools in two groups of smart contract: a database comprised of a large amount of unlabeled smart contract extracted directly from the Ethereum mainnet and a small database of labeled smart contracts.

The authors noted the absence of a labeled database with vulnerable smart contracts for vulnerability detection tools validation that they could use as the small labeled database. Pointing that this represents a major setback for the detection tools comparison processes, the authors compiled a group of 69 smart contracts, from several sources, that were examples of know vulnerabilities. While not ideal, this approach was deemed acceptable by the authors, who urged the community to address the lack of a reliable validation and comparison labeled smart contract database.

As the survey results, the [6] work noted a sharp difference from each of the vulnerability detection tools measured accuracy and the accuracy presented in the papers that presents each of the studied vulnerability detection tools. Figure 1 presents the obtained results. The best performing vulnerability detection tools, which was Mythril [4], could detect only 27% of the labeled vulnerabilities and detected 215 vulnerabilities in the unlabeled database. It is important to notice that a vulnerability detection tool could be developed specifically for a group of vulnerabilities. Considering that, a low detection accuracy in a comprehensive survey such as that its not enough to question the current

state of the art of the SC vulnerability detection tools. However, considering the output of all the tools, only 42% of the the labeled database were correctly labeled. This fact points to a gap in the current vulnerability detection capabilities.

Table 5: Vulnerabilities identified per category by each tool. The number of vulnerabilities identified by a single tool is shown in brackets.

Category	HoneyBadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0/19 0%	0/19 0%	4/19 21%	4/19 21%	0/19 0%	0/19 0%	0/19 0%	4/19 21% (1)	2/19 11%	5/19 26%
Arithmetic	0/22 0%	0/22 0%	4/22 18%	15/22 68%	11/22 50% (2)	12/22 55% (2)	0/22 0%	0/22 0%	1/22 5%	19/22 86%
Denial Service	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%
Front Running	0/7 0%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	2/7 29%
Reentrancy	0/8 0%	0/8 0%	2/8 25%	5/8 62%	5/8 62%	5/8 62%	5/8 62%	7/8 88% (2)	5/8 62%	7/8 88%
Time Manipulation	0/5 0%	0/5 0%	1/5 20%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	2/5 40% (1)	1/5 20% (1)	3/5 60%
Unchecked Low Calls	0/12 0%	0/12 0%	2/12 17%	5/12 42% (1)	0/12 0%	0/12 0%	3/12 25%	4/12 33% (3)	4/12 33% (1)	9/12 75%
Other	2/3 67%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	3/3 100% (1)	0/3 0%	3/3 100%
Total	2/115 2%	0/115 0%	13/115 11%	31/115 27%	16/115 14%	17/115 15%	10/115 9%	20/115 17%	13/115 11%	48/115 42%

Table 6: Total number of detected vulnerabilities by each tool, including vulnerabilities not tagged in the dataset.

Category	HoneyBadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0	10	28	24	0	0	6	20	3	91
Arithmetic	0	0	11	92	62	69	0	0	23	257
Denial of Service	0	0	0	0	27	11	0	2	19	59
Front Running	0	0	0	21	0	0	55	0	0	76
Reentrancy	0	0	4	16	5	5	32	15	7	84
Time Manipulation	0	0	4	0	4	5	0	5	2	20
Unchecked Low Level Calls	0	0	4	30	0	0	21	13	14	82
Unknown Unknowns	5	2	25	32	0	0	0	28	8	100
Total	5	12	76	215	98	90	114	83	76	769

Figure 1 – Results obtained in the [6] survey

As Mythril is developed by an audition company, ConsenSys, and maintained by the community, there is no scientific paper describing its development and tests. As an example of the disparity between the vulnerability detection tools in their tests and the results obtained in [6], the second best performing tool can be considered, which is Slither [7]. This vulnerability detection tool correctly detected only 17% of all the labeled vulnerabilities.

In [8] the authors created an API that enables developers to easily reproduce the methodology presented in [6] by providing the database used in a user-friendly platform. The authors also upgraded the labeled database to encompass 143 labeled SCs presenting the same 10 vulnerabilities. The presented framework also can automatically test new vulnerability detection tools and compare them to the ones that were test in the [6] review. To the knowledge of the author, this is the only platform of its kind presented to the developer community to this day.

Following [6] work, the authors of [26] conducted another review in the current state of the art vulnerability detection tools capabilities together with the proposal of a novel SC bugs and vulnerabilities classification. The database compiled to test the selected detection tools was made of one example of SC with each vulnerability, one without and, in some cases, misleading scenarios were the tool could easily misinterpret a certain piece of code. The results obtained were somehow better that the ones presented by [6], but still

were less promising than the ones presented by most of the detection tools presentation validation tests.

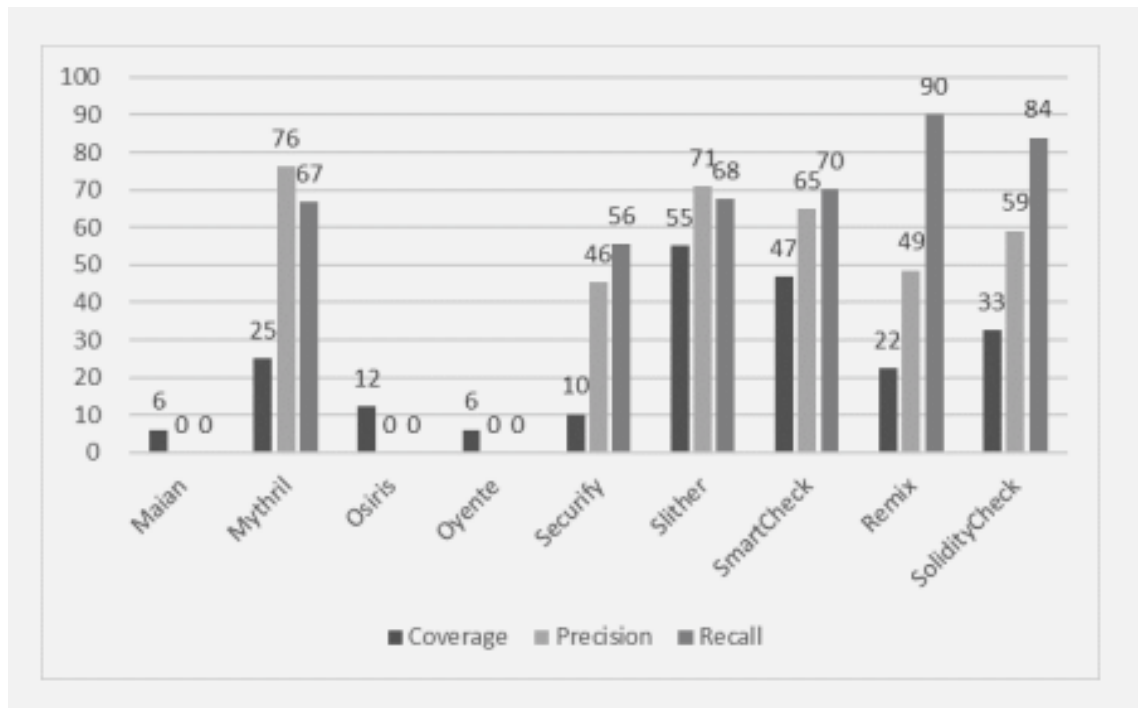


Figure 2 – Results obtained in the [26] survey

In order to obtain a labeled database of vulnerable smart contracts, the authors of [24] presented the following approach: first, a large amount of smart contracts were extracted by from the Ethereum mainnet. The authors then selected a group of five vulnerability detection tool that was deemed reliable and inspected the extracted database with the tools. To determine if a certain smart contract should be labeled as vulnerable ad which vulnerability should be attributed to it the authors performed a majority vote from the five detection tools. One of the challenges noted by the authors was the incompatibility among vulnerability detection tools which makes it difficult to execute them in parallel. To tackle this problem the authors implemented a docker container containing all the tools and their respective dependencies. The authors did not perform a survey of relevant detection tools using their data set but pointed to doing it in a future work.

The authors of [27] constructed a public labeled following a similar approach. First, they collected a wide array of safety reports by various consulting groups, most notably ConsenSys, a widely recognized block chain company that also works with smart contract security. Most of the collected reports were of complex DApps owned by various individuals and institutions. The authors then have put together a large group of block chain specialists and students of the CHINA university block chain lab to analyse both, the reports and their respective DApp smart contracts. This work resulted in a 1600 plus DApps database comprised of more than 23000 labeled smart contracts.

Unlike the previously cited work, the authors of this paper noted that, as complex decentralized applications, most of the smart contracts of a same DApp, have a high degree of dependence among themselves. This issue poses a challenge for vulnerability detection tools validation and comparison because every detection tool deals with dependencies in a unique way. To overcome this difficulty the authors came up with an ingenious solution: they created a dependency resolution tool that resolves the dependencies of a group of smart contracts before the vulnerability detection tool get to analyse them.

The authors used the constructed DApps database together with their newly developed dependencies resolution tool to perform a survey of some of the most cited vulnerability detection tools in the scientific literature. During the comparison of the chosen detection tools capabilities, the authors noted that most of them were not able to analyse most of the smart contracts in the database. The most successful tool in just being able to analyse a certain contract was able to inspect just above 8000 of the 23000 smart contracts, which corresponds to less than 35% of the database. This result prompted the authors to raise concerns about the quality of the current state of the art vulnerability detection tools. This 2023 work corroborated the concerns pointed out by [6], that most of the state of the art vulnerability detection capabilities may be severely overstated.

However, considering the labeling of a large SC database, the authors of [13] were the first to cite, in the work itself, the challenges posed by manually *a posteriori* labeling of a large SC database. In their work, they selected a group of 250 test SCs with vulnerabilities for the task of validating their newly developed vulnerability detection tool from a group of already filtered 5700 SCs. The authors explain that it took the work of 11 SC vulnerability experts more than 440 hours of work to label all the 5700 SC. This corresponds to roughly three months work, considering a workload of 8 hours per day, 20 days per month. Note that, in the case presented at [13] work, each SC were labeled in just above one hour.

Unlike the two previous works, the authors in [9] proposed a vulnerability detection tool validation framework based in a vulnerability insertion tool. The logic behind this approach is that it is unreasonable to construct a large labeled vulnerabilities in a reliable way. Therefore, the solution presented by this group was to collect a large group of smart contracts and perform vulnerabilities insertions using their newly developed tool. The inserted vulnerabilities consisted of "code snippets", which are small pieces of code that implement a certain vulnerability in a generic way. The insertion tool would then insert this code snippets in a location that it deemed fit. The vulnerabilities inserted would be labeled in the process of insertion and the validation would only consider inserted vulnerabilities.

3 Method

To construct a SC vulnerability database that possess all the characteristics described in the 4 chapter, the methodology adopted to guide the development of this work was as follows:

First, the current vulnerability detection tool validation, testing and comparison were surveyed. The selected papers were mostly detection tool presentations and surveys. In the survey, the focus was on the methodology regarding the testing of the tool(s) phase. The most cited of the reviewed papers were selected as the source of the proposed test framework method, which ended being the implementation of a vulnerability database that comprised of simple, representative and accurately labeled SCs to be used together with another much larger unlabeled database, as explained in the contributions section of chapter 6.

In order select the vulnerabilities that would be considered for addition in the database, a survey of the scientific literature were conducted in order to identify the most commonly cited vulnerabilities. Those that were most frequently cited in the selected papers were chosen. Besides that, all the chosen vulnerabilities were certified to be included in the Smart Contract Weakness Classification Repository [19], updated to [1]. It is important to notice that the selected nomenclature did not follow the cited sources, but it was certified that the vulnerability represented an entry or sub-entry of this classification.

To obtain a representative database, that is, a database that represents real usages of SCs, it was decided that all the developed SCs (except for one of each) would implement a real functionality. To determine which were the affected applications for every vulnerability, an AI tool was used. The results presented by the AI tool were then searched using Google search engine in order to identify if the results were trustworthy. If the the search resulted in many forums, technical sites, such Wired and Medium, blogs, scientific works (particularly [14]) and other media corroborating and commenting into the application, the AI output was considered accurate.

First, every considered vulnerability was studied and understood in order to determine which kinds of applications would be threatened by it. Then, some applications were randomly selected to be implemented with the vulnerability. The final developed SC was completely functional in terms of the selected application since the applications usability was also considered during the tests.

To keep the SCs developed different to one another in terms of how the vulnerability presets itself, the development of the database followed an approach that guaranteed three main implementation sources, the author, a generative AI and third party developers.

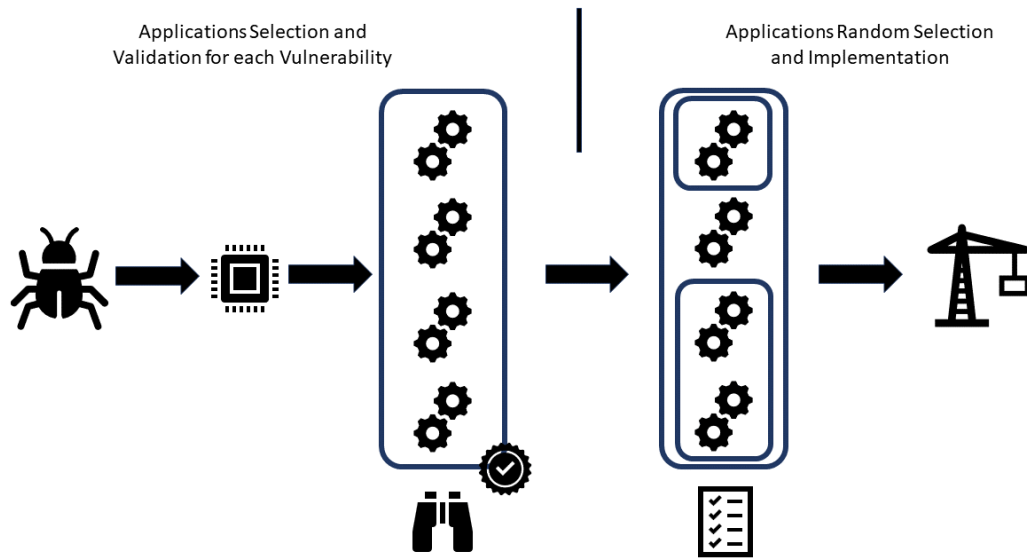


Figure 3 – Selection of SC Applications to be Implemented

Three SCs, on average, were manually implemented; another two SCs, on average, were implemented using a generative AI tool combined with a manual revision phase; other databases that presented similar characteristics to the developed work were also added. This step considered that each implementation method would result in, at least, small differences in the way a certain functionality was developed, much like the difference between human developer's codes.

Under the addition of SCs from other databases, only databases that were presented in the scientific literature were considered, except in the case of those sourced at ConsenSys. ConsenSys is a company that performs the audit of SCs and is regarded as a reference for smart contract vulnerability classification and detection. The inclusion criteria for SCs from external databases were that the vulnerability were clearly labeled and explained and that the SC were considered simple enough to be comparable to those developed. That is, they have less than 300 lines of code. The SCs that were not added to the database were also compiled, but were not included in the database.

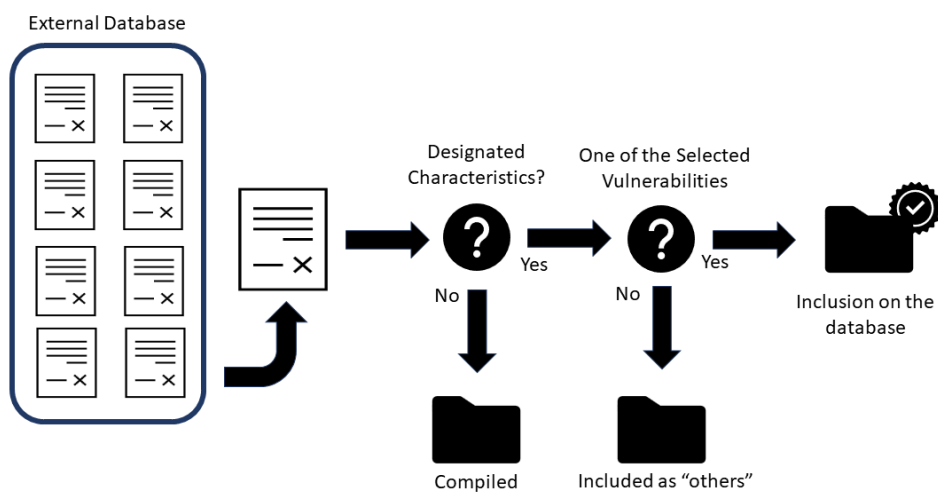


Figure 4 – Inclusion Framework for external Data Sets SCs

4 Requisites Specifications

The developed work aimed to implement a database similar to the labeled database presented in [6]. In order to achieve that, the developed database were specified as follows:

4.1 Database Characteristics Specifications

The proposed database will be used for validating and testing the capabilities of vulnerability detection tools. To achieve that the developed database should follow a series of specifications that will ensure that the results of its usage are accurate and reliable. First, the usage workflow of the database would be as follows: First, the user should provide the way for the database to call the selected vulnerability detection tool. Then the selected tool should be executed in all of the SCs in the database and the outputs consolidated in the selected directory. This workflow is also presented in figure 5.

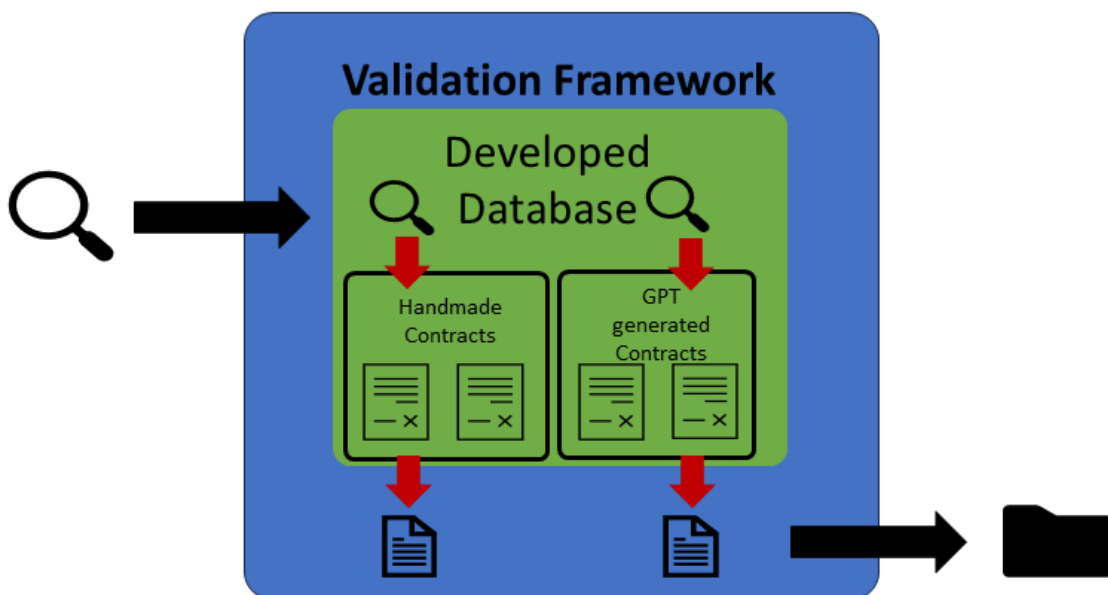


Figure 5 – Workflow

The developed database should also be able to be utilized autonomously. It is unreasonable to offer a database with dozens of SCs without offering the user the ability to run a certain vulnerability detection tool in all of the database automatically. The autonomous usage of the database should also be simple and transparent. That is, its usage should require only one simple interaction from the end user, such as the need to

write just a command line, and the module should transparently inform the user of each step it is performing.

Another important aspect that need to be taken into account is the compatibility of the database with the different detection tools. Some of them analyze only solidity code, other analyze byte code as well. Therefor the database ware implemented in solidity only and utilized the latest solidity version available to avoid any kind of incompatibility during the analysis. Besides that, the presented automation technique using a simple bash file following the vulnerability detection tool command enables a standardized execution mechanism.

Lastly, any user should be able to add its own SCs to the database without having to modify the automation module. That is achieved by setting the bash script to run the detection tool in all the files inside the Base directory. That enables the user to add any number of SCs to the database and even add new types of vulnerabilities by adding new folders. The project will be made open source in a convenient future date and will enable any user to contribute to the database or modify it for its personal usage.

4.2 Developed Smart Contract Specifications

The main characteristic regarding the labeled database presented in [6] is that its SCs were simple and accurately labeled. In order to implement a database that had this same characteristics it was decided that the SCs would have a maximum of 300 lines of code, ideally averaging 100 lines per contract. To obtain a high accuracy in the labels in the SCs in the database the vulnerability that the implemented SC would have was decided *a priori*, that is, before the implementation started. This step, together with test focusing in verifying the vulnerability exploitability, ensured that the vulnerability was present, functional and correctly labeled.

Another important aspect specified for the developed of the database is that all the SCs should be representative of the real life use cases of SCs in the Ethereum network. That is, all of the SCs in the database should represent an usual application presented in the Ethereum mainnet, such as fungible and unfungible tokens marketplace, ethers wallet, lottery, loan, SC based investments and others. However, each of the vulnerabilities selected would also include a SC that only implements the vulnerability itself. The reason for that is to have a basic example of each of the vulnerabilities and to serve as a base for the other SCs implementations. Besides the SC application being representative of real life usages of SCs, the vulnerabilities should also be considered relevant to the scientific and developer communities.

4.3 Externally Sourced Smart Contracts Specifications

As presented in the state of the art review (in chapter 2), small and incomplete databases of SCs labeled vulnerabilities examples have already been compiled in the scientific literature, particularly the 69 SC database compiled in [6]. Those examples SCs have the setback of being mostly simple examples, with many of them presenting a SC that implements only the vulnerability, without any other kind of application. This characterized this SCs as not representative of real life applications of SCs in the Ethereum network. However, their addition to the developed database could potentially enrich it by adding SCs implemented by third parties. This is positive because since every developer has its own unique way of writing code, some variations could arouse in the vulnerability implementation.

Besides that, some vulnerability detection tools offer a similar database of simple, example-like SCs that implement vulnerabilities. The reasons for those SCs is to offer the tool user a simple data set to observe the tool detection capabilities and to understand the vulnerabilities that the detection tool aims to detect. This SCs also can represent a positive impact in the developed database for the same reasons cited above. It is important to notice that many of this simple data sets were poorly documented. Many of them are labeled as containing a certain vulnerability, but does not specifies were it is located into the code.

5 Development

This section aims to provide a detailed explanation of the steps taken during the development of this work. The utilized technologies will be presented together with a justification for its usage. In addition to that, a timeline of activities will be presented. Note that this timeline is a portion of the complete activities that will be undertaken by the author in his master, which will be a continuation of this work.

5.1 Technologies

The development of this complex work requested a wide range of technologies. Those tools enabled the implementation of the project, as well as the tests and evaluation of the constructed database and other sub-systems that comprise the entire framework.

There are many different tools that enable a developer to implement and test smart contracts for the Ethereum network. Most of this tools offer a text editor configured to highlight the reserved words o the Solidity code. The chosen implementation environment for the development of the smart contracts that comprise of the presented database was the Remix IDE. This tool offers many advantages over simple Solidity text editors. Those are (see more in the Annex A):

1. **Access:** The Remix IDE [18] offers the ability to develop smart contracts online in a common web browser, such as Firefox or Chrome. The online environment offers a default cloud workspace directory for the user with no login required. The user can also create as many workspaces as he or she wants. But the most advantageous characteristic of the Remix IDE is the ability to connect and modify local files using a sub tool called Remixd.
2. **Capabilities:** The Remix IDE is a playground for smart contract implementations that offer way more than just a simple text editor for the Solidity programming language. The code is also analysed by Slither, a widely recognised vulnerability detection tool and the results are highlighted in the smart contract code. This capability was really useful in the implementation of the faulty smart contracts of the database and offered an early insight in the ability of this specific detection tool capability. Another capability of this tool that was imperative in the choice of using it is that it offer a wide array of compiler versions and a Ethereum blockchain test-net with a standard of 10 accounts. In the test-net the user can deploy his or hers developed smart contracts and perform transactions in order to test it. This built-in capability enabled the author of this work to easily test the developed smart

contracts along all the development. Besides this default test-net, it is possible to connect the Remix IDE with various blockchain test-nets effortlessly.

3. **Usability:** The Remix IDE has an intuitive interface that presents all its capabilities in an easy and orderly manner. This aspect is important because it enabled a smooth learning curve of its usage and minimized the time necessary to learn how to operate it.

Remix IDE offered the ability to develop and test smart contracts. But the default blockchain test-net offered built-in in this tool does not enable the user to see the operations undertaken in the blockchain level by the network. In order to be able to execute the smart contracts in a blockchain test-net with all the operation information Ganache [3] was used. This tool makes it possible to create a local, custom blockchain transparent to the user, that is, informations such as blocks information, mining information, individual transaction information, errors logs, etc. One limitation of Ganache is that, at the time of the development of this work, Ganache only accepts opcodes up to Solidity version 0.8.19.

This tool was useful for more in depth tests (see more in Annex B) in the developed smart contracts. It is important to notice that not all of the smart contracts implemented were tested in Ganache. The simple ones, such as the lead example of each of the vulnerabilities, did not require this amount of information as the tests performed in the Remix IDE were satisfactory enough.

The specifications of the project include a module that enables the user of the database to execute his or her newly developed vulnerability detection tool in all the database entries and compile the results in a specific directory chosen by the user. This simple functionality was developed as a simple bash script. The development of this script used the Notepad++ [10] text editor. This software offers a wide range of options for code highlighting, which makes it easier to develop simple codes without the need of a full suite, such as Visual Studio, or an advanced text editor, such as Visual Studio Code. No extra package was used in Notepad++.

Lastly, an IA tool was chosen for the fast implementation of smart contracts with minimized bias. The tool chosen for that was the GPT based chatGPT [17]. The selection of this tool was based in the recent reviews of this newly developed AI tool and its focus in code generation. It is important to notice that the author of this work is well aware of the limitations of this kind of technology and thoroughly tested all the functionalities in the smart contracts developed by chatGPT in order to assert that the code was developed correctly. The author also analysed the output code for vulnerabilities.

An important aspect of the tools presented is that all of them are free. This aspect was really important in order to avoid unnecessary costs in the development of this work.

Besides that, avoiding proprietary software and tools make it easier to reproduce the this work.

5.2 Database Construction

The first step in building the vulnerability database was to perform an in depth review of the types of vulnerabilities already know to the scientific and developer communities. As a scientific work, the performed survey focused in scientific papers. First, the string *solidity AND smart contract AND (vulnerability OR vulnerabilities OR bug OR bugs) AND (survey OR review)* were used in Google Scholar and Semantic Scholar scientific search engines. The results were then narrowed down by eliminating non-survey papers and by reading the remaining works abstracts. The selected works comprise the survey papers pointed out in the references of this work.

After selecting and exterminating the survey papers, roughly three kinds of vulnerabilities were identified in the Ethereum blockchain environment [12] [28] [11]. Those were:

1. **Solidity language related issues:** Those vulnerabilities arouse from the solidity language itself, which is the programming language used to implement SCs in the Ethereum network. This kinds of vulnerability include arithmetic bugs, gas related issues inter-contractual calls and other logic bugs.
2. **Ethereum VM related issues:** Those vulnerabilities are closely related to the way the Ethereum Virtual Machine operates. Those include the immutable characteristic of the Ethereum blockchain (note that not all blockchain are inherently immutable. Even the Ethereum network has already implemented the reversion of several blocks due to "theDAO" attack back in 2016 [23] [2]) and issues related to Ethers lost during a transfer.
3. **Ethereum Blockchain Design related issues:** Those vulnerabilities are related to the mining policy adopted in the Ethereum network and the block related variables that are available to the SCs. Besides that, this kind of vulnerabilities also include possible external data feeds that could be harmful to the smart contract correct execution. Examples of this vulnerability category are block timestamp and transaction ordering dependency and untrustworthy external oracles.

The sub-types of vulnerabilities presented in the figure above were not considered for the selection of vulnerabilities.

The vulnerability types selected from the three presented vulnerability kinds was the Solidity language related issues and some of the Ethereum blockchain design related

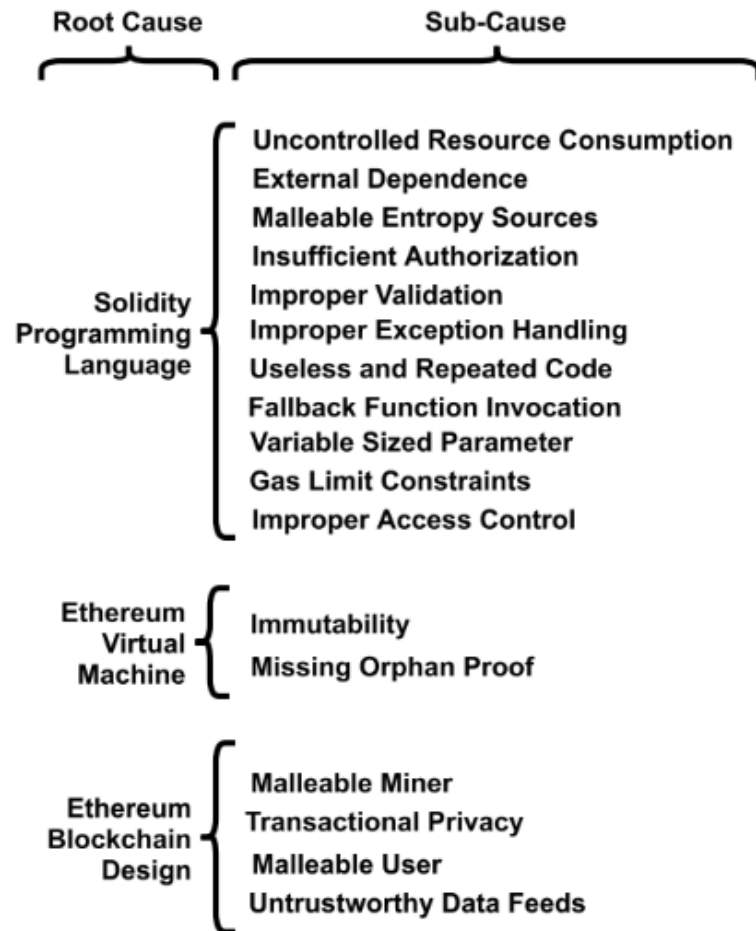


Figure 6 – The three types of vulnerabilities, as specified in [12]

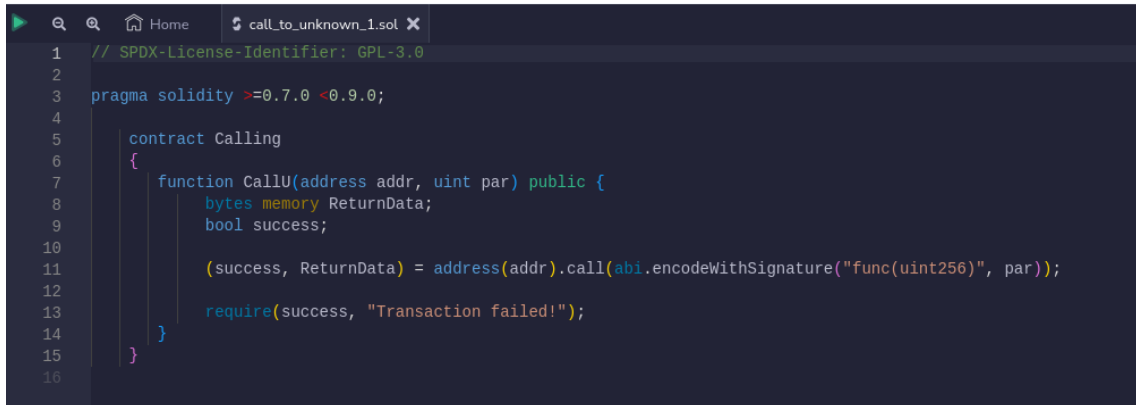
issues. This choice is due to the close relationship that this kind of vulnerability has with the implemented SC code. Ethereum blockchain design related issues have, in most cases, some aspect that manifests itself in the SC code. The Ethereum VM issues were not considered since this kind of vulnerability can hardly be detected in a SC code and mostly occurs due to the Ethereum VM specificities during a transaction.

With the selection of which kind of vulnerabilities would be considered, the selection of which vulnerabilities would be included took place. This task were achieved by identifying vulnerabilities how were present in more then two thirds of the revised survey papers with a cap of 10 vulnerabilities. This cap aimed to guarantee that the development of the database would be smoothly and with the vulnerabilities examples would have the desired quality given the project timeline.

The selected vulnerabilities were the following:

1. **Call to Unknown:** This kind of vulnerability is characterized by a call to another SC, that is ether a "call" or a "delegatecall", to an external untrustworthy address.

In that aspect every external call is deemed insecure and worthy of a warning, even if the called address is defined by a supposedly trusted administrator of the SC. It is important to remember that in a decentralized environment, in the user perspective, most other users are to be considered untrustworthy. An example of this kind of vulnerability is as follows:



```
1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity >=0.7.0 <0.9.0;
4
5 contract Calling
6 {
7     function CallU(address addr, uint par) public {
8         bytes memory ReturnData;
9         bool success;
10
11         (success, ReturnData) = address(addr).call(abi.encodeWithSignature("func(uint256)", par));
12
13         require(success, "Transaction failed!");
14     }
15 }
16
```

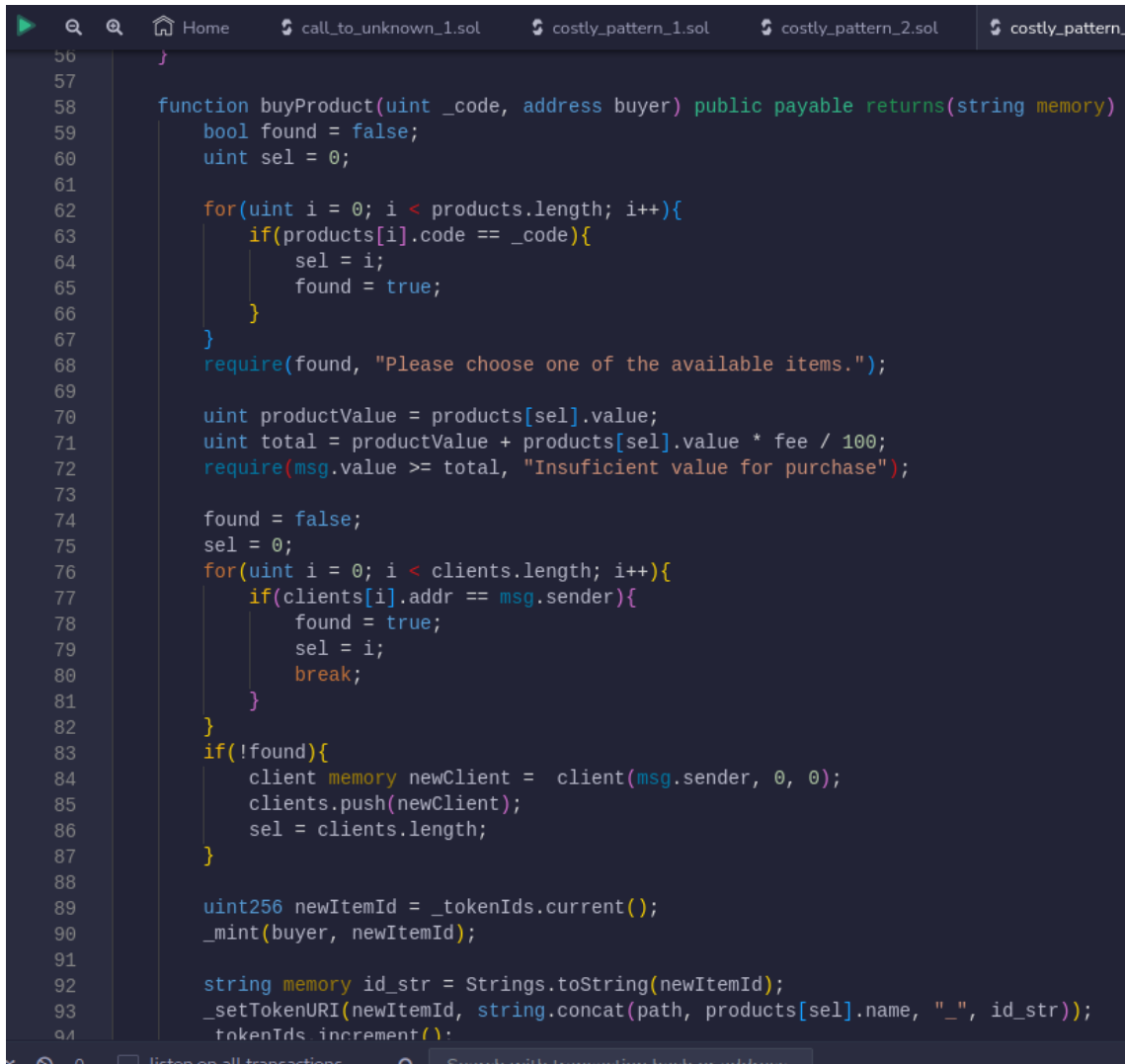
Figure 7 – Example of call to the unknown SC

Note that, in the line 11 of the code in figure 7, the SC executes a call to an address given by an unknown user in the function *CallU* arguments. This is a typical example of a call to the unknown vulnerability. The address that will be called by the function is unknown to anyone except the user calling the function until the moment that the SC function is executed.

2. **Gas Costly Pattern:** The gas costly pattern vulnerability is characterized by an computational expansive pattern in the SC code. This complex execution fluxes can result in a high gas fee to the function caller, even if the user service request is simple, like the reading of an array element, an can even empower a malicious agent to perform a denial of service in the SC by requesting unreasonable complex operations.

This kind of bug is common in a vast array of applications and programming languages, regardless of them having a relationship with blockchain technologies. Many of the computational complex operation regard as gas costly patterns are well known by the scientific and the developer communities. However, the Solidity language have some specificities that are unique to it, such as the logging of events in a transaction. For example, SC developers can log events during a transaction by using the "log" or the "event" functions. The implementation of "log" is simpler, but it has a bigger computational burden when compared to the "event" function.

In figure 8 one can notice the usage of two "for" structures (lines 62 and 76) in a same function. Those functions perform a linear unstructured search in two different arrays, products and clients respectively. "For" stances, specially in an unstructured



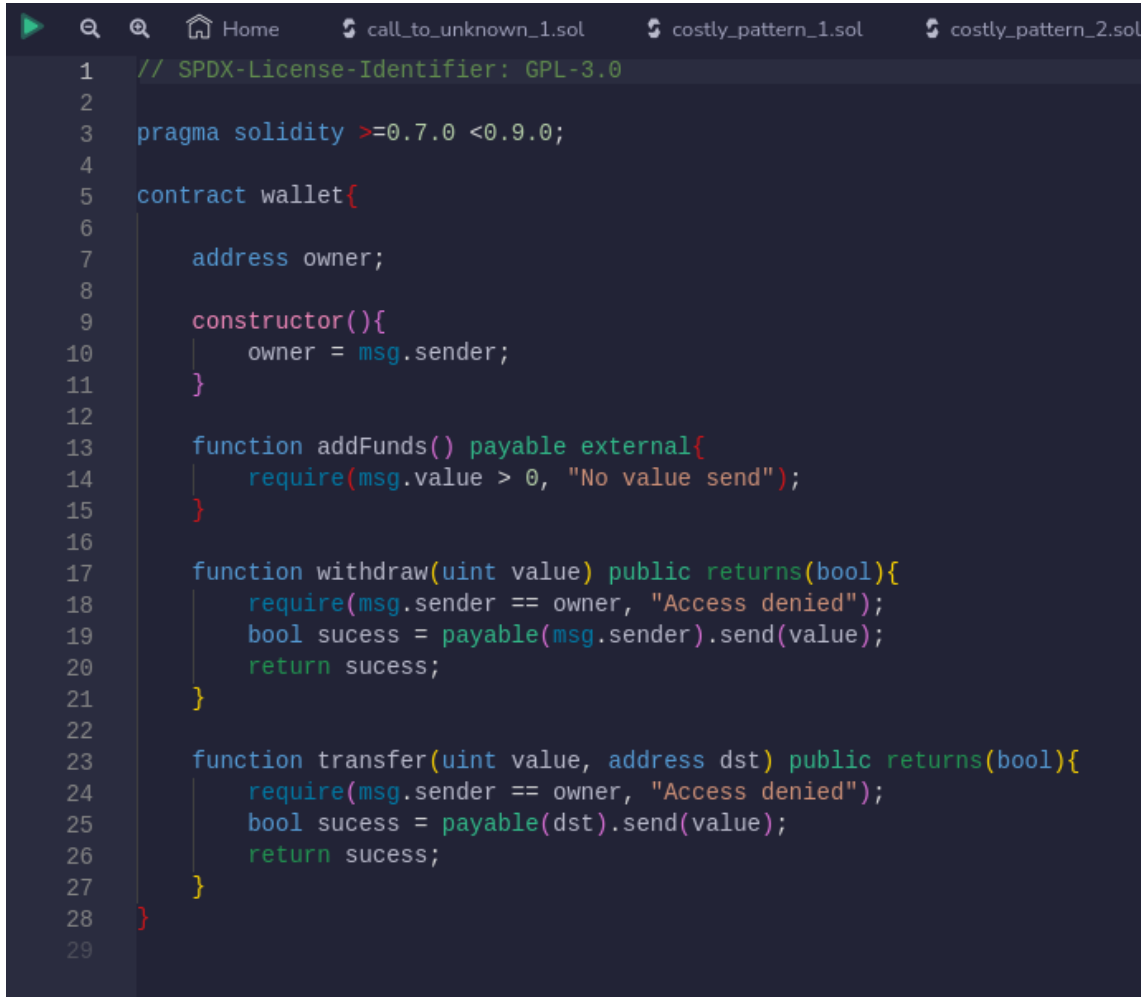
```
56 }
57
58 function buyProduct(uint _code, address buyer) public payable returns(string memory)
59     bool found = false;
60     uint sel = 0;
61
62     for(uint i = 0; i < products.length; i++){
63         if(products[i].code == _code){
64             sel = i;
65             found = true;
66         }
67     }
68     require(found, "Please choose one of the available items.");
69
70     uint productValue = products[sel].value;
71     uint total = productValue + products[sel].value * fee / 100;
72     require(msg.value >= total, "Insuficient value for purchase");
73
74     found = false;
75     sel = 0;
76     for(uint i = 0; i < clients.length; i++){
77         if(clients[i].addr == msg.sender){
78             found = true;
79             sel = i;
80             break;
81         }
82     }
83     if(!found){
84         client memory newClient = client(msg.sender, 0, 0);
85         clients.push(newClient);
86         sel = clients.length;
87     }
88
89     uint256 newItemId = _tokenIds.current();
90     _mint(buyer, newItemId);
91
92     string memory id_str = Strings.toString(newItemId);
93     _setTokenURI(newItemId, string.concat(path, products[sel].name, "_", id_str));
94     tokenIds.increment();
```

Figure 8 – Example of gas costly pattern SC

manner, are highly unadvised due to they having a high computational cost. The function "buyProduct" performs two of those operations, meaning that it is a perfect example of an non-optimized, gas costly SC function. Lastly, iterating in an array length, such as in the presented example, is a risky operation since many applications do not have a direct control over a certain array size.

3. **Gasless Send:** Gasless send, a sub-group of the mishandled exception venerability type, is a situation when an funds transfer fail without reverting the whole transaction. The issue of not reverting an unsuccessful operation is that the gas payed for the mining of the transaction is lost besides the possibility of spurious states in the SC execution, which could lead to the unusability of the SC service. This vulnerability arouse by the usage of low level functions, which are a class of functions in the Solidity language that ensures the continuity of the transaction regardless of the correct execution of the function. This functions offer a boolean return value that

is true in case of a successful function execution and false otherwise. This return parameter can be easily forgot during the SC implementation and make the contract vulnerable.



```
1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity >=0.7.0 <0.9.0;
4
5 contract wallet{
6
7     address owner;
8
9     constructor(){
10         owner = msg.sender;
11     }
12
13     function addFunds() payable external{
14         require(msg.value > 0, "No value send");
15     }
16
17     function withdraw(uint value) public returns(bool){
18         require(msg.sender == owner, "Access denied");
19         bool sucess = payable(msg.sender).send(value);
20         return sucess;
21     }
22
23     function transfer(uint value, address dst) public returns(bool){
24         require(msg.sender == owner, "Access denied");
25         bool sucess = payable(dst).send(value);
26         return sucess;
27     }
28 }
29
```

Figure 9 – Example of gasless send SC

In figure 9 a simple example of this kind of vulnerability is presented. Note that lines 19 and 25, of the "withdraw" and "transfer" functions respectively, implement an Ether transfer using "address.send(value)", which is a low level function. The return boolean of the ".send" function is not being used to revert the transaction in case of a transfer failure of this function, which characterizes a typical gasless send vulnerability.

4. **Hash Collision:** Hash collision is a situation where two distinct entries to a certain hash function result in the same output. This situation represent serious problems to hash dependant application such as message signing using hash functions. In a hash collision scenario, an attacker could provide a valid, supposedly signed, malicious entry to an application. This is the case in the Ethereum SCs environment as well. The lines 20 and 21, in the "addUsers" function, in the code presented in figure 10

represents a SC that accepts signed messages to add users. The users argument is configured in the following way:

$$[[user1, user2, \dots], [admin1, admin2, \dots]]$$

```

1 // SPDX-License-Identifier: GPL-3.0
2 //Source: https://gist.github.com/kadenzipfel/4c4d10f187f43e4ec1117f0229d45484#file-accesscontrol-sol
3
4 pragma solidity >=0.7.0 <0.9.0;
5 import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
6
7 contract AccessControl {
8     using ECDSA for bytes32;
9     mapping(address => bool) isAdmin;
10    mapping(address => bool) isRegularUser;
11
12    constructor(){
13        isAdmin[msg.sender] = true;
14    }
15
16    // Add admins and regular users.
17    function addUsers(address[] calldata admins, address[] calldata regularUsers, bytes calldata signature) public {
18        if (!isAdmin[msg.sender]) {
19            // Allow calls to be relayed with an admin's signature.
20            bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers));
21            address signer = hash.toEthSignedMessageHash().recover(signature);
22
23            require(isAdmin[signer], "Only admins can add users.");
24        }
25        for (uint256 i = 0; i < admins.length; i++) {
26            isAdmin[admins[i]] = true;
27        }
28        for (uint256 i = 0; i < regularUsers.length; i++) {
29            isRegularUser[regularUsers[i]] = true;
30        }
31    }
32 }
33

```

Figure 10 – Example of Hash Collision Sign Verification SC

The acceptance of signed messages as valid is not an issue by itself. The vulnerability emerges due to an ambiguity in a specific kind of in code signing process. Solidity offers two functions to encode and sign data, which are "abi.encode(data)" and "abi.encodePacked(data)". In the second case, as exemplified in lines 36, 39 and 42 of the "encode" function in figure 11, the functions perform a simplification of the encoded data before encoding and signing it. In the case of invoking the abi.encodePacked() function with the data structure presented as argument for the function in figure 10 the result would be equivalent to:

$$abi.encode([user1, user2, \dots, admin1, admin2, \dots])$$

If an attacker changes the arrays to the following:

$$[[user1, user2], [\dots, admin1, admin2, \dots]]$$

The encoded data would be the same:

$$abi.encode([user1, user2, \dots, admin1, admin2, \dots])$$

5. **Mishandled Exception:** The mishandled exception vulnerability is also a kind of issue that is not specifically related to SCs, but could be present in any kind of computational service. It is characterized by an error situation during the execution of a transaction that is not handled properly and can result in lost of funds (such as



```

25     admins[_admins[i]] = true;
26   }
27
28   for (uint256 i = 0; i < _users.length; i++) {
29     users[_users[i]] = true;
30   }
31 }
32
33 function encode(bytes memory data1, bytes memory data2, bytes memory data3, uint256 mode) internal pure returns(bytes memory)
34 {
35   if(mode == 1){
36     return abi.encodePacked(data1);
37   }
38   if(mode == 2){
39     return abi.encodePacked(data1, data2);
40   }
41   if(mode == 3){
42     return abi.encodePacked(data1, data2, data3);
43   }
44
45   revert();
46 }
47

```

Figure 11 – Example of Hash Collision Signing SC

in the case of a gasless send) and inconsistent states of execution in the SC. This kind of issue emerges with the usage of low level functions.



```

1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity ^0.8.0;
4
5 contract MishandledException {
6     mapping(address => uint256) public balances;
7
8     function deposit() public payable {
9         balances[msg.sender] += msg.value;
10    }
11
12    function withdraw(uint256 _amount) public {
13        require(balances[msg.sender] >= _amount, "Insufficient balance");
14
15        payable(msg.sender).send(_amount);
16    }
17 }
18

```

Figure 12 – Example of Mishandled Exception SC

The example presented in figure 12 the vulnerability presents itself in the way way as in a gasless send vulnerability. That is, a ".send" function is being used to transfer Ethers from the SC to another address and accepts the transaction (does not reverse it) regardless if the ethers transfer was successful. Another case that can result in a mishandled exception is the ill implementation of the code in the "catch" statement in an "trycatch" structure.

6. **Overflow and Underflow:** An overflow or underflow is a situation where a sum (or subtraction) results in a value bigger (or smaller) than the one supported by the used variable. This kind of vulnerability can result in ambiguous execution states

and in the circumvent of time restrictions for transacting in a certain SC. The most simple example possible for this kind of vulnerability is presented in figure 13.



```
1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity 0.7.6;
4
5 contract Sum{
6
7     function int_sum(int a, int b) public pure returns(int){
8         return a + b;
9     }
10
11 }
12
```

Figure 13 – Example of Overflow SC

Note that, in line 3 of the code presented in figure 13, the Solidity compiler version is set to 0.7.6, while other SC examples presented in this work use compiler versions above 0.8.0. This is due to the fact that, beginning in Solidity version 0.8.0, overflow and underflow situations are automatically considered exceptions and throw an error that reverts the transaction instead of proceeding. Besides that, all SCs deployed in the Ethereum mainnet that does not manually check for overflow and underflow situations are prone to this kind of issue.

The situation is aggravated by the fact that Solidity offers a wide range of unsigned integer sizes, beginning in "uint8", which is comprised of 8 bits, to "uint256", which is comprised of 256 bits. To reduce gas costs in transacting with the SC, developers usually set the unsigned integer to the lowest value thought to be necessary for the SC application to perform correctly. This practice, besides justified and encouraged when using Solidity version 0.8.0 or above, can make old SC particularly vulnerable to this kind of issues.

7. **Reentrancy:** Reentrancy issues are one of the most famous kinds of vulnerabilities present in Ethereum SCs. That is due to the fact that the "the DAO" attack, which was cited in the 1 chapter, and result in the lost of more than US\$320.000.000,00, was due to a reentrancy vulnerability. This attack was a landmark for the research of Ethereum SCs vulnerabilities and ways to try to tackle them because it drew a lot of attention to the sum that was misappropriated and the decision of the Ethereum community to revert all transactions that occurred after the attack in order to "undo" the theft.

This vulnerability is due to a simple logic error in the implementation of transfer functions in wallet SCs. The problem consists in transferring the funds in a transaction

before updating the internal variable that represents the user balance, as is shown in the lines 20 to 24 in figure 14. When a SC receives Ethers it can automatically run arbitrary code in the so called "fallback" function. This enables the attacker SC to call the function to withdraw funds again, before the balance value being updated. The logic behind performing that way is due to the sense that a failed transfer could result in an inconsistent state in the SC where a user end up losing funds due to not receiving the fund of the transfer and having its balance updated. This is line of thought comes from a developing mindset that is not compatible with a blockchain SC environment. In an SC where a transfer fails and is correctly managed, all the transaction is reversed, including the update to the balance value.

```
4
5 contract publicWallet {
6     mapping(address => uint) private balances;
7     bool called;
8
9     function deposit() public payable {
10        balances[msg.sender] += msg.value;
11    }
12
13    function withdraw(uint amount) public {
14        require(!called);
15        called = true;
16
17        uint bal = balances[msg.sender];
18        require(bal >= amount);
19
20        (bool sent, ) = msg.sender.call{value: bal}("");
21
22        require(sent, "Failed to send Ether");
23
24        balances[msg.sender] = bal - amount;
25        called = false;
26    }
27
28    function transfer(address to, uint amount) public returns(bool){
29        uint bal = balances[msg.sender];
30        require(bal >= amount);
31
32        balances[to] += amount;
33        bal -= amount;
34
35        return true;
36    }
37
38    function getBalance(address user) public view returns(uint){
39        return balances[user];
40    }
41 }
```

Figure 14 – Example of Reentrancy SC

It is important to notice that the reentrancy vulnerability is usually due to the transferring logic presented above together with the usage of the low level function "call" for performing the funds transfer. Other transferring functions, such as "send" and "transfer" usually avoid the problem, even if the transfer is being made before the user balance update. This is due to the gas limit that those functions implement, which is less than needed to run any command in the fallback function of the receiving SC. However, using those functions to keep transferring before updating the user balance is ill advised.

8. **Self-destruct Misuse:** SCs in the Ethereum blockchain can make use of a function called "self-destruct". This function essentially marks the SC as destroyed and makes it impossible to call any of its functions. This function also transfer all Ethers stored in the SC to a given address. The misuse of this function can result in two kinds of vulnerabilities, one related with the call to the self-destruct function in a SC, and another related to SCs that does not expect to receive values from "self-destructing" contracts.

In the SC presented in figure 15 the self-destruct function, in line 27, occurs when a user calls the "destroy" function. It is noticeable that any user can call this function, which could result in a malicious or unintentional destruction of the SC. This means a definitive denial of service to the service implemented by the SC until the owner or another user decides to deploy a similar SC, and that all the funds in the SC will be transferred to the "owner" address, regardless of the proposed fund management of the SC.

Figure 16 presents a different kind of vulnerability related to the self-destruct function. In this case, the SC implements a simple game where players are able to transfer the sum of 1 ether to the SC. When a transfer results in the balance of the SC being equal to 7, the player who did the last transaction is considered the winner receives all the 7 ethers from the SC.

In this scenario, an ill-intended user of the Ethereum network could try to lock the game in the EtherGame SC by transferring more than 7 Ethers to it. It is impossible to do so by directly transferring Ethers to this SC since the receiving function "deposit", in line 25, only accepts 1 ether per transaction. But if the user creates a SC that receives more than 7 Ethers and "selfdestruct it" marking the EtherGame SC as the receiver of the funds in his or hers SC, the EtherGame SC would be forced to accept the transaction, and the game would end with no winner. That is, all the Ethers deposit in this SC would be lost.

9. **Tx.origin:** This vulnerability is related to the usage of the tx.origin transaction flag. This variable keeps the address of the account who started the transaction. The misuse of this value as an equivalent to msg.sender, which is a flag that stores the

A screenshot of a code editor window titled 'selfdestruct_2.sol'. The code is written in Solidity and defines a contract named 'simpleWallet'. The code includes a license identifier, a pragma statement for Solidity version ^0.8.20, and a constructor that sets the owner to the sender. It also includes three public functions: 'add' which requires a positive value and increments the sender's balance; 'withdraw' which requires the sender's balance to be greater than or equal to the amount and transfers that amount; and 'destroy' which calls 'selfdestruct' on the owner. The code is as follows:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.20;
3
4 contract simpleWallet{
5
6     address private owner;
7     mapping(address => uint256) balances;
8
9     constructor(){
10         owner = msg.sender;
11     }
12
13     function add() public payable{
14         require(msg.value > 0);
15
16         balances[msg.sender] += msg.value;
17     }
18
19     function withdraw(uint256 _amount) public{
20         require(balances[msg.sender] <= _amount);
21
22         balances[msg.sender] -= _amount;
23         payable(msg.sender).transfer(_amount);
24     }
25
26     function destroy() public{
27         selfdestruct(payable(owner));
28     }
29 }
30
```

Figure 15 – Example of Self-destruct Unprotected Call SC

address of the caller of the latest action, could lead to serious security breaches in the target SC restricted access functions. For this attack to work the attacker needs to perform social engineering in the target SC administrators.

In the example presented in figure 17 the functions "withdrawFunds", "transferFunds" and "transferOwnership" of the "Wallet" SC perform an identity check in the function caller in the lines 18, 25 and 32 respectively. If an attacker tries to break the access restriction in those functions by calling them directly he or she won't have any success. However, the attacker could try calling those functions indirectly.

Suppose the attacker develops an SC as the example presented in 18. This SC references the Wallet SC in the "wallet" variable. The "awesomeFunction" calls the


```

4
5 // The goal of this game is to be the 7th player to deposit 1 Ether.
6 // Players can deposit only 1 Ether at a time.
7 // Winner will be able to withdraw all Ether.
8
9 /*
10 1. Deploy EtherGame
11 2. Players (say Alice and Bob) decides to play, deposits 1 Ether each.
12 2. Deploy Attack with address of EtherGame
13 3. Call Attack.attack sending 5 ether. This will break the game
14    No one can become the winner.
15
16 What happened?
17 Attack forced the balance of EtherGame to equal 7 ether.
18 Now no one can deposit and the winner cannot be set.
19 */
20
21 contract EtherGame {
22     uint public targetAmount = 7 ether;
23     address public winner;
24
25     function deposit() public payable {
26         require(msg.value == 1 ether, "You can only send 1 Ether");
27
28         uint balance = address(this).balance;
29         require(balance <= targetAmount, "Game is over");
30
31         if (balance == targetAmount) {
32             winner = msg.sender;
33         }
34     }
35
36     function claimReward() public {
37         require(msg.sender == winner, "Not winner");
38
39         (bool sent, ) = msg.sender.call{value: address(this).balance}("");
40         require(sent, "Failed to send Ether");
41     }

```

Figure 16 – Example of Unprepared for Self-destruct SC

withdrawFunds function in the Wallet SC with the value of 1 ether and transfer this value to the owner of the attack (lines 50 and 51 respectively). In this scenario, the msg.sender flag of the transaction will be the address of the "CoolService" SC, while the tx.origin flag will be the address of the account who called the "awesomeFunction". If the attacker could successfully convince one of the Wallet SC users to call the awesomeFunction in his or hers SC, the tx.origin flag would be the address of this user and the attack would be successful.

10. **Weak rand:** Random numbers generation is an important step in many computational operations. In Ethereum SC random number also play an important role in many situations. As in other computational applications, a good sources of entropy are scarce in the Ethereum environment. To tackle that issue a random number

```

1 // SPDX-License-Identifier: MIT
2
3 pragma solidity >=0.7.0 <0.9.0;
4
5
6 contract Wallet {
7     address public owner;
8
9     constructor(){
10        owner = msg.sender;
11    }
12
13    function addFunds() public payable returns(bool){
14        return true;
15    }
16
17    function withdrawFunds(uint256 value) public payable returns(bool){
18        require(owner == tx.origin, "Access denied!");
19        bool result = payable(msg.sender).send(value);
20
21        return result;
22    }
23
24    function transferFunds(address payable to, uint256 value) public payable returns(bool){
25        require(owner == tx.origin, "Access denied!");
26        bool result = to.send(value);
27
28        return result;
29    }
30
31    function transferOwnership(address newOwner) public returns(bool) {
32        require(owner == tx.origin, "Access denied!");
33        owner = newOwner;
34
35        return true;
36    }
37
38 }

```

Figure 17 – Example tx.origin SC

```

40 contract CoolService {
41     Wallet private wallet;
42     address private owner;
43
44     constructor(Wallet _wallet){
45         owner = msg.sender;
46         wallet = _wallet;
47     }
48
49     function awesomeFunction() public returns (string memory) {
50         wallet.withdrawFunds(1 ether);
51         payable(owner).transfer(address(this).balance);
52
53         return "Awesome return!!!";
54     }
55 }

```

Figure 18 – Example tx.origin attack SC

sourced from a high entropy source is used as the 256 bits seed to the build-in keccak random number generator. Keccak is a proven random number generation algorithm that is widely used in many computational areas. The vulnerability arises when the seed to the keccak algorithm comes from a low entropy source or can be manipulated by a third party.

The image shows a screenshot of a code editor with a dark theme. The editor displays Solidity code for a lottery application. The code includes a private array for player tickets, public variables for ticket price and a random number generator, a restricted modifier, a constructor, a buyTickets function, a random function, and a pickWinner function. The pickWinner function includes a 1% fee for the manager and a prize for the winner. The code is numbered from 6 to 44.

```
6 address[] private playerTickets;
7 uint256 public ticketPrice;
8 uint256 public addRand;
9
10 modifier restricted() {
11     require(msg.sender == manager, "Only the manager can call this function");
12     _;
13 }
14
15 constructor(uint256 price) {
16     manager = msg.sender;
17     ticketPrice = price;
18 }
19
20 function buyTickets() public payable {
21     require(msg.value >= ticketPrice * 1 wei, "Insufficient amount sent to buy tickets");
22     uint256 tickets = msg.value / ticketPrice;
23
24     for(uint i = 0; i < tickets; i++){
25         playerTickets.push(msg.sender);
26     }
27 }
28
29 function random() private view returns (uint) {
30     return uint(keccak256(abi.encodePacked(block.timestamp + addRand)));
31 }
32
33 function pickWinner(uint256 newPrice) public restricted {
34     uint index = random() % playerTickets.length;
35     address winner = playerTickets[index];
36     payable(manager).transfer(address(this).balance/100); //1% fee for the manager
37     payable(winner).transfer(address(this).balance); //prize for winner
38
39     // Reset the players array for the next round
40     playerTickets = new address[](0);
41     addRand++;
42     ticketPrice = newPrice;
43 }
44
```

Figure 19 – Example Weak Rand SC

The SC presented in figure 19 is a simple lottery application that generates a weak random number. The keccak function, in line 30, receives the block.timestamp and an iterator as seed. The block.timestamp variable is the timestamp of the signed block where the transaction is located in the Ethereum chain. This value can be tempered by the transaction miner, who has the ability to mine a certain block in a 900 milliseconds window. The developer of this SC also added an iterator to the block.timestamp seed in order to get an increased entropy. However, note that the iterator is updated only when a winner is selected, in line 41 of the "picWinner" function. That is, during a same game in this lottery SC, the random generator entropy source will be only that of the block.timestamp variable. This is a typical scenario of a weak random number generation.

After the selection of which vulnerabilities would be included in the constructed database, another literature review took place in order to provide an overview of the construction methods that were being used to validate, compare and review SC vulnerability detection tools capabilities. The papers were selected by searching Google Scholar and Semantic Scholar with the following search strings: *solidity AND ("validation database" OR "dataset" OR "datasets") AND (vulnerability OR vulnerabilities OR bug OR bugs) AND "Automated Analysis Tools"* and *solidity AND "smart contract" AND "vulnerability detection tool"*. From the presented results, 50 papers were selected by only inspecting the title. From those 50, a further selection was undertaken based on the papers abstract.

The selected papers were vulnerability detection tools reviews and vulnerability detection tools. The focus in the analysis of those papers were in the technique used to test vulnerabilities detection tools, particularly the construction of the test database in each case. The results of this literature review were already presented in the 2 chapter.

After the selection of the 10 vulnerability kinds and the literature review aimed at validation database construction methods, the database implementation took place. This implementation followed the henceforth steps:

5.2.1 Lead Example

The first step in the database construction process was the implementation of a SC that contains one of the selected vulnerabilities. This SC only included the functions needed to implement the selected vulnerability and was used as the main reference for the implementation of the vulnerability variations. Another important function of implementing an SC that included only the vulnerability was to help this work author to understand each of the studied vulnerabilities and have the opportunity to inspect the exploitation of those issues. This contracts are also intended to be used as a minimum baseline for the detection tools tests and validation.

It is important to notice that this lead example of each vulnerability was not necessarily developed specifically for this study or by the research authors. Some of the works reviewed brought examples of the vulnerability analyzed. When these example were tested and proved functional (older works utilized outdated solidity implementations) they were added without any modifications and the source was properly identified as a comment in the SC .sol file.

5.2.2 Variations Implementations

With a good lead example for reference, more SCs with variations of the vulnerability were implemented. Those contracts added other functionalities to the lead example. The new functionalities could be related to the vulnerability or not. For example, a wallet

contract with a reentrancy vulnerability could have an added functionality that enables the user to view his or her value invested in the contract. This functionality is not related to the core function of the vulnerability.

The following variations were considered as having an influence in the way the vulnerability works:

1. **Line Order Exchange:** The first and most simple variation in a vulnerable code itself is changing of the position of the lines that implement the vulnerability or the location of the vulnerability in the function itself. Most of the vulnerabilities present in the database have undergone this kind of variation. One example of this technique, in the case of a tx.origin vulnerability, would be to perform the identity check at the end or in the middle of the function, instead of in the start of it, as it is usually done.
2. **Conditional Exploitation:** Another kind of variation considered was the implementation of a SC that, in a same function or service provided, only presented itself if a specific set of conditions were met. One example of this kind of variations can be observed in the hash collision vulnerability. Only the abi.encodePacked encoding and signing function is vulnerable to hash collision due to ambiguity. However, a function can be implemented to encode data for several reasons, and the user can have the ability to choose which encoding mechanism he or she wants. In that case, the vulnerability would only be noted, in terms of execution, when the user selects the abi.encodePacked function as the encoding method.
3. **Vulnerable Service "Modularization":** Another variation considered in a vulnerability exploitation was the implementation of the vulnerability in a different function. In the case of the tx.origin vulnerability, that would mean that the caller verification would be done in a separate function or in a modifier, which is a specific kind of function in Ethereum SCs that is usually used to check any kind of data.
4. **Similar Vulnerability Implementation:** One of the most important kind of vulnerability variation that was considered, similar implementations consists in modifying the specific lines that implement a vulnerability. For example, in the case of an external call to the unknown, most of the examples presented in the literature consist of the "call" function. However, Solidity also implements a similar function called "delegatecall". The difference between the two is the the latter calls the external code in the same context as the caller SC. That is, the caller can access and change the state of the caller SC.

Another important example of this kind of vulnerability variation presents itself in the mishandled exception vulnerability. Most examples presented in the literature consider a mishandled exception situations where the SC code either ignores or does

not deal correctly with a low level function return call. However, for more general external calls exception handling, Solidity also provides a try/catch stance. The ill-implementation of the catch stance can also be considered a mishandled exception.

It is important to notice that not all the vulnerabilities considered are prone to all of the variations presented above. Besides that, when possible, multiple variations were implemented at once in order to configure a greater variation from the usual cases. The usage of other implementation methods, such as the generative AI, also contributed to adding variations to each of the vulnerabilities implementation variations.

The importance of performing this step were to obtain a database that truly implements real life applications of SC. Besides that, adopting this step in the developments adds to the challenge that the database presents to the tested vulnerability detection tool. This enables the user of the database to perform a most accurate test of the tool.

5.2.3 IA Assisted Implementation

Lastly, the selected GPT based tool was used to assist the implementation of more variations of each of the vulnerabilities. It is important to notice that the generated codes were not added to the database as they were implemented by the AI tool. The author have modified every generated code in order to guarantee its functionalities and the vulnerability exploitability. The implementation of examples using the AI tool was based in the following steps:

1. **Potential Threats:** First, a query was generated asking the tool about the kinds of threats that a certain vulnerability offered to different SCs applications. That is, the functionalities that could be affected by the vulnerability (based in the knowledge of the AI tool) were listed. This list was then filtered by the author in order to avoid results that could characterize an AI daydream. An AI daydream is a situation were an generative AI tool generates a result unrelated to the user query. This situation result from several reasons and can't be avoided completely.
2. **Threat Example:** After the enumeration of potential threats to possible Smart Contract functionalities and their filtration, two of them were selected to be implemented. This implementation consisted in querying a request for the AI tool of an example of the given vulnerability in an Smart Contract in a way the its is possible to observe the selected threat.
3. **Modification of Generated Code:** The code generated by the GPT based tool usually contained inconvenient characteristics, such as over simplistic SCs, Scs that did not contain the requested vulnerability and faulty code. That prompted the author to modify the generated codes in order to attain higher quality SCs.

In all the modifications the original functionality of the SC requested to the AI tool was preserved. Modifications only modified the implementation and added other functionalities that complemented the target functionality generated.

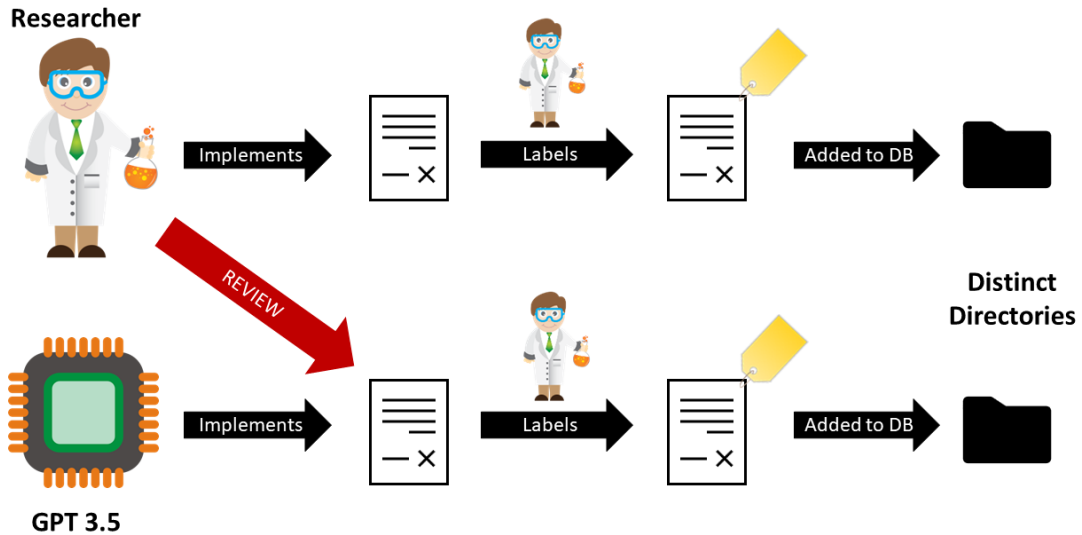


Figure 20 – SCs Implementation Workflow

5.2.4 Addition of Externally Implemented Smart Contracts

The main third party vulnerable, labeled, SC source selected was the one presented in the [6] work. This database already compile many other small databases and potentially presents greatly accurate labels since it was specifically curated for a scientific work. Besides that, some vulnerability detection tools that present a labeled example database were also included. The selected vulnerability detection tools that offered such database and were selected were [22], [4], [21] and [16].

Some of the additions from external SC databases were examples of vulnerabilities that were not part of the 10 selected vulnerabilities of the constructed database. This SCs were also included in the database in a separate directory called "others". When the automation module is called for the autonomous usage of the database, the results regarding this folder is also presented separately from the other SCs results.

It is noticeable that all of the third party SCs the were added to the database were from Solidity version 0.4.x, where "x" represents an arbitrary version. In order to keep the consistency of the database with the specification of implementing SCs with the latest Solidity version at the time of the development of this work, all the SCs that were not added to the "other" classification were updated to be compatible with Solidity version

0.8.19. The exclusion of the SCs that were not included in the 10 selected vulnerabilities presented in the update process were due to the time constrains.

At the end of this step, the developed database comprised 182 SCs, from which 50 were developed specifically for this work, 77 were added in the selected vulnerabilities and 55 were added as "others". The database itself already points to a gap in balance of the databases already developed since more than 53% of the added SCs that were not tagged as "others" were only from one kind of vulnerability (Over/Underflow). It is important to keep in mind this unbalance when using the presented database.

Database Composition			
Source	Call to Unknown	Constly Pattern	Gasless Send
Manual	3	3	3
GPT	2	2	2
Curated [6]	0	2	4
Mythril [4]	1	2	0
SmartCheck [21]	0	0	0
HoneyBadger [22]	0	0	0
Manticore [16]	0	0	0
TOTAL	6	9	9

Table 1 – Number of each Vulnerability in The Database by Source (Part 1)

Database Composition			
Source	Hash Collision	Mishandled Ex-ception	Over/Underflow
Manual	2	3	3
GPT	1	3	2
Curated [6]	0	1	15
Mythril [4]	0	2	0
SmartCheck [21]	0	0	0
HoneyBadger [22]	0	0	0
Manticore [16]	0	0	21
TOTAL	3	9	41

Table 2 – Number of each Vulnerability in The Database by Source (Part 1)

5.2.5 Database Smart Contracts Labeling

The labeling was undertaken before and during the development of each of the database's SCs. Before the start of the implementation of a certain SC it was decided which functionality the SC would have and which vulnerability would be present. During the development of the SC the vulnerability label was added as soon as the vulnerability

Database Composition			
Source	Reentrancy	Selfdestruct	TX.origin
Manual	3	3	3
GPT	2	2	2
Curated [6]	10	0	2
Mythril [4]	1	2	1
SmartCheck [21]	0	0	2
HoneyBadger [22]	0	0	0
Manticore [16]	1	0	0
TOTAL	17	7	10

Table 3 – Number of each Vulnerability in The Database by Source (Part 1)

Database Composition		
Source	Weak Random	Others
Manual	4	0
GPT	2	0
Curated [6]	9	25
Mythril [4]	1	2
SmartCheck [6]	0	0
HoneyBadger [6]	0	0
Manticore [6]	0	28
TOTAL	16	55

Table 4 – Number of each Vulnerability in The Database by Source (Part 1)

itself was implemented in the SC. This strategy guarantees that the labeled vulnerabilities in all the SC in the database have 100% accuracy considering the Solidity version during the time of development.

It is important to notice that all the SCs in the database are prone to having unlabeled vulnerabilities in addition to those that were labeled. The labeling step took into account the focus vulnerability who was considered during the development of each specific SC.

The labels were consolidated as a JSON file. The structure adopted to the organization of this file was a label for all the file content; a label for each of the 10 vulnerabilities present in the database; two labels for each vulnerability, representing the SCs that were manually developed and the ones that were developed with AI assistance; a list of the files from each vulnerability, implemented in a specific way (manual or GPT) with the information regarding the file name and the name of the SC implemented in the specified file; a list of vulnerabilities for each SC with the information of the function where each vulnerability is located, the code line and the severity of the vulnerability. In figure 21 presents the vulnerability label for the file "call-to-unknown-1.sol", which is the file that implements the code in figure 7.

```

1  {
2  "datasetLabels": {
3  "call_to_uknown": {
4  "Manual": [
5  {
6  "fileName": "call_to_uknown_1.sol",
7  "contractName": "Calling",
8  "vulnerabilities": [{
9  "function": "callU",
10 "line": 11,
11 "severity": "high"
12 }]}
13 },

```

Figure 21 – JSON file configuration

5.3 Automation Module Implementation

The automation module of the database enables the user to execute a certain vulnerability detection tool in all the implemented SCs automatically, that is, without the need to the user to execute the command to call his or hers vulnerability detection tool in every one of the SCs in the database manually. The implementation consisted in a simple bash code that reads a string that takes as argument the calling code to the vulnerability detection tool with all its required arguments and the output directory to the tool reports. The script then executes the desired vulnerability detection tool in all the SCs in the database and consolidates all the reports in a directory selected by the user, as explained in chapter 4. The resulting script is presented in figure 22.

A bash code is a type of file in Linux systems that enables the caller to run the command lines written in the code automatically in the Linux terminal. To call the script the user only need to type in the command terminal the following line:

```
.run-on-database.sh -c "cmd with args" -o output/directory -j
```

In the command line presented above the following elements can be identified:

1. **.run-on-database.sh**: this element calls the script in the "run-on-database.sh" file
2. **-c "cmd with args"**: the flag "-c" indicates that everything inside the quotes will be interpreted as the command to call the vulnerability detection tool that the users wants to run in all the SCs of the database. The calling command for the tool should include all the tool flags that the user wants to use in the execution, and the place were the user text the smart contract name in the tool command should be substituted with a "%sc". This symbol will be inform the script were the SC name should be inserted in the detection tool command line.

```

1  #!/bin/bash
2
3
4  usage() {
5      echo "$(basename "$0") [-c 'command' -o out_dir -j]"
6      exit;
7  }
8
9  out_file_type="txt"
10 while getopts c:o:j flag; do
11     case "${flag}" in
12         c) command=${OPTARG};;
13         o) outdir=${OPTARG};;
14         j) out_file_type="json";;
15         ?) usage;;
16     esac
17 done
18 if [ -z "${command}" ] || [ -z "${outdir}" ]; then
19     usage
20 fi
21 if [ ! -d $outdir ]; then
22     mkdir $outdir
23 fi
24
25 vulnerabilities=$(ls Base)
26 for vulnerability in $vulnerabilities; do
27     print "\nAnalyzing files on vulnerability: $vulnerability\n"
28     development_types=$(ls Base/$vulnerability)
29     for development_type in $development_types; do
30         contracts=$(ls Base/$vulnerability/$development_type)
31         for contract in $contracts; do
32             out_file_name=${vulnerability}_${development_type}_${contract%.*}.${out_file_type}
33             run_instruction=$(echo $command | sed "s/%scl/Base/$vulnerability}/${development_type}/${contract}/g")
34             run_instruction="$run_instruction"
35             echo "Analyzing contract $contract with command: $run_instruction"
36             $run_instruction > $outdir/$out_file_name 2> /dev/null
37         done
38     done
39 done

```

Figure 22 – Bash Code that Implements the Automation Module

3. **-o output/directory:** the flag "-o" indicates that the following string will be interpreted as the output directory where the user wants to compile all the execution reports that the vulnerability detection tool generates. All the output files follow same convention: SCFileName-ManualGPT.txt. Note that the output is a text file.
4. **-j :** the flag "-j" is an optional flag that indicates that the detection tool output will follow the JSON format and should be identified as such. In the presence of this flags all the output files will be JSON files (.json) and will continue to follow the same name convention as specified for the text files in the item above.

In order to enhance the user usability, a simple help menu was also implemented and is shown as a result of any incorrect call to the script. Besides that, all the steps undertaken automatically by the script are printed in the command terminal aiming to guarantee that the user is aware of every step undertaken. In figure 23 and example of the output generated by the automation script is presented. In this case, the vulnerability detection tool selected for an automatic test in the database was SmartCheck [21].

```

ryan@ryan-virtual-machine:~/Documents/Contracts/TCC$ ./run_on_database.sh -c "smartcheck -p %sc" -o saida
Analyzing files on vulnerability: call_to_uknown
Analyzing contract call_to_uknow_4.sol with command: smartcheck -p Base/call_to_uknown/GPT/call_to_uknow_4.sol
Analyzing contract call_to_uknow_5.sol with command: smartcheck -p Base/call_to_uknown/GPT/call_to_uknow_5.sol
Analyzing contract call_to_uknown_1.sol with command: smartcheck -p Base/call_to_uknown/Manual/call_to_uknown_1.sol
Analyzing contract call_to_uknown_2.sol with command: smartcheck -p Base/call_to_uknown/Manual/call_to_uknown_2.sol
Analyzing contract call_to_uknown_3.sol with command: smartcheck -p Base/call_to_uknown/Manual/call_to_uknown_3.sol

Analyzing files on vulnerability: costly_pattern
Analyzing contract costly_pattern_4.sol with command: smartcheck -p Base/costly_pattern/GPT/costly_pattern_4.sol
Analyzing contract costly_pattern_5.sol with command: smartcheck -p Base/costly_pattern/GPT/costly_pattern_5.sol
Analyzing contract costly_pattern_1.sol with command: smartcheck -p Base/costly_pattern/Manual/costly_pattern_1.sol
Analyzing contract costly_pattern_2.sol with command: smartcheck -p Base/costly_pattern/Manual/costly_pattern_2.sol
Analyzing contract costly_pattern_3.sol with command: smartcheck -p Base/costly_pattern/Manual/costly_pattern_3.sol

Analyzing files on vulnerability: gasless_send
Analyzing contract gasless_send_4.sol with command: smartcheck -p Base/gasless_send/GPT/gasless_send_4.sol
Analyzing contract gasless_send_5.sol with command: smartcheck -p Base/gasless_send/GPT/gasless_send_5.sol
Analyzing contract gasless_send_1.sol with command: smartcheck -p Base/gasless_send/Manual/gasless_send_1.sol
Analyzing contract gasless_send_2.sol with command: smartcheck -p Base/gasless_send/Manual/gasless_send_2.sol
Analyzing contract gasless_send_3.sol with command: smartcheck -p Base/gasless_send/Manual/gasless_send_3.sol

Analyzing files on vulnerability: hash_collision
Analyzing contract hash_collision_3.sol with command: smartcheck -p Base/hash_collision/GPT/hash_collision_3.sol
Analyzing contract hash_collision_1.sol with command: smartcheck -p Base/hash_collision/Manual/hash_collision_1.sol
Analyzing contract hash_collision_2.sol with command: smartcheck -p Base/hash_collision/Manual/hash_collision_2.sol

Analyzing files on vulnerability: mishandled_exception

```

Figure 23 – Example of Automatic Usage of the Database Using SmartCheck

5.4 Tests and Evaluation

One of the most important steps in any system development is the testing of the implemented functionalities. The developed database was tested in two different ways. The first one considered the implementation of each of the SCs individually and its of their functionalities. The tests undertook were the following:

1. **Compilation:** The smart contract were compiled using the Remix IDE. The IDE offers a comprehensive analysis of the code besides simple logic checks before compiling it, such as the results of Slither, a vulnerability detection tool that is well regarded in the scientific community. Besides that, this platform offers warnings for situations that does not prevent compilation but could result in execution issues, such as deprecated code, unused local variables and incorrect or inconsistent function visibility.

Every smart contract was modified until no errors were reported by the platform using the latest Solidity version compatible with Ganache, which was 0.8.19. The modifications also sought to resolve most warnings, but no all of them. The reason for that is that some of the vulnerabilities and its side-effects in the SC code are already know in the community and therefor marked in the code as warnings. It is important to notice that, in the case of Gasless Send vulnerabilities, the compilation used Solidity version 0.7.6.

2. **Vulnerability Exploitability:** After all errors and non-related warnings were resolved, the vulnerability was tested in order to guarantee that it was exploitable.

That included the correct implementation of the vulnerability and a logical situation that permitted the vulnerability to induce unwanted behavior in the SC. For example, an reentrancy vulnerability for funds transfer is only usefully, in terms of its maleficent usage, in wallet SCs that receives ethers from many different user. In this scenario a wallet implemented for the usage of a single user would be considered a logical error in the vulnerability implementation.

The tests were undertaken in both, the test environment build-in in the Remix-IDE and in a tailor made local blockchain configured in the Ganache testing tool. The test followed a unitary test strategy where all possible input were tested in order to ensure the vulnerability usability. Multiple scenarios were also tested in order to observe how the vulnerability would affect the SC execution. Those scenarios included, for example, invalid requests.

3. **Side Functions:** The DAPPScan [27] work pointed out that many of the test databases constructed specifically for testing vulnerability detection tools comprised of the so called "toy smart Contracts". The authors defined those as simple SCs that included the basic functions needed to observe the vulnerability in action. The authors pointed that this simple SCs have few to none real world application and does not represent the real world applications that the SCs are usually for. In order to avoid the problems pointed in the cited work, all the SCs in the database implement a common real world application.

Because of that, all other functions in each of the SCs, besides the ones affected by the vulnerability that the SC implements, were tested in order to guarantee their functionality. This test aimed to assert that the SCs really implemented the desired real life application and could henceforth be considered examples of real life SCs.

All the SCs included in the were able to pass all the specified tests presented above and were then deemed with sufficient quality to be considered as part of the database.

Before conducting the cited tests in each of the SCs, a second test were undertook. The second test aimed to ensure that all of the SCs in the database could be swept by vulnerability detection tools and that the automation script developed was working correctly. Besides the automation script functionality check, this test, in the context of the two database test framework presented by [6], would assert that the database could be considered the small, simple and highly accurately labeled database presented in the work.

This test consisted in selecting two vulnerability detection tools, one static and another dynamic, and to observe the output generated by both of them when the automatic script was called with each one of them. The tools selected were SmartCheck (static) [21], due to its simplicity and fast execution, and Mythril (dynamic) [4], due to its active development community and wide range of detected vulnerabilities. The test was considered

successful when both tools were able to generate valid output reports for all the SCs in the database. It is important to notice that a valid output does not necessarily mean a correct output. That is, the stopping criteria of the test considered as valid outputs non-black reports and the absence of error messages.

6 Final Considerations

6.1 Conclusions

The main objective of this work, which was the development of a simple, accurately labeled vulnerable SC database, that was representative of the real life applications of Ethereum SCs, was successfully achieved. The developed database also demonstrated the desired characteristic of being able to be correctly scanned by both, static and dynamic vulnerability detection tools, regardless of the tools ability to correctly identify the vulnerabilities presented in the database. This important characteristic enables the the presented database to be used as a simple benchmark in evaluating vulnerability detection tools detection capabilities in regards to the vulnerabilities footprint in a Solidity SC in a simplified environment.

The main difficulties noticed during the development of the database were the following:

1. The quality of the SCs implemented by generative AI was poorer then expected. All the SC that were automatically generated were thoroughly reviewed, as explained in chapter 5. However, many SCs generated by the GPT based AI were only partially functional or presented a vulnerability who was not exploitable. In rare cases, the output ignored the part of the query asking for a vulnerable SC. This resulted in a greater then expected time reviewing and rewriting entire portion of the SCs generated.

There are two main reasons identified as the root cause of this issues when implementing vulnerable SCs via generative AI. Firstly, the chosen AI is likely trained to avoid generating any programming code that can be harmful in any way, specially if the query ask specifically to generate a vulnerable code. The second reason is the limitations of the language recognising mechanism of the chosen generative AI as well as the possibility of the queries being inaccurate or mildly accurate.

2. Blockchain technologies, such as SCs, are experiencing the effects of the so called hype cycle. This phenomenon is an observed reaction from developers, scientists and users regarding the adoption of a certain technology. Developed by Gartner Company, the hype cycle of any technology is divided into five regions. Particularly, Gartner identifies one of this regions as the so called "Peak of Inflated Expectations", which corresponds to a great volume of adoption of the technology even if the

environment where the technology is being applied does not have a direct relation with the technologies real capabilities.

According to Gartner, in 2019, SCs (figure 24) were in the "peak of inflated expectations". Despite the wide adoption of this technology at this point, the spread of information regarding SCs functionalities, capabilities, vulnerabilities and security mechanism was occurring in a fast, unorganized pace. This resulted in a great deal of information regarding SCs vulnerabilities being generated without the academic formalism, which resulted in many non-scientific sources of information.

Hype Cycle for Blockchain Business, 2019

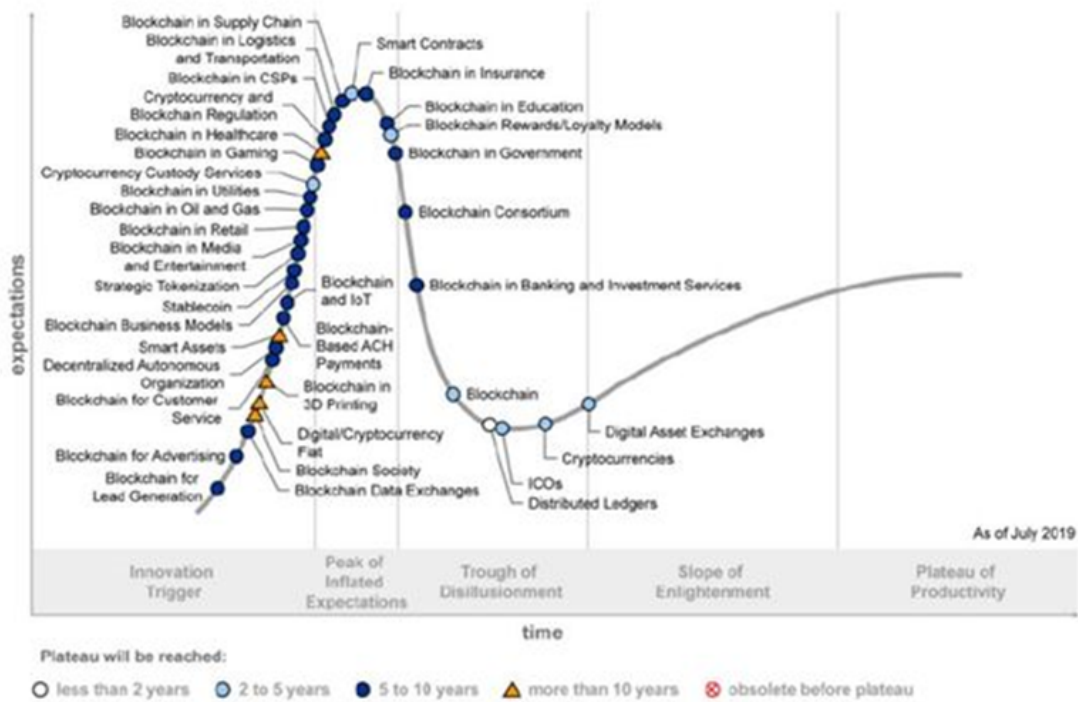


Figure 24 – Blockchain Hype in 2019

Regarding that aspect, filtering high quality information was a challenging task. Besides that, some vulnerabilities are not yet fully characterized in the scientific literature. In the scope of this work, the author decided to consider only information that was corroborated by scientific works, with the exception of Mythrill vulnerability detection tool. This exception is due to this tool being the result of the work of ConsenSys, a well known SC audition company, and being consistently updated by the developed community. This choice limited the amount of information who was taken into account during the development of this work.

6.2 Contributions

The most important contribution to the research area comprised of the development and validation of vulnerabilities detection tools is the database itself. As noted in [6] and to the knowledge of the author of this work there is no other database that was developed specifically for vulnerability detection tool validation following the method adopted in the development of this work.

It is important to notice that the development of all the smart contracts were made by the author of this work. Some code pieces from external sources, such as internet articles and github repositories, were also used and are all identified in the SC code as a comment. That is, the corresponding piece that was taken from an external source is labeled as such and the source is also provided.

As its noticed by [27], much of the vulnerability detection tools of Ethereum SCs are not able to be executed in complex scenarios, such as a complete DApp comprised of several interconnected SCs, internal and external libraries. As much as a setback as that is, [6] points that a complete validation should follow the following steps:

1. First, the vulnerability detection tool should be tested in a simplified environment. That is a database comprised of SCs that does not have many dependencies and complex execution fluxes, but have their vulnerabilities labeled *a priori* and with a high degree of certainty.
2. Then, the detection tool should be tested in a second database, this time comprised of real life SCs extracted from the Ethereum mainnet. The second database is not labeled and the results of the tested vulnerability detection tool should be analyzed in terms of false positives as in [6].

This project proposes the usage of the presented database as the simple *a priori* labeled test database and any other wide, real life, unlabeled SC database. That is, the same steps undertook in the [6] work should be taken using the presented database as the set of labeled SCs. Besides that, since the database will be made public, the author of this work invites all interested researchers and developers to contribute with the database, since paving the ground for future surveys and improvements in the state of the art in vulnerability detection tools.

6.3 Future Works

After the development of this work many new possible research paths were identified as worthy of investigation. The corresponding future research opportunities are:

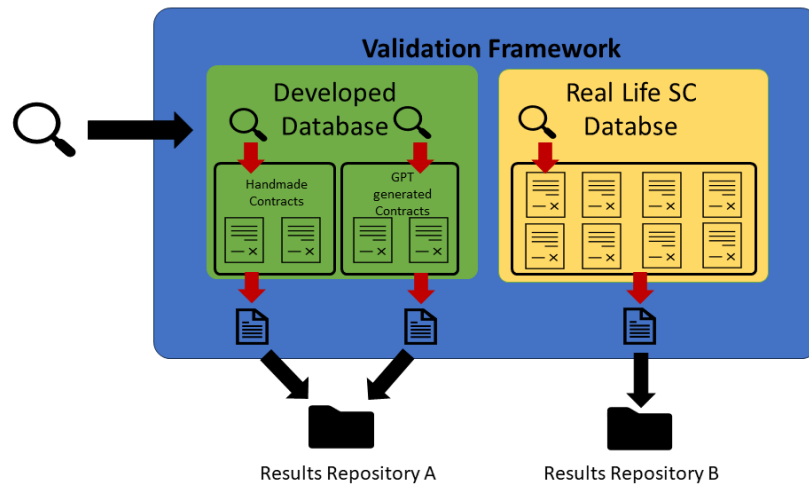


Figure 25 – Proposed Workflow for Vulnerability Detection Tool Validation

1. The further development of the constructed database, prior to its expansion, would be a worthy endeavour. Tasks identified as possible future steps include the labeling of other vulnerabilities in the database, besides the focus ones; the update of the SCs deemed "other"; the standardization of the vulnerabilities names according to preferably [1].
2. The developed smart contract database represents only ten types of vulnerabilities. According to the Smart Contract Weakness Classification Repository, there were over 30 types of vulnerabilities already know to the developer community in the time of the development of this work. It is important to notice that new vulnerabilities are being discovered as time passes. Therefore, expanding the database representative of vulnerabilities by adding other types of vulnerabilities that were not yet included is worthy endeavour for future development.
3. Since the developed database adopted a new method of implementation, it would be reasonable to conduct a survey of the current state of the art vulnerability detection tools capabilities using the database. The survey, together with other surveys already present in the literature, would enable the scientific community to have a broader view of the current state of the art.

Bibliography

- [1] Chaals Nevile et al. *EEA EthTrust Security Levels Specification v1*. Access in 2023. URL: <https://entethalliance.org/specs/ethtrust-sl/>.
- [2] Vitalik Buterin. *Hard Fork Completed*. Access in 2023. URL: <https://blog.ethereum.org/2016/07/20/hard-fork-completed>.
- [3] ConsenSys. *Ganache*. Access in 2023. URL: <https://trufflesuite.com/ganache/>.
- [4] ConsenSys. *Mythril*. Access in 2023. URL: <https://github.com/ConsenSys/mythril>.
- [5] Filippo Contro et al. “EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode”. In: *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 2021, pp. 127–137. DOI: [10.1109/ICPC52881.2021.00021](https://doi.org/10.1109/ICPC52881.2021.00021).
- [6] Thomas Durieux et al. “Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. ICSE ’20*. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 530–541. ISBN: 9781450371216. DOI: [10.1145/3377811.3380364](https://doi.org/10.1145/3377811.3380364). URL: <https://doi.org/10.1145/3377811.3380364>.
- [7] Josselin Feist, Gustavo Grieco, and Alex Groce. “Slither: A Static Analysis Framework for Smart Contracts”. In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 2019, pp. 8–15. DOI: [10.1109/WETSEB.2019.00008](https://doi.org/10.1109/WETSEB.2019.00008).
- [8] João F. Ferreira et al. “SmartBugs: A Framework to Analyze Solidity Smart Contracts”. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. ASE ’20*. Virtual Event, Australia: Association for Computing Machinery, 2021, pp. 1349–1352. ISBN: 9781450367684. DOI: [10.1145/3324884.3415298](https://doi.org/10.1145/3324884.3415298). URL: <https://doi.org/10.1145/3324884.3415298>.
- [9] Asem Ghaleb and Karthik Pattabiraman. “How Effective Are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2020*. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 415–427. ISBN: 9781450380089. DOI: [10.1145/3395363.3397385](https://doi.org/10.1145/3395363.3397385). URL: <https://doi.org/10.1145/3395363.3397385>.

- [10] Don Ho. *Notepad++*. Access in 2023. URL: <https://notepad-plus-plus.org/>.
- [11] Zulfiqar Ali Khan and Akbar Siami Namin. “Ethereum Smart Contracts: Vulnerabilities and their Classifications”. In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, pp. 1–10. DOI: [10.1109/BigData50022.2020.9439088](https://doi.org/10.1109/BigData50022.2020.9439088).
- [12] Satpal Singh Kushwaha et al. “Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract”. In: *IEEE Access* 10 (2022), pp. 6605–6621. DOI: [10.1109/ACCESS.2021.3140091](https://doi.org/10.1109/ACCESS.2021.3140091).
- [13] Zeqin Liao et al. “SmartDagger: A Bytecode-Based Static Analysis Approach for Detecting Cross-Contract Vulnerability”. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. <conf-loc>, <city>Virtual</city>, <country>South Korea</country>, </conf-loc>: Association for Computing Machinery, 2022, pp. 752–764. ISBN: 9781450393799. DOI: [10.1145/3533767.3534222](https://doi.org/10.1145/3533767.3534222). URL: <https://doi.org/10.1145/3533767.3534222>.
- [14] Lin. “A survey of application research based on blockchain smart contract”. In: (2022), pp. 635–690. DOI: [10.1007/s11276-021-02874-x](https://doi.org/10.1007/s11276-021-02874-x). URL: <https://doi.org/10.1007/s11276-021-02874-x>.
- [15] Loi Luu et al. “Making Smart Contracts Smarter”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: Association for Computing Machinery, 2016, pp. 254–269. ISBN: 9781450341394. DOI: [10.1145/2976749.2978309](https://doi.org/10.1145/2976749.2978309). URL: <https://doi.org/10.1145/2976749.2978309>.
- [16] Mark Mossberg et al. *Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts*. 2019. arXiv: [1907.03890](https://arxiv.org/abs/1907.03890) [cs.SE].
- [17] OpenAI. *chat GPT*. Access in 2023. URL: <https://openai.com/chatgpt>.
- [18] Remix. *Remix IDE*. Access in 2023. URL: <https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.22+commit.4fc1097e.js>.
- [19] SmartContractSecurity. *Smart Contract Weakness Classification*. Access in 2023. URL: <https://swcregistry.io>.
- [20] Wesley Joon-Wie Tann et al. *Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Security Threats*. 2019. arXiv: [1811.06632](https://arxiv.org/abs/1811.06632) [cs.CR].

- [21] Sergei Tikhomirov et al. “SmartCheck: Static Analysis of Ethereum Smart Contracts”. In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. WETSEB '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 9–16. ISBN: 9781450357265. DOI: [10.1145/3194113.3194115](https://doi.org/10.1145/3194113.3194115). URL: <https://doi.org/10.1145/3194113.3194115>.
- [22] Christof Ferreira Torres, Mathis Steichen, and Radu State. *The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts*. 2019. arXiv: [1902.06976](https://arxiv.org/abs/1902.06976) [cs.CR].
- [23] Jeffrey Wilcke. *To fork or not to fork*. Access in 2023. URL: <https://blog.ethereum.org/2016/07/15/to-fork-or-not-to-fork>.
- [24] Chavhan Sujeet Yashavant, Saurabh Kumar, and Amey Karkare. *ScrawlD: A Dataset of Real World Ethereum Smart Contracts Labelled with Vulnerabilities*. 2022. arXiv: [2202.11409](https://arxiv.org/abs/2202.11409) [cs.CR].
- [25] Jiaming Ye et al. “Vulpedia: Detecting Vulnerable Ethereum Smart Contracts via Abstracted Vulnerability Signatures”. In: *J. Syst. Softw.* 192.C (Oct. 2022). ISSN: 0164-1212. DOI: [10.1016/j.jss.2022.111410](https://doi.org/10.1016/j.jss.2022.111410). URL: <https://doi.org/10.1016/j.jss.2022.111410>.
- [26] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. “A Framework and DataSet for Bugs in Ethereum Smart Contracts”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2020, pp. 139–150. DOI: [10.1109/ICSME46990.2020.00023](https://doi.org/10.1109/ICSME46990.2020.00023).
- [27] Zibin Zheng et al. *DAppSCAN: Building Large-Scale Datasets for Smart Contract Weaknesses in DApp Projects*. 2023. arXiv: [2305.08456](https://arxiv.org/abs/2305.08456) [cs.SE].
- [28] Haozhe Zhou, Amin Milani Fard, and Adetokunbo Makanju. “The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support”. In: *Journal of Cybersecurity and Privacy* 2.2 (2022), pp. 358–378. ISSN: 2624-800X. DOI: [10.3390/jcp2020019](https://www.mdpi.com/2624-800X/2/2/19). URL: <https://www.mdpi.com/2624-800X/2/2/19>.

Annex

ANNEX A – RemixIDE Interface and Capabilities

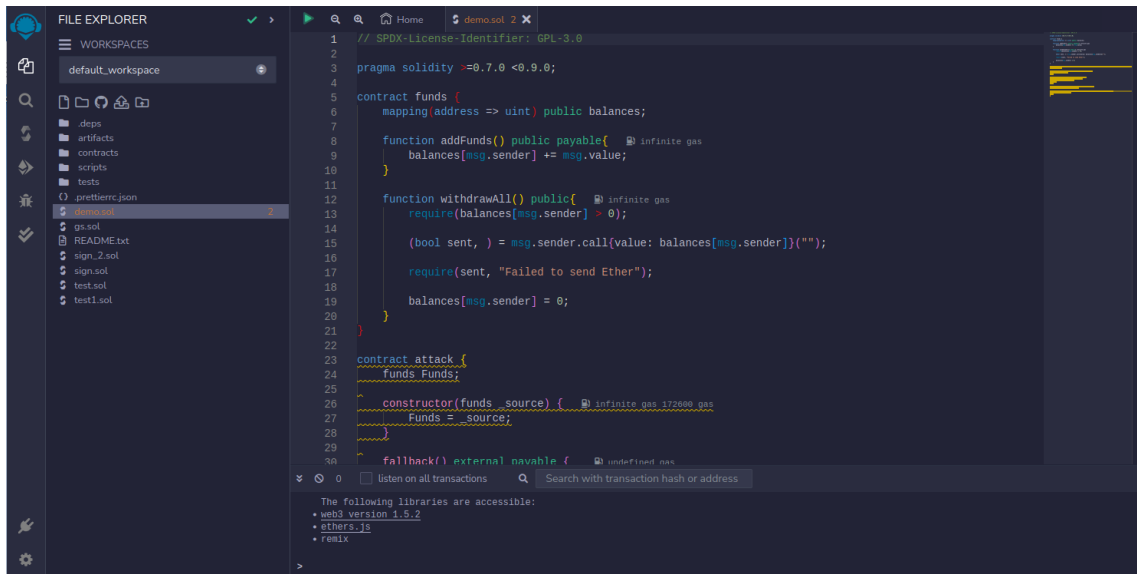


Figure 26 – Example of the RemixIDE developer interface

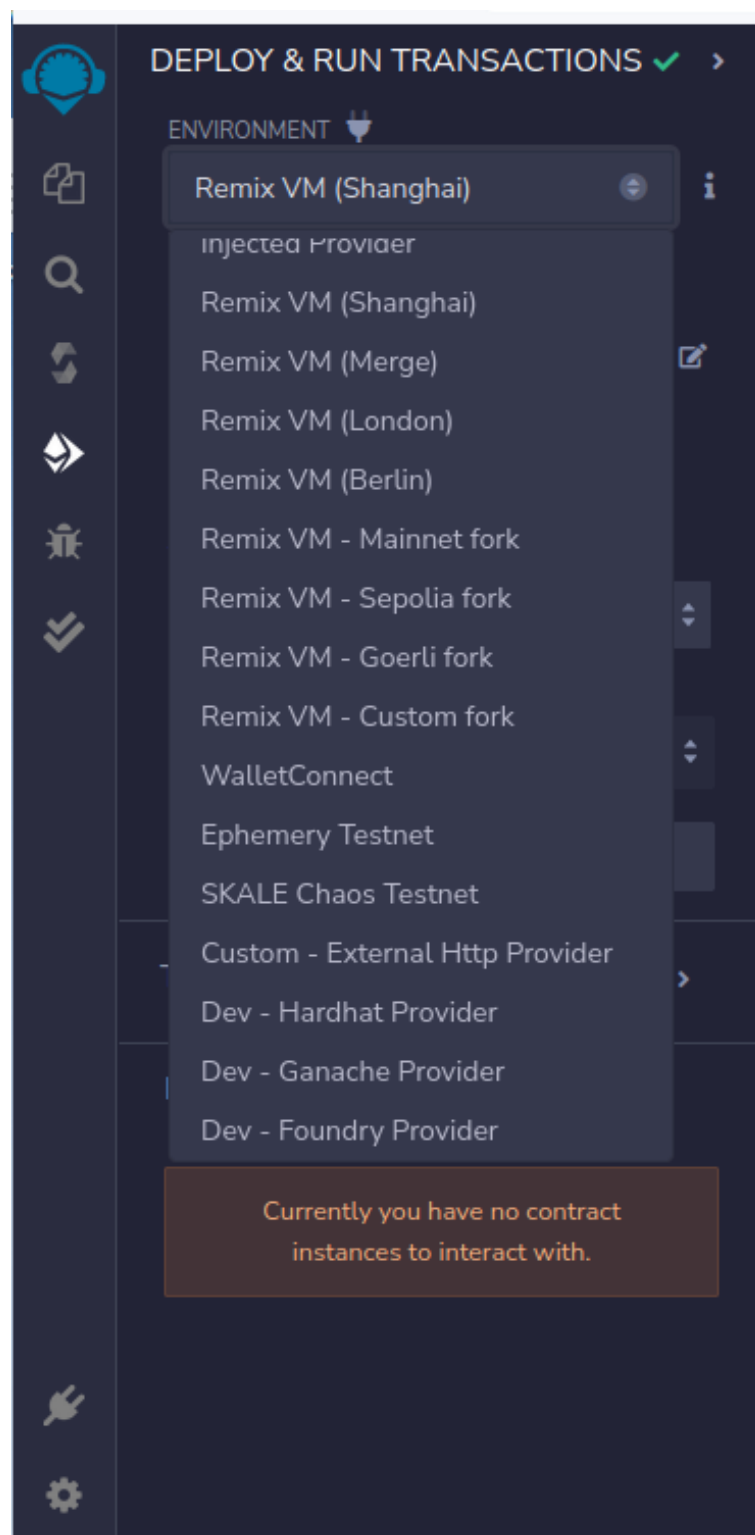


Figure 27 – RemixIDE Ethereum Execution Environments

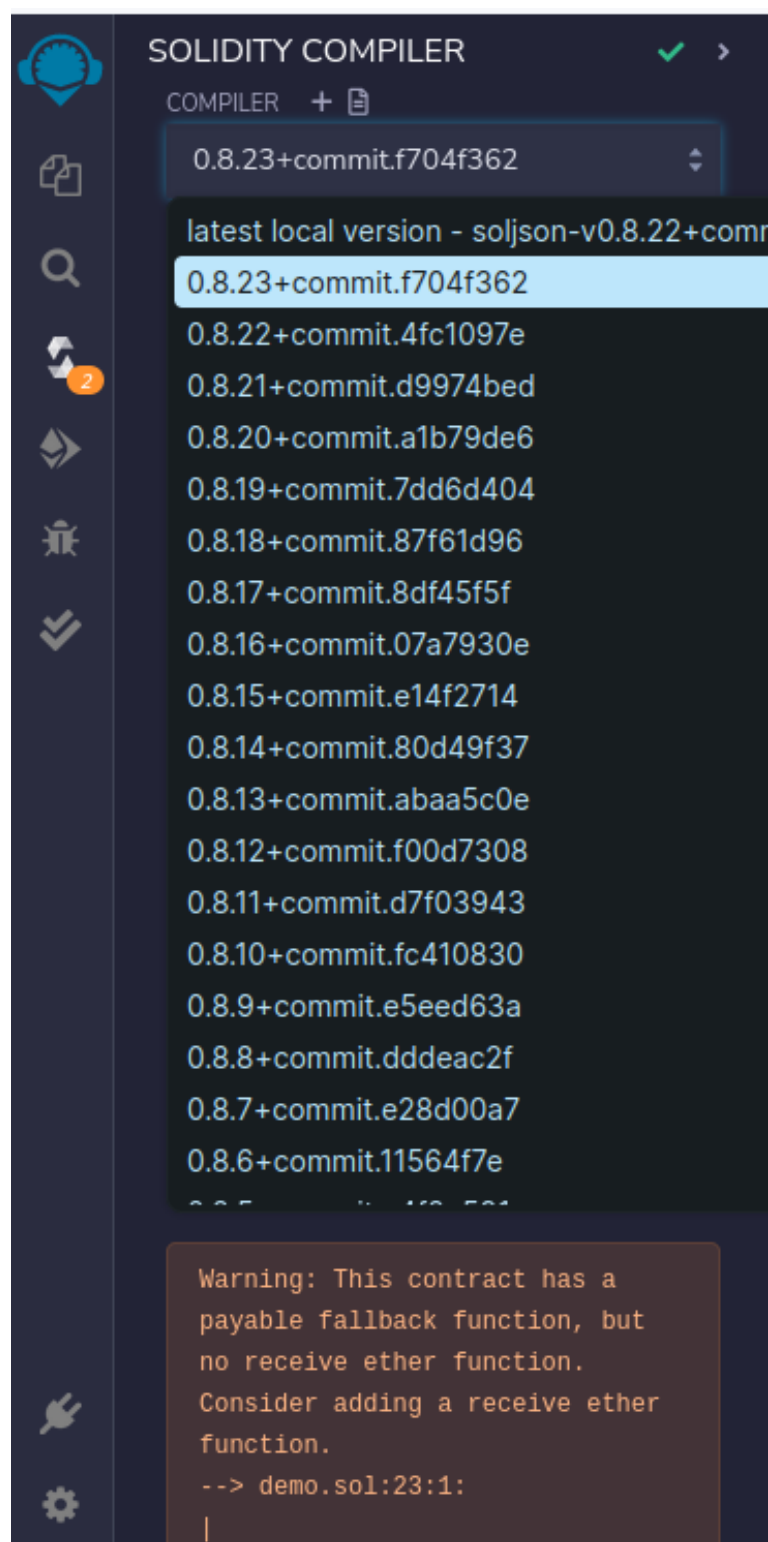


Figure 28 – RemixIDE Supported Compiler Versions (Incomplete) Sample

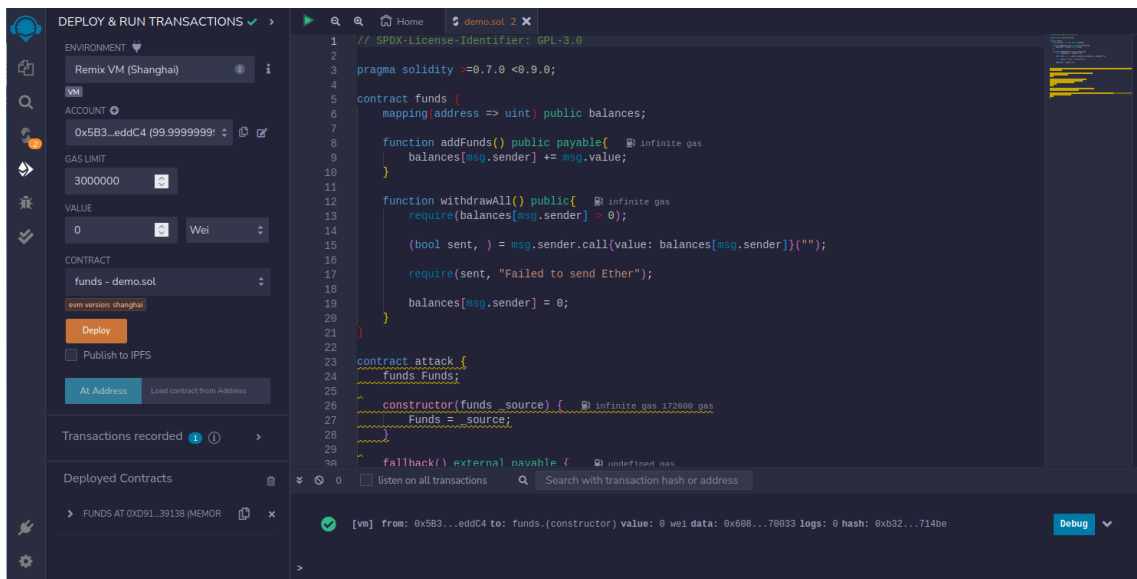


Figure 29 – Example the Deployment of a SC in the Shanghai Test Environment

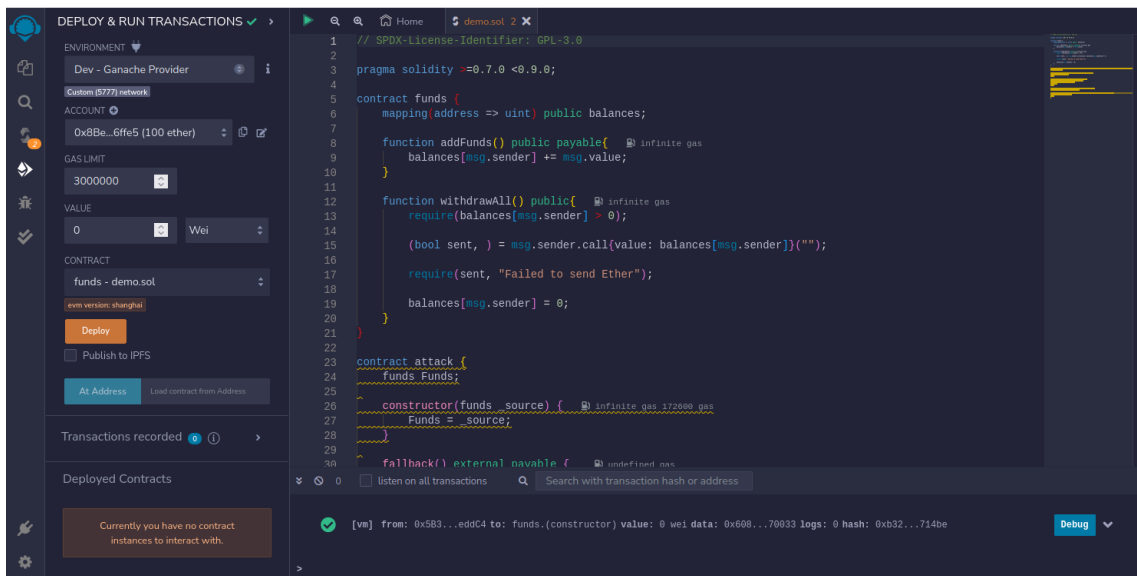


Figure 30 – RemixIDE Connected to a Locally Generated Ganache Testnet

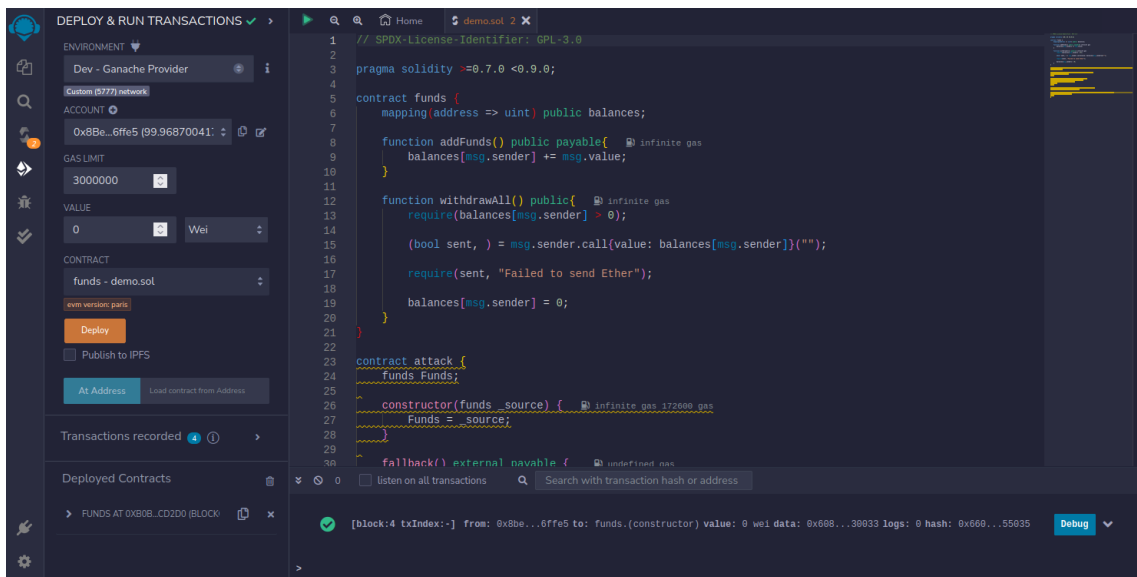


Figure 31 – Example the Deployment of a SC in the Local Ganache Testnet

ANNEX B – Ganache Interface and Capabilities

ACCOUNTS | BLOCKS | TRANSACTIONS | CONTRACTS | EVENTS | LOGS

SEARCH FOR BLOCK NUMBERS OR TX HASHES

CURRENT BLOCK: 0 | GAS PRICE: 2000000000 | GAS LIMIT: 6721975 | HARDFORK MERGE | NETWORK ID: 5777 | RPC SERVER: HTTP://127.0.0.1:7545 | MINING STATUS: AUTOMINING | WORKSPACE: SOLID-CLOCKS | SWITCH | Settings

MNEMONIC ? **HD PATH**
 suit noble easy spawn when debris describe isolate firm extra squeeze polar
 m44'60'0'0account_index

ADDRESS	BALANCE	TX COUNT	INDEX	
0x8Be6493Daa9FD69Bdd7194cE2b44A424A856ffe5	100.00 ETH	0	0	🔗
0x4a02f682D6066d0C8C4F8447233d0E1c0F619b3	100.00 ETH	0	1	🔗
0x8984cF1f3D86460f3587468FC48F797eAE456436	100.00 ETH	0	2	🔗
0xC25912BFEF042E97F4c086437A4b7e910126dc40	100.00 ETH	0	3	🔗
0xE40431243efaf7d545b875AD922f9a7b720Ba4cC	100.00 ETH	0	4	🔗
0x3C34d5B55633DFEF6BdB8E0c21bb3d4508A965D	100.00 ETH	0	5	🔗
0x5E667dF477e2A73f3e50d7eD6cd7D0aBAc5820e6	100.00 ETH	0	6	🔗
0xa27F207120885615bf9Ca0554a52526fB5AF00bf	100.00 ETH	0	7	🔗

Figure 32 – Example of Locally Created Testnet Accounts

CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE	SWITCH	SETTINGS
4	2000000000	6721975	MERGE	5777	HTTP://127.0.0.1:7545	AUTOMINING	SOLID-CLOCKS		
BLOCK 4	MINED ON 2023-11-23 17:46:57		GAS USED 308313		1 TRANSACTION				
BLOCK 3	MINED ON 2023-11-23 17:45:51		GAS USED 3000000		1 TRANSACTION				
BLOCK 2	MINED ON 2023-11-23 17:44:27		GAS USED 3000000		1 TRANSACTION				
BLOCK 1	MINED ON 2023-11-23 17:43:34		GAS USED 3000000		1 TRANSACTION				
BLOCK 0	MINED ON 2023-11-23 17:42:07		GAS USED 0		NO TRANSACTIONS				

Figure 33 – Example of Blocks Generated by Interacting with the Ganache Testnet

ACCOUNTS	BLOCKS	TRANSACTIONS	CONTRACTS	EVENTS	LOGS	SEARCH FOR BLOCK NUMBERS OR TX HASHES			
CURRENT BLOCK	GAS PRICE	GAS LIMIT	HARDFORK	NETWORK ID	RPC SERVER	MINING STATUS	WORKSPACE	SWITCH	SETTINGS
4	2000000000	6721975	MERGE	5777	HTTP://127.0.0.1:7545	AUTOMINING	SOLID-CLOCKS		
BLOCK 4									
GAS USED	GAS LIMIT	MINED ON	BLOCK HASH						
308313	6721975	2023-11-23 17:46:57	0x660dcd3e20c4620d99b3e639f6a5b5e4082d1941b767d9956b3becc235355035						
TX HASH									
0x68b852938bb4629d617b78711fe8799cd720926bd93148583bb1fb9a877c738c CONTRACT CREATION									
FROM ADDRESS	CREATED CONTRACT ADDRESS			GAS USED	VALUE				
0x8Be6493Daa9FD698dd7194cE2b44A424A856ffe5	0xB0bE90E35E26378bfE454382dDa6A082b4ecd2D0			308313	0				

Figure 34 – Example of Block Content in the Ganache Testnet

Figure 35 – Example of Transaction Content in the Ganache Testnet