INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Solidity Code Security Analysis with Generative AI

FRANCISCO JOSÉ DE SOUSA FERREIRA DA SILVA
setembro de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Solidity Code Security Analysis with Generative AI

Francisco José de Sousa Ferreira da Silva

Dissertation for a Master's Degree in Informatics Engineering,
Specialisation in Software Engineering

Supervisor: Isabel Azevedo

Porto, September 2025

Statement of Integrity

I hereby declare having that I have conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore, the work presented in this document is original and authored by me, having not previously been used for any other end. The exceptions are explicitly recognised in the section “Ethical considerations” of the first chapter. This section also states how AI tools were used and for what purpose.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, 15 September 2025.

Acknowledgement

I would like to express my sincere gratitude to my supervisor, Professor Isabel Azevedo. I am grateful for her constant support and invaluable help.

I would also like to extend my thanks to all my colleagues, especially João Campelo, Bruno Ferreira and Hugo Amaral, with whom I have had the privilege of sharing these years and who, in their own way, contributed to the achievement of this milestone.

A special thanks to Cátia for her support, patience and encouragement throughout my academic and professional life. Without her, this achievement would not have been possible.

And finally, to my family and all my friends, who, through simple gestures, contributed to making my journey easier and possible.

Abstract

Blockchain technology has revolutionised how businesses are conducted, and smart contracts are at the forefront of this change. Smart contracts are digital programs that run on a blockchain when specific conditions are met, replicating the terms of real-world agreements with greater efficiency and lower costs. The Ethereum platform is the most popular for developing smart contracts, owing to its decentralised Turing-complete machine, known as the Ethereum Virtual Machine (EVM), which can execute scripts across a global network of public compute nodes. Despite this being a powerful tool, smart contracts can still be vulnerable to hacking.

This study delves into the current state of security vulnerability detection for Solidity code, which is the main programming language for Ethereum smart contracts. This thesis aims to understand and evaluate whether the LLM can detect security vulnerabilities and if they are more effective than static analysis tools. The analysis is focused mainly on two of the vulnerabilities that generated the largest monetary losses in 2024: Access Control ($953.2M) and Reentrancy($35.7M).

The comparative analysis was carried out on 150 smart contracts using Slither, a static analysis tool widely used, able to detect up to 100 vulnerabilities and optimisations, and an LLM model-codellama, an open-source model pre-trained with Solidity, which is specialised in code generation and discussion.

The analysis shows that Slither is currently more mature and reliable than Codellama, achieving an accuracy of 0.95%, which means that the tool was able to identify all vulnerabilities correctly. Furthermore, when comparing Slither to Codellama, it can be concluded that Slither is much faster at performing the analysis, with an average time of 2.17 seconds per Solidity file. Although Slither is rated as more effective than Codellama, the analysis also confirms the potential of large language model-based approaches in detecting security vulnerabilities in Solidity code.

Keywords: Smart Contracts, LLM, security, vulnerabilities, Solidity

Resumo

A tecnologia *blockchain* revolucionou a forma como os negócios são conduzidos, e os contratos inteligentes estão na vanguarda dessa mudança. Os contratos inteligentes são programas digitais que são executados numa blockchain quando determinadas condições são cumpridas, replicando os termos dos contratos do mundo real com maior eficiência e menores custos. A plataforma *Ethereum* é a mais popular para o desenvolvimento de contratos inteligentes, devido à sua máquina descentralizada *Turing complete*, conhecida como *Ethereum Virtual Machine* (EVM), que pode executar scripts numa rede global de nodes de computação públicos. Apesar de esta ser uma ferramenta poderosa, os contratos inteligentes ainda podem ser vulneráveis a hackers.

O presente estudo pretende investigar o atual estado da deteção de vulnerabilidades de segurança para o código *Solidity*, que é a principal linguagem de programação para contratos inteligentes em *Ethereum*. Esta tese tem como objetivo compreender e avaliar se o LLM pode detetar vulnerabilidades de segurança e se estas são mais eficazes do que as ferramentas de análise estática. A análise centra-se principalmente em duas das vulnerabilidades que geraram as maiores perdas monetárias em 2024: controlo de acessos (953,2 milhões de dólares) e reentrância (35,7 milhões de dólares).

A análise comparativa foi realizada em 150 contratos inteligentes usando o *Slither*, uma ferramenta de análise estática amplamente utilizada, capaz de detetar até 100 vulnerabilidades e otimizações, e um modelo LLM - codellama, modelo de código aberto pré-treinado com Solidity, especializado em geração e discussão de código.

A análise mostra que o *Slither* é atualmente mais maduro e confiável do que o *Codellama*, alcançando uma precisão de 0,95%, o que significa que a ferramenta foi capaz de identificar todas as vulnerabilidades corretamente. Além disso, ao comparar o Slither com o Codellama, foi possível concluir que o *Slither* é muito mais rápido na realização da análise, com um tempo médio de 2,17 segundos por ficheiro Solidity. Embora o *Slither* seja classificado como mais eficaz do que o Codellama, a análise também confirma o potencial das abordagens baseadas em modelos de linguagem de grande porte na deteção de vulnerabilidades de segurança no código Solidity.

Palavras-chave: contratos inteligentes, LLM, segurança, vulnerabilidades, Solidity

Table of Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

AI	Artificial Intelligence
AGI	Artificial General Intelligence
ANI	Artificial Narrow Intelligence
API	Application Protocol Interface
ASI	Artificial Super Intelligence
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CPU	Central Processing Unit
CSV	Comma-Separated Values
DAO	Decentralized Autonomous Organizations
EVM	Ethereum Virtual Machine
FP	False Positives
FN	False Negatives
IEEE	Institute of Electrical and Electronics Engineers
LLM	Large Language Model
NLP	Natural language Process
PC	Program Counter
PICOC	Population, Intervention, Comparison, Outcome, and Context
SCs	Smart Contracts
SLR	Systematic Literature Review
SQL	Structured query language
SWC	Smart Contract Weakness Classification
TDD	Test Driven Design
TLD	Test-Last Development
TP	True Positives

TN	True Negatives
UI	User interface

1 Introduction

In the introduction chapter, besides the problem description and the definition of its context, the main objectives will be described. Besides that, the structure of the document is presented, as well as the ethical considerations for this work.

1.1 Context

Blockchain is a technology introduced by Satoshi Nakamoto in 2008 [1]. This technology consists of a chain of blocks of economic transactions that can be programmed to record virtually everything of value. Although this technology is usually associated with the financial sector, it can be used in several areas, such as healthcare, property records, insurance and energy.

Smart Contracts (SCs), one of the main blockchain technologies, are simply programs stored on a blockchain that run when predetermined conditions are met [2]. This technology aims to replicate the terms of real-world contracts in the digital domain. Comparing them to conventional contracts, SCs offer the benefits of reducing transaction risk, administration and service costs and enhancing the efficiency of corporate processes [3].

This technology dates to the 1990s, but its use has become more popular with the introduction of platforms like Bitcoin and Ethereum in 2008 and 2013, respectively. Ethereum, as the most used platform nowadays, differs from the others because it allows the creation of arbitrary contracts for any type of transaction or application through the programming language Solidity.

1.2 Problem

Although smart contracts offer many benefits, this technology needs extra attention in terms of safety and security, because it needs to handle a variety of risks, including incorrect code, malicious inputs and attacks on the blockchain network, to avoid losing money and their trust [3]. Some hacking incidents have resulted in significant financial loss cases, for instance, Decentralized Autonomous Organizations (DAO), a crowdfunding effort, was hacked in 2016 because of a security vulnerability, and more than 60 million dollars were stolen [4]; Parity, a

prominent Ethereum wallet, was hacked in 2017 and more than 30 million dollars were stolen due to a vulnerability in an SC library, and, one of the most famous cases, Binance Smart Chain Exploits, which happened in 2021, resulting in a 200 million dollar loss because of a vulnerability detected on SC [6].

However, even if programmers are encouraged to write secure smart contracts, supportive tools must be available to help them to detect and mitigate possible security vulnerabilities in order to avoid episodes like the ones previously presented . To reduce the number of hacking related to SCs many approaches have been proposed [8]. Artificial Intelligence (AI) tools such as ChatGPT and Google's Bard (recently renamed as Gemini) have recently begun to be used[11], nevertheless, the available studies are scarce.

1.3 Objectives

The main objective behind the current work is to evaluate the effectiveness and compare the usage of Large Language Models (LLMS) and static analysis tools to detect security vulnerabilities in Solidity code. To achieve that, two major steps are needed. The first step consists of the systematic literature review methodology, where the knowledge is acquired and documented. In the second step, using the knowledge obtained, a comparative analysis will be conducted between the static analysis tools and the LLMS. Thus, the main objective of this work is to answer the following questions:

- Are LLMS able to detect security vulnerabilities in Solidity code?
- Are the LLMS more effective than the static analysis tools to detect security vulnerabilities in Solidity code?

Given the vast number of tools available in this field, the analysis cannot be carried out for all available tools, so the answer to these questions will be based solely on the selected tools presented in chapter 4.2.

1.4 Document Structure

This document is organised into six main chapters, and the main purpose of this chapter is to explain how to dive into this work.

- Chapter 1 'Introduction' - In the introduction chapter it will be given the context of the work, including the problem description and the main objectives. This chapter ends with the ethical considerations taken.
- Chapter 2 'Background' - The purpose of the background chapter is to introduce the most relevant topics for the current work. Topics like blockchain, Ethereum, Smart Contracts, Solidity, Static and Dynamic Analysis Tools and Artificial Intelligence will be properly addressed.
- Chapter 3 'State of the art'- The main goal of the state of the art is to explore the current knowledge in the subjects related to this work. All details needed to conduct the

literature review, such as inclusion and exclusion criteria, keywords and the engines, are also documented in this chapter.

- Chapter 4 'Design' - Once all the knowledge related to the aim of this work has been acquired, the design chapter will describe all the decisions that were taken into consideration for the comparison analysis. Besides presenting the tools to be used, it also shows the dataset, the prompt design strategy, and the solution architecture. It ends with a presentation of the performance metrics that will be evaluated.
- Chapter 5 'Implementation' – The chapter implementation will focus on explaining all the technical details of the analysis. This chapter is divided into three main phases: the exploratory phase, which describes how the tools were installed and presents the environment used for the analysis. The second phase, development, addresses all the technical details considered for implementation. The last phase is the testing phase, which is presented along with a small example. At the end of the chapter, the results are presented and explained how they were extracted and classified.
- Chapter 6 'Conclusion' – The last chapter, Conclusion, will describe all the achievements made and some proposals for future work will be presented.

1.5 Ethical Considerations

When conducting research in the software engineering field, ethical considerations and a code of conduct are essential. The standards that will be followed in this paper are the Institute of Electrical and Electronics Engineers (IEEE) Code of Ethics [12] and the IEEE Author Ethics Guidelines [13].

Based on the guidelines provided by these two standards, the development of this project aims to make a significant intellectual contribution with original research that has not been previously published and is not currently submitted elsewhere. Following this approach, all information presented in the report is properly cited, according to the IEEE standards, with the commitment of not committing plagiarism.

Concerning the use of artificial intelligence, given that the work to be carried out is related to that topic, it will be used judiciously only for tasks included in the study itself, excluding all functions related to text generation that would assist in the production of the document.

2 Background

The following chapter aims to introduce the fundamentals of blockchain, providing details of the technology itself and some related topics - such as Ethereum and Smart Contracts -, focusing on some key aspects of Solidity, which is the most used programming language to develop smart contracts. Besides that, the most common vulnerabilities in smart contracts and the concept of Static and Dynamic Analysis Tools are presented. This chapter also addresses Artificial Intelligence, mainly focused on Large Language Model and Natural Language Processing.

2.1 Blockchain

The term *blockchain* was initially introduced by Satoshi Nakamoto in 2008 through a white paper called “Bitcoin: A Peer-to-Peer Electronic Cash System" [1]. Despite being a widely used term today, there is no consensus on the definition of blockchain technology. One of the most popular definitions was developed by Don and Alex and describes blockchain as “an incorruptible digital ledger of economic transactions that can be programmed to record not just financial transactions but virtually everything of value”[14].

Records in this technology are grouped to form blocks. These blocks are cryptographically protected, and their content is hashed and added to the block header. Then, the blocks are chained in a way that the cryptographically secured hash value of the previous block is linked with the next block. Thus, each block not only depends on its own data content, but also on the previous block’s hash value, as illustrated in Figure 1.

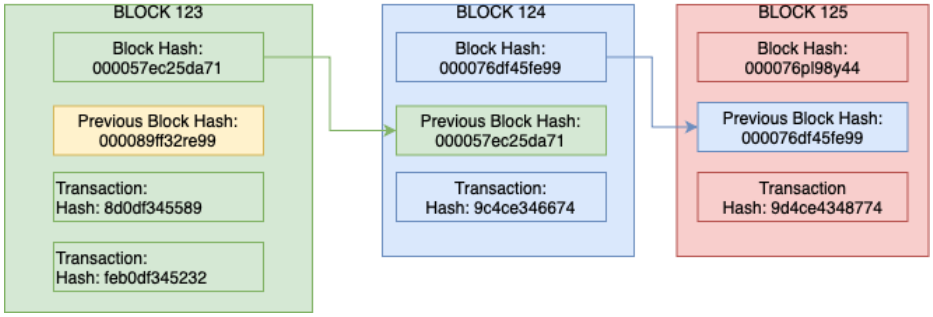

Figure 1 - The sequence of the blocks in Blockchain.

The number of transactions performed is directly related to the block size: the larger the block size, the more transactions are performed, but there is an increase in the security threat [15].

A blockchain can be classified into two categories: permission-less blockchain and permissioned blockchain. In the permission-less blockchain, also called public blockchain, anyone can access the chain's network whenever they want; it is open source, and the transactions are publicly visible, meaning that everyone can see the transaction details; Bitcoin and Ethereum are examples of public blockchains. In the case of permissioned blockchain, only identifiable and authorised nodes can access the network by providing an additional security system since the identity of each node is maintained. A node is the user or the highly configured computer that played a key role in the transactions involved in the blockchain [15].

There are three types of blockchains, which are categorised as permissioned blockchains [15]:

- Private Blockchain
  A private blockchain is a restrictive blockchain applicable to smaller organisations, where only a small number of nodes have access. Only those who have access to the private blockchain can see the transactions. There are many examples of this type of blockchain, such as HydraChain and Hyperledger Fabric.

- Consortium or federated blockchain
  A consortium blockchain can be managed by more than one organisation. Only members who belong to one of the organisations that manage the network can see the transactions. This blockchain combines both private and public blockchains; while the important information is made private, the other information remains public. Corda R3 and Quorum are examples of this type of blockchain.

- Hybrid Blockchain
  The hybrid blockchain combines private and public blockchains, incorporating the best features of each one. It is very similar to consortium blockchain, but it allows users to control the data that they want to be public, keeping the remaining data restricted. One example of this type of blockchain is Komodo.

2.1.1 Ethereum

Ethereum, a public blockchain introduced by Vitalik Buterin's paper [16] in 2013. Ethereum is a public decentralised platform with an embedded Turing complete system where "arbitrary contracts can be created for any type of transaction or application" [16]. This tool allows users to create several systems by writing up the logic in a few lines of code without requiring a trusted entity to execute them.

The following subsections will describe the fundamentals behind Ethereum.

Turing Complete

Turing Complete machine was introduced by Alan Turing in 1936 as a mathematical model of computation capable of solving any kind of computational problem. Every system can be categorised as Turing Complete if it allows any computation to resolve a problem given enough time and memory, even if it would take several million years [17].

Ethereum allows developers to write code using the Solidity programming language and execute it using the Ethereum Virtual Machine, which is a Turing complete machine.

Ethereum Virtual Machine

Ethereum Virtual Machine (EVM) parses the contract source code into binary code to execute it and process the transactions properly. All smart contracts written in this platform are compiled to the same format, which is Ethereum bytecode, and executed by the EVM [19]. The architecture of EVM is illustrated in Figure 2.

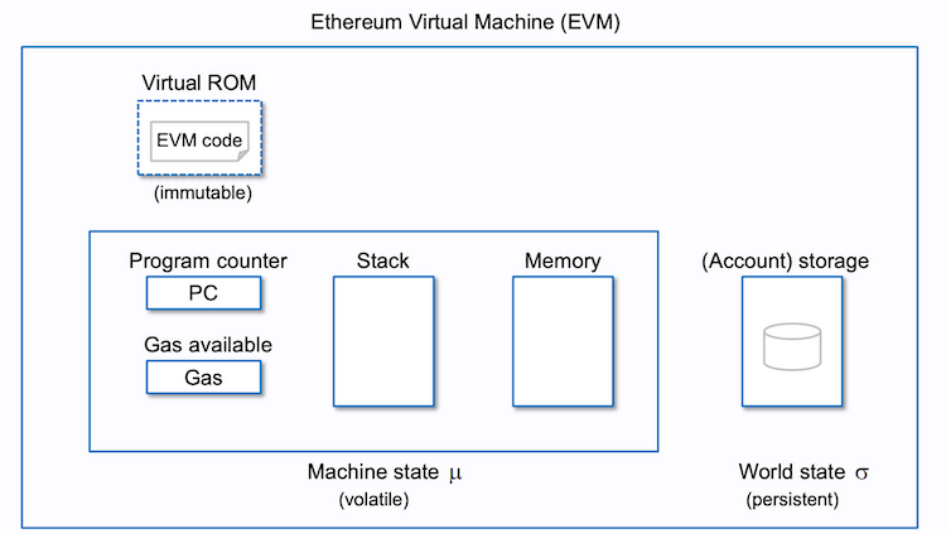

Figure 2 - Ethereum Virtual Machine Architecture. From [20].

When the code is executed or transactions are made in EVM, an amount of Gas and Program Counter is calculated. Gas is a unit that measures the computational effort required to execute

operations. Program Counter (PC) tracks the EVM instruction to be executed, ensuring that the operations are done in a correct sequence.

If the amount of gas is sufficient to execute the instruction, it is stored in memory, and the operations are executed and stored in the EVM stack, and the Program Counter is incremented.

2.1.2 Smart Contracts

Although smart contracts are a concept independent of blockchain, both are strongly connected since smart contracts are only viable because of blockchain technology. Smart Contracts (SCs) were initially introduced by Nick Szabo in the mid-1990s; at that time, SCs emerged as the translation of clauses of a contract into code and embedded them into software or hardware to make them self-execute, to minimise contracting costs between transacting parties and to avoid accidental exceptions or malicious actions during contract performance [19].

Nowadays, the term 'smart contract' is used in diverse ways: "Some referred 'smart contract' as a legal contract which (or at least elements of which) could be represented by software, while some others took 'smart contract' as code scripts which are designed to execute certain tasks once pre-defined conditions are met" [19]. Simplifying, smart contracts are programs stored in the blockchain that are executed automatically once the preconditions are met. One of the main characteristics of these programs is that, once deployed, their code is immutable, meaning that it is no longer suitable for changes.

Smart contracts in Ethereum can be written in several programming languages. Only Solidity, the most used programming language to develop smart contracts, will be analysed in this work. Figure 3 shows the percentage of smart contracts developed in each language.

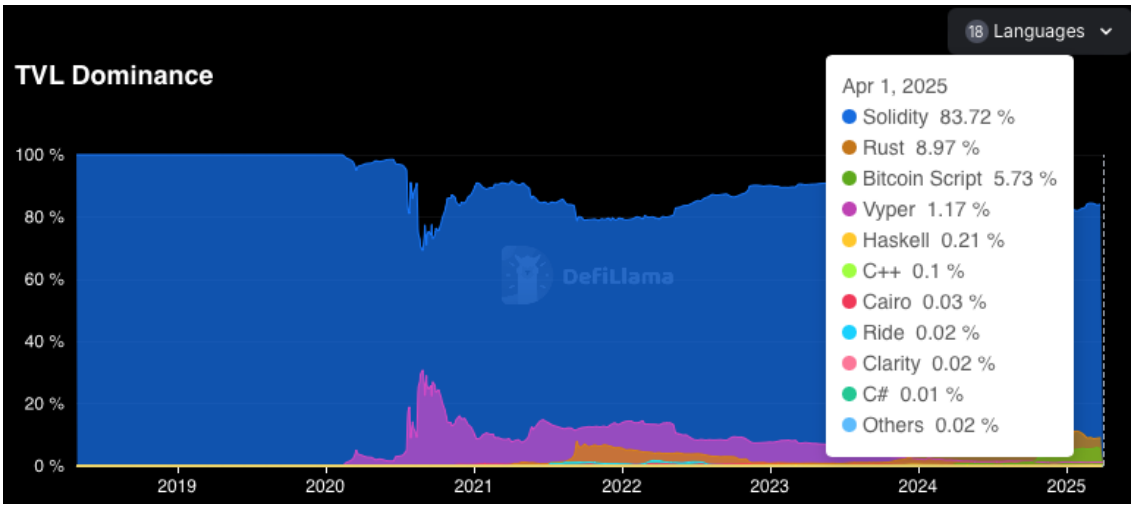

Figure 3 - Most used programming languages to develop Smart contracts. From [21].

From the analysis of Figure 3, it can be seen that, besides Solidity, other languages like RUST [22] and Vyper [23] can also be used to develop smart contracts in Ethereum.

Solidity

Solidity [24] was created by Dr Gravin Wood, and it is defined as a high-level contract-oriented language with similarities to JavaScript and C languages. It allows anyone to develop contracts and compile to EVM bytecode [4]. At the time of writing, Solidity is at version 0.8.30. When deploying contracts, the latest version must be taken, since only the latest version receives the security fixes.

Solidity can be used to create contracts for several uses, such as voting, crowdfunding, blind auctions, and multi-signature wallets. The most notable features of Solidity are that it is statically typed, supports inheritance, libraries, and complex user-defined types, among other features.

Each Solidity program can contain an arbitrary number of contract definitions, imports, pragma, enum, functions, and constant variable definitions. The file extension used in Solidity files is *.sol.*

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.6.0 <0.9.0;
3
4 contract HelloWorld {
5    function helloWorld() external pure returns (string memory) {
6        return "Hello, World!";
```

Source Code 1 - Example of a contract in Solidity.

The main idea behind Source Code 1 is to replicate the typical example of printing a “Hello World”. The code begins with the definition of SPDX License Identifier (line 1). Although this is optional, it is highly recommended to indicate the license to understand whether the source code is open-source or not.

Pragma (line 2) is used to enable certain compiler features or checks. The version in which the smart contract can be executed, specified by this keyword, must be limited to ensure that the code behaves in the way intended by the developer. After that, a smart contract (line 4-8) is defined whose purpose, when called, is to return "Hello World!".

The main elements of a Solidity contract [25] will be described in the sections below.

State Variables and Data Types

State variables are variables whose values are either permanently stored in contract storage or temporarily stored and cleaned at the end of each transaction. In Solidity, the concept of “undefined” or “null” does not exist, but each variable always has a default value depending on the data type, and each variable has one data type associated. The most used data types are the following:

- *Boolean (bool):* can store Boolean values as *true* or *false.* This data type supports all binary operators.
- *Integer (uint, int):* Supports all signed and unsigned integers from 8 up to 256 bits with steps of 8 *(uint8, unit16, etc)*.
- *Fixed point numbers (fixed, ufixed):* Signed and Unsigned fixed point numbers defined by ufixedMxN and fixedMxN, where M represents the number of bits taken by the type, and N represents how many decimal points are available.
- *Address:* This data type stores an Ethereum account address. If the idea behind this address is to be used to transfer ether, then it must be declared explicitly as an *address payable*.
- *Fixed- sized byte arrays (bytes)* - Stores a sequence of bytes from 1 up to 32. It is defined by the keyword *bytesx* where x represents the number of bytes.
- *Enum:* can store enumerated values. The first value defined is considered as the default value.

Solidity has a wide range of data types that can be easily combined to generate more complex data types.

Access Modifiers

The modifiers in Solidity are used to control access to functions or state variables within the smart contracts. They determine who can call a particular function or access a state variable. There are four different modifiers:

- *Public*: It is the least restrictive modifier. State variables and functions can be easily accessed by everyone.
- *Private*: It is the most restrictive modifier. This modifier ensures that state variables and functions can only be accessed within the smart contracts.
- *External*: Using this modifier, the state variables and functions can only be accessed from outside of the smart contract where it is declared.
- *Internal*: With this modifier, functions and state variables can be accessed within the smart contract and in those that derive from it.

Functions, Modifiers and Fallbacks

Functions

Functions can be defined inside or outside the contracts. Independent of both being executed in the context of the contract, the main difference between them is that the functions that are defined inside the contract, called the free functions, have access to the variable ***this,*** and the others do not.

Modifiers

Modifiers are functions that can be used to change the behaviour of the function. The modifiers, in some cases, are like rules that are defined under the main contracts, and when that rule is

not respected, an exception must be thrown. Source Code 2 illustrates an example of a modifier where it is defined that if the owner of the contract calls this function, it can be executed; otherwise, it will end the execution with an exception.

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.0

contract Owner {
    constructor (){ owner = payable(msg.sender); }
    address payable owner;

    modifier onlyOwner {
      require(
         msg.sender == owner,
         "Only owner can call this function."
      );
```

Source Code 2 - Example of a modifier in Solidity. From [25].

Fallback

The *fallback* function is called once the contract is called. Typically, this function does not receive any argument and does not have any return type. In the Source Code 3 can be seen an example of a fallback in Solidity.

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.0

contract Beta {
    function () external{ x=1; }
     unit x;
```

Source Code 3 - Example of a fallback in Solidity.

Events

The *events* in Ethereum can be seen as a mechanism for emitting and recording data. They are stored in the transaction log with the information of the contract that emitted them and can be accessed only outside the contract.

In Solidity, it is also possible to define events using the keyword *event,* preceded by an identifier and the number of parameters desired. These events can then be filtered and organised by their parameters.

Security Vulnerabilities for Smart Contracts

As previously mentioned, smart contracts can be used in various areas, such as finance, healthcare and crowdfunding, which are considered critical areas as they usually handle a lot of money and/or data. Due to this, this technology is often the target of attacks to steal data/money. After some research, it was possible to find the most common vulnerabilities defined by the Smart Contract Weakness Classification (SWC) [26]:

- SWC-100 - Function Default Visibility
- SWC-101 - Integer Overflow and Underflow
- SWC-102 - Outdated Compiler Version
- SWC-103 - Floating Pragma
- SWC-104 - Unchecked Call Return Value
- SWC-105 - Unprotected Ether Withdrawal
- SWC-106 - Unprotected SELFDESTRUCT Instruction
- SWC-107 - Reentrancy
- SWC-108 - State Variable Default Visibility
- SWC-109 - Uninitialized Storage Pointer
- SWC-110 - Assert Violation
- SWC-111 - Use of Deprecated Solidity Functions
- SWC-112 - Delegatecall to Untrusted Callee
- SWC-113 - DoS with Failed Call
- SWC-114 - Transaction Order Dependence
- SWC-115 - Authorization through tx.origin
- SWC-116 - Block values as a proxy for time
- SWC-117 - Signature Malleability
- SWC-118 - Incorrect Constructor Name
- SWC-119 - Shadowing State Variables
- SWC-120 - Weak Sources of Randomness from Chain Attributes
- SWC-121 - Missing Protection against Signature Replay Attacks
- SWC-122 - Lack of Proper Signature Verification
- SWC-123 - Requirement Violation
- SWC-124 - Write to Arbitrary Storage Location
- SWC-125 - Incorrect Inheritance Order
- SWC-126 - Insufficient Gas Griefing
- SWC-127 - Arbitrary Jump with Function Type Variable
- SWC-128 - DoS With Block Gas Limit
- SWC-129 - Typographical Error
- SWC-130 - Right-To-Left-Override control character (U+202E)
- SWC-131 - Presence of unused variables
- SWC-132 - Unexpected Ether balance
- SWC-133 - Hash Collisions With Multiple Variable Length Arguments
- SWC-134 - Message call with hardcoded gas amount
- SWC-135 - Code With No Effects
- SWC-136 - Unencrypted Private Data On-Chain

The description of these vulnerabilities, as well as the way to prevent them, can be analysed in detail in Annex A. Some of these vulnerabilities are no longer present in the latest versions of Solidity due to updates that have been made to the language.

2.2 Static and Dynamic Analysis Tools

The most traditional way to detect the security vulnerabilities previously presented is through Static or Dynamic Analysis Tools.

Static Code Analysis tools allow the code to be analysed without executing it. These kinds of tools can be used to detect possible defects and irregularities. Besides that, these tools can also be used to help developers understand the behaviour of a program [27].

By comparing static code analysis to conventional testing, it can be concluded that the code review using the tool is also much faster and more efficient [28]. One of the main advantages of these tools is that they analyse code based on rules or patterns. These rules/patterns are usually defined with the most common defects/issues to allow the tool to detect them easily. Poor definition or lack of definition of these rules can lead to problems not being detected.

These tools were usually dedicated to a single programming language, but nowadays there are some tools where it is possible to work with several languages. Typically, this adaptation can be done easily through the configuration files.

Although static analysis is typically quicker and less resource-intensive than dynamic analysis, it might not be able to identify some problems that can only be seen in real-world situations. Dynamic analysis can give a more thorough perspective of how the code behaves, but it can take more time and may need a specific tool to gather and analyse/extract the logs [29]. These tools are generally used to detect problems such as memory leaks, race conditions, null and undefined behaviour, as well as security issues like buffer overflows, SQL injection.

Both Static and Dynamic Analysis have different benefits and drawbacks, and depending on the type of code being analysed and the study's objectives, the type of analysis tool (Static or Dynamic) must be carefully selected. Table 1 represents the outcome of the study [29], which highlights the main differences between these two types of tools.

Comparison Factors	Static Code Analysis	Dynamic Code Analysis
Timing of Analysis	Performed in source code without executing it	Performed during runtime by executing the code
Detection of Issues	Can detect issues only by examining the code, such as syntax errors, unused code, and dead code	Can detect issues that occur during program execution, such as memory leaks, race conditions, and crashes
Coverage	Can analyse the entire codebase	Limited to the execution of code that occurs during the runtime environment
Types of Issues	Best suited for identifying coding standards, security vulnerabilities, and software defects	Best suited for finding issues that arise during program execution, such as runtime errors and performance bottlenecks
Feedback Time	Can provide immediate feedback as part of the development process	Feedback may come after code has been deployed to production environments
Resource Requirements	Requires less resources, such as memory and CPU, than dynamic code analysis	Requires the execution of code, which may require more resources than static code analysis

Table 1 - Comparison between Static Code Analysis and Dynamic Code Analysis. From[29].

2.3 Artificial Intelligence (AI)

Artificial intelligence (AI) is a term that is commonly used to "describe computer systems dedicated to performing tasks close to human intelligence, such as speech recognition and language translation" [30, p. 26]. According to one of the founders of AI, John McCarthy, "AI it is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable."[31].

Although this term has started to be used more consistently recently, it was mentioned around the 1950s and became more popular around the 2000s [30, p. 6]. In 2023, with the introduction of GPT, research in this field began to be more sought after and valued.

Depending on the intelligence level embedded into a robot, Artificial intelligence can be categorised into three types:

- Artificial Narrow Intelligence (ANI) - (also known as Narrow AI or Weak AI) is a type of Artificial Intelligence focused on one single narrow task. This type of AI is mostly used daily through services like Google Assistant, Google Translate, Siri, Cortana or Alexa.
- Artificial General Intelligence (AGI) - is referred to as a type of AI that is about as capable as humans. General intelligence implies an ability to acquire and apply knowledge and to reason and think in a variety of domains, not just in a single area like, say, chess, game-playing, languages, mathematics or rugby [32].
- Artificial Super Intelligence (ASI) - pretends to achieve a machine more capable than a human. The main idea behind this type of AI is to create machines capable of exceeding the performance in several areas such as arts, decision making and emotional relationships.

The intelligence level required for the current work is the ANI, more specifically, for Large Language Models (LLMS). These services are machines that use Natural Language Processing (NLP), a term introduced in section 2.3.2.

2.3.1 Large Language Model (LLM)

A Large Language Model is designed mainly to understand and generate human-readable text (also known as natural language). These models, such as GPT, PaLM, Claude, and LLaMA, can be classified as proprietary and open models.

The proprietary models are generally developed by companies that do not share their code, training data, and full architecture details, and the way to interact with these models is generally through the API. A huge benefit of these models is that the user does not need to have a strong Central Processing Unit (CPU) to use them; the provider takes care of hosting and running the model and generally has more computing available. On the other hand, the open modes share their code to run them locally, but in most cases, it requires a powerful CPU.

Figure 4 shows, chronologically, some of the models that emerged after ChatGPT. This model was used as a reference because its launch had a huge impact on the domain and led some to designate 2023 as 'The Year of Generative AI' [30].

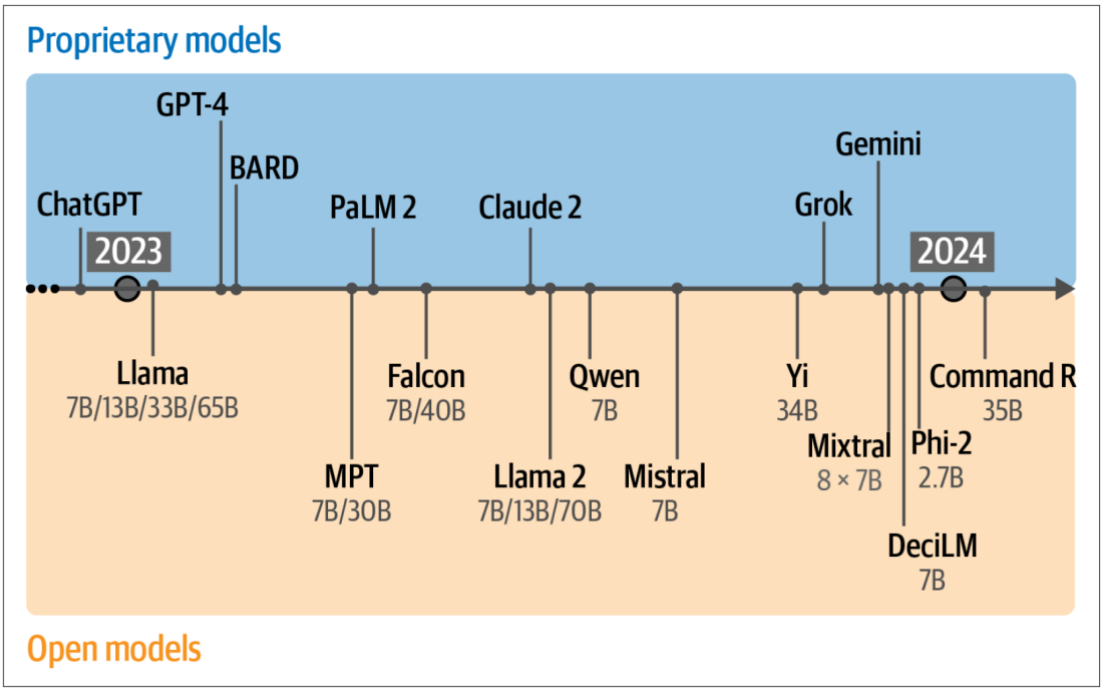

Figure 4 - The Year of Generative AI (Appearance of LLMS in chronological order). From [30].

The creation of the LLMS typically includes the following two steps:

- **Pretraining**: This step takes most of the computation and training time. An LLM is trained on a vast corpus of internet text, allowing the model to learn grammar, context, and language patterns. At the end of this step, the resulting model is often referred to as a foundation/base model.
- **Fine-Tuning:** This step can also be called post-training; it involves using the previously trained model and further training it on a narrower task. This allows the LLM to adapt to specific tasks or to exhibit desired behaviour.

LLMS can be used for a wide range of tasks such as text generation, summarisation, translation, question answering, code generation, etc. However, the creation of LLMS can also be limited by the resources available, because training it requires a lot of computational power. Since they are not constantly updated, the information they provide may even be wrong or out of date.

2.3.2 Natural Language Processing (NLP)

Natural Language Processing (NLP) is a subfield of computer science and AI that uses machine learning to enable computers to understand and communicate with human language. This language enables the computer to recognise, understand and generate text by combining

computational linguistics, rule-based modelling of human language together with statistical modelling, machine learning and deep learning [33].

The biggest benefit of using NLP is that it allows the user to interact with the computer using the language they use every day, without having to learn new forms of communication such as commands.

3 State of the Art

This chapter aims to review the literature, beginning with the review methodology, highlighting which methodology was chosen, as well as the research questions used to conduct it. The chapter concludes with the results summary by addressing the research questions.

3.1 Research Methodology

Performing a review of the literature is a crucial step to understand the state of the art and to identify possible gaps and upsets in the field. The literature review methodology adopted was the Systematic Literature Review (SLR). This type of methodology sets out a series of steps to methodically organise the review. The main objective of the SLR is to define a strategy that will enable the research questions defined in chapter 3.2 to be addressed. Once all stages have been defined, this literature review can be easily reproducible.

Based on the guidelines provided by [34], a strategy was designed to follow the SLR, divided into two main stages: planning and conducting.

To ensure that the literature review is conducted in a clear and objective manner, planning is indispensable. This planning not only defines the procedures to be followed for conducting the review, but it is also responsible for ensuring that these procedures are replicable.

The next chapter presents important aspects for the planning phase, such as the Population, Intervention, Comparison, Outcome, and Context (PICOC). framework, research questions and research process.

3.2 PICOC framework and Research Questions

Before defining the research questions, a research framework must be adopted in order to guarantee that the questions are aligned with the objectives of the investigation The framework adopted, PICOC, breaks down the SLR's objectives into searchable keywords and

helps formulate research questions [35]. Table 2 shows the application of the PICOC framework for the systematic literature review to be conducted.

Population	Single solidity developers or companies interested in developing secure code.
Intervention	Analyse if LLMs can detect security vulnerabilities in solidity code.
Comparison	Compare static code analysis tools for solidity code.
Outcome	Static code analysis tools for solidity code. LLMS used to detect security vulnerabilities for solidity code.
Context	Ethereum's Solidity programming environment.

Table 2 - Application of PICOC framework.

The population chosen for this study focuses specifically on Solidity code developers, whether they represent themselves individually or through a company. The growing interest in this area means that those involved in it are concerned with various aspects, such as the development of a more secure code.

The main focus of the research is defined in the intervention; in this case, it is intended to analyse whether LLMS can detect security vulnerabilities in Solidity code. Concerning the comparison, it will be done against the static code analysis tools. These tools will be used as a basis, since they are commonly used by the industry and have proven to be a key part of project development in this area.

The research process's outcome is the d static code analysis tools for Solidity code, and if the LLMS can detect security vulnerabilities. Finally, the context is Ethereum's Solidity programming environment, as most of the code developed is done through this ecosystem.

From the analysis of the PICOC framework, it is possible to extract several questions to address the security analysis of Solidity code with LLMS. These questions are focused on understanding the current status of the tools and their performance. The research questions are:

- What are the static analysis tools to analyse security vulnerabilities in Solidity?
- Are there any studies related to the usage of LLMS to detect security vulnerabilities? Are these tools effective?

3.3 Research Process

At this stage, the research process will be presented in detail. Starting by defining which engines were used for the research, followed by the keywords, and ending with the inclusion and exclusion criteria.

3.3.1 Engines

The engines, also known as digital library sources, must be carefully selected since the validity of a study will depend on the proper selection and on whether the engine itself is adequate for the area under investigation. Based on that, the engines selected were:

- IEEE Xplore: is a digital library related to computer science, where scientific and technical content, such as articles, journals, and books published by IEEE, can be found.
- ACM Digital Library: is a digital library of ACM publications, including journals, conference proceedings, technical magazines, newsletters, and books.
- Science Direct: is a digital library for scientific, health and technical literature.

3.3.2 Keywords

Based on the PICOC framework, the keywords for the research were defined iteratively. During this definition, the keywords were added one by one and replaced by the respective synonym if needed. That process resulted in the following search string:

solidity AND security AND vulnerabilities AND (static analysis OR LLM)

The search string was adapted in order to be used in all engines mentioned in chapter 3.3.1. The keywords were applicable only in the abstract to avoid out-of-topic articles, and because abstracts are responsible for transmitting a clear and concise idea of what will be presented in the article.

3.3.3 Inclusion and exclusion criteria

In order to be able to collect only relevant articles for the research, inclusion and exclusion criteria were defined. Starting by defining that the research will be mainly focused on books, conference papers, journals and technical reports. The language of the articles must be preferably English, given that this will reach a wider scientific community. Since it is a new subject, only articles from the last 5 years were considered. All the sources which are not helpful to answer the research questions are automatically excluded, as well as all duplicated articles.

There are at least two ways to analyse vulnerabilities in Solidity code. Through the .sol file, where the code is human-readable, and through the EVM bytecode, which is the binary file generated after compiling the smart contract. Articles related to analysis through EVM bytecode will be excluded. Furthermore, studies that focus solely on a single security vulnerability should not be considered.

Table 3 summarises the inclusion and exclusion criteria.

Inclusion	Exclusion
Papers written in English	Works in other domains not relevant for the research questions
Books, conference papers, journals and technical reports	Works related to the analyse of EVM bytecode.
Articles from the last 5 years	Studies focused solely on a single security vulnerability

Table 3 - Inclusion and Exclusion Criteria.

3.4 Study Selection

After defining the entire research process, the actual selection of articles began by removing those that were irrelevant. In the first phase, after applying all the inclusion criteria, the selection of studies began by reading all the titles and abstracts to understand which ones met the exclusion criteria "Works in other domains not relevant for the research questions".

Annex B presents the number of relevant studies to be analysed, as well as the search string that was used in each of the engines. It is important to note that the numbers presented in the column "Number of papers before exclusion criteria" already include the inclusion criteria defined in chapter 3.3.3.

The graphic in

Figure 5 shows the relation between the number of articles selected by engine. Although 27 articles are shown, only a total of 25 articles will be considered, since articles [36] and [37] were developed in collaboration between IEEE Xplore and the ACM Digital Library.

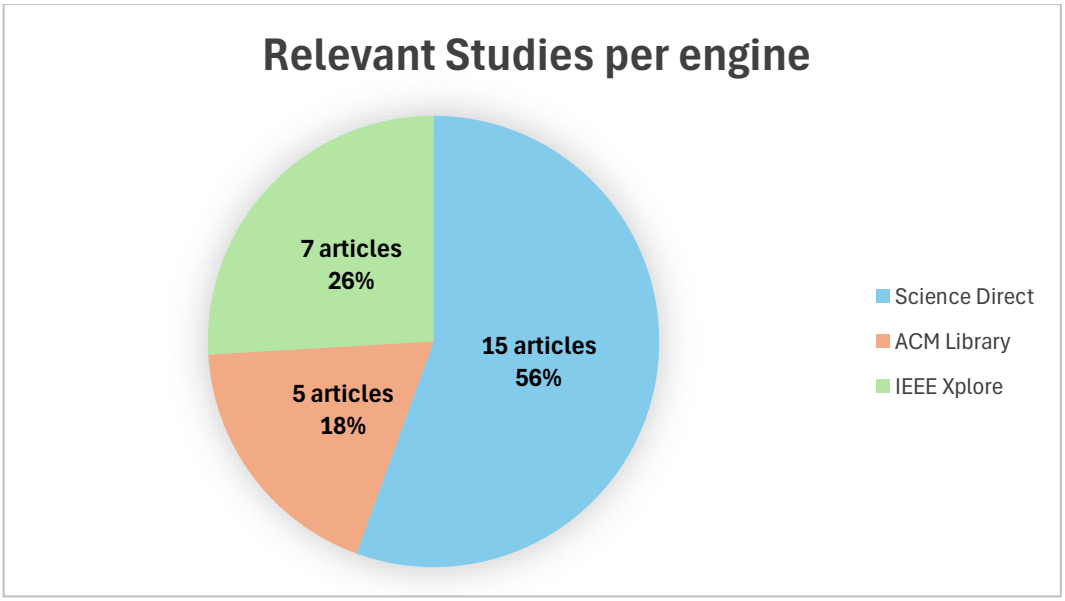

Figure 5 - Relevant Studies per engine.

3.5 Results Summary

Finally, the last step in a systematic literature review is presenting the results/knowledge obtained. This process will be summarised and aims to directly answer the previously defined research questions.

One of the conclusions that can be drawn from reviewing the literature is that the scientific community has made a significant effort to solve the problem of the detection of security vulnerabilities in Solidity code. Although not addressed in this paper, solutions such as using neural networks for detecting security vulnerabilities are beginning to emerge. After a more detailed analysis of the selected studies, it was possible to retain sufficient knowledge to answer the research questions with a high level of confidence.

What are the static analysis tools to analyse security vulnerabilities in Solidity?

Through the studies analysed, it was possible to verify that there are several code analysis tools, which are mainly divided into static code analysis tools and dynamic code analysis tools. The main differences between them are documented in chapter 2.2. Some approaches that combined both methodologies (static and dynamic analysis), called hybrid analysis tools, were excluded, so only pure static code analysis tools were considered for the analysis.

Table 4 presents a list of static analysis tools found in the articles, as well as their source code if available.

Tool	Source
EtherSolve	https://github.com/SeUniVr/EtherSolve
EthLint	https://github.com/duaraghav8/Ethlint
MadMax	https://github.com/nevillegrech/MadMax
Mythril	https://github.com/ConsenSys/mythril
Osiris	https://github.com/christoftorres/Osiris
Oyente	https://github.com/enzymefinance/oyente
Pluto	https://github.com/PlutoAnalyzer/pluto
RA	https://github.com/wanidon/RA
Rattle	https://github.com/crytic/rattle
smartAce	https://github.com/contract-ace/smartace
SmartCheck	https://github.com/smartdec/smartcheck
SmartDagger	Not available
SmartFast	https://github.com/SmartContractTools/SmartFast
SmartPulse	https://github.com/utopia-group/SmartPulseTool
Slither	https://github.com/crytic/slither
SoMo	https://github.com/VPRLab/SoMo
Solhint	https://github.com/protofire/solhint
solscan	https://github.com/riczardo/solscan

Table 4 - Static Analysis Tools extracted from the literature review.

Are there any studies related to the usage of LLMS to detect security vulnerabilities? Are these tools effective?

As previously stated, the process to detect security vulnerabilities through the traditional tools (static analysers) can be complex and sometimes time-consuming. So, the usage of the LLMS leverages the strengths of machine intelligence to operate at high speed, scale, and without fatigue [38]. In line with this, there are some studies dedicated to exploring this topic.

LLMSmartSec

Authors in [38] propose LLMSmartSec a hybrid solution that combines the use of LLM with CFG (Control Flow Graphs). CFG is basically a graphical representation of control flow during the execution of the program or applications.

The proposal was built in several stages. Firstly, they start to train an LLM using GPT-4 to understand solidity code. After that, they provide some solidity code to be analysed from three different perspectives: developer (LLMDev), auditor (LLMAudit), and ethical hacker (LLMeHack), and to detect possible security vulnerabilities as well as their fixes. Thus, while training the LLMs they have stored all the code and respective vulnerabilities/fixes in a CFG annotated. After building the CFG annotated based on that, they start to train another LLM (LLMGraphAgent), an open-source one, to be able to use it without associated costs.

At the end, they have created a User interface (UI) which allows direct interaction with the previously trained LLM. As shown in Figure 6, once this request is made, it will be redirected to LLMGraphAgent which will analyse if the code matches the existing patterns or not. If it matches, the analysis summary obtained from the LLMGraphAgent will be redirected to the LLMAudit, which will create the audit report. On the other hand, if the code does not match, it will be analysed by other LLMs (LLMDev and LLMeHack) and then provide the audit report to the user.

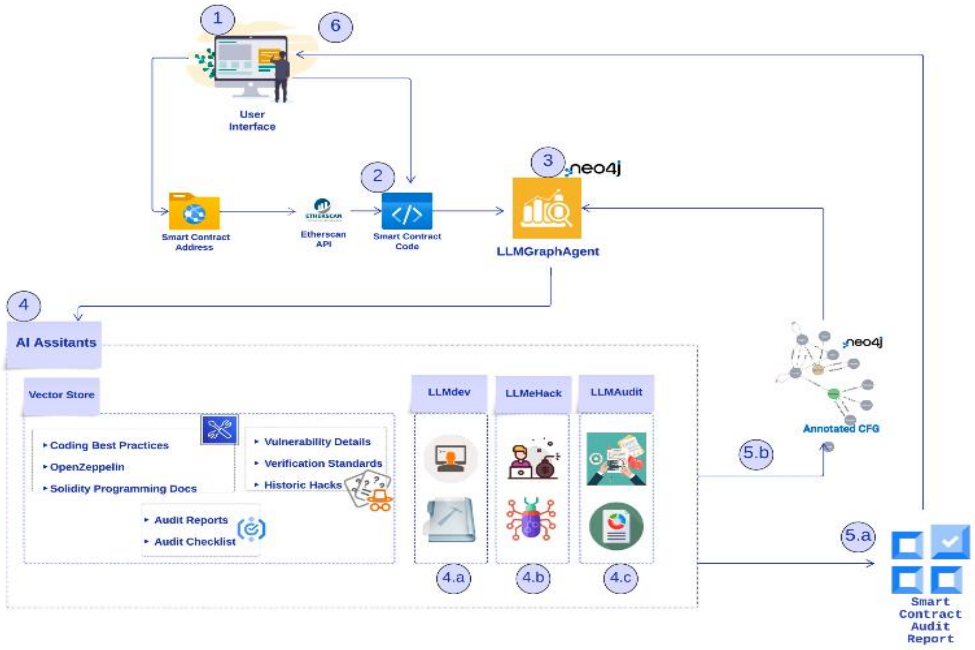

Figure 6 - LLMSmartSec Architecture. From [38].

Regarding the efficiency of this tool, compared to a manual audit of the entire project [39], it was found that, in addition to being much faster, it was also able to identify more vulnerabilities and fix more issues.

FELLMVP

Unlike LLMSmartSec, which aims to detect any type of vulnerability, there are other proposals that focus on a restricted set of vulnerabilities, as is the case of FELLMVP [40]. Instead of using just one LLM for the detection of vulnerabilities, the approach proposed for FELLMVP is slightly different, and the idea behind that is to have one LLM per vulnerability. There are only eight vulnerabilities the tool can detect:

- Unchecked External Call;
- Timestamp Dependency;
- Reentrancy;
- Integer Overflow;
- Ether Frozen;
- Dangerous Delegatecall;
- Block Number Dependency;
- Ether Strict Equality.

The tool begins by analysing the solidity file, extracting the contracts and functions present in it, which will subsequently create an CEC file (textual representation of smart contracts introduced by the authors for contract-level analysis). Those files will be grouped into eight

different datasets that are directly related to each LLM. Then, based on the datasets created, all eight LLMs (Gemma-7B) are trained to be able to detect the specific vulnerability assigned to them. One layer above, there is an AI agent that is tasked with predicting vulnerability types in smart contracts using the pre-trained LLMS.

This proposal was evaluated against other LLMS, such as Gemma-7b Base and trained Gemma-7b and it was demonstrated that FELLMVP significantly outperforms existing methods with an accuracy of 98.8%.

GPTScan

GTPScan, proposed by [36], described as the first tool combining GPT with static analysis for smart contract logic vulnerability detection, was implemented using the GPT-3.5-turbo. Although some GPT models are capable of detecting vulnerabilities through high-level vulnerability descriptions, the idea behind GPTScan is to adopt a different approach by breaking down vulnerability types into scenarios and properties. For example, for the vulnerability "Unauthorized Transfer", the scenario is "involve transferring token from an address different from message sender" and the property is "and there is no check of allowance/approval from the address owner".

In order to reduce the possibility of obtaining random results, some prompt rules were also defined, with special emphasis on the fact that the response generated does not have to be explained and should be limited to a "yes" or "no". After GPT analyse the scenario and the property, in case of vulnerability confirmed, a static analysis is performed to confirm the existence of a security vulnerability.

This proposal was deeply analysed with three datasets with around 400 contract projects and 3000 Solidity files, achieving a high precision (over 90%) for token contracts and acceptable precision (57.14%) for large projects. GPTScan is fast, cost-effective, and capable of discovering vulnerabilities missed by human auditors.

Vulnerability Detection using LLM and Graph Structural Analysis

One study [41] suggests an approach that combines three key modules. The first module is related to the Code Extraction, which preprocesses Solidity code by parsing and transforming it into meaningful representations. The second module: Graph Extraction, works in parallel to capture structural relationships using Abstract Syntax Tree (AST) and CFG to generate graph-based features and to provide a more comprehensive view of the contract's architecture.

The third module: Vulnerability Detection, integrates the outcome from the second module and after normalisation, it is used by the LLM (GPT-3.5-Turbo) to detect vulnerability analysis. Figure 7 illustrates the architecture of the solution proposed by [41].

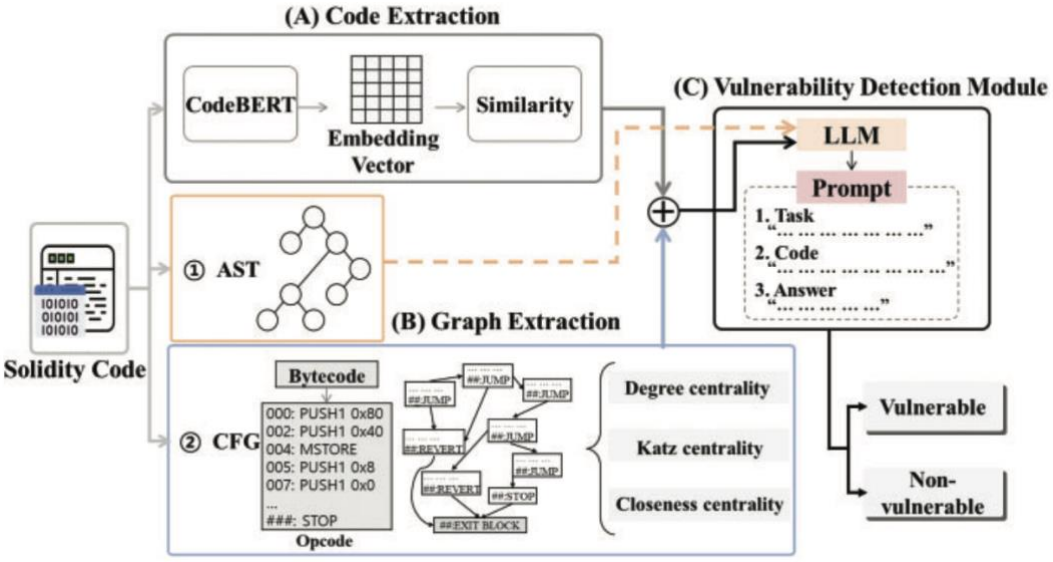

Figure 7 - Architecture of the proposed solution by [41].

In addition to the works presented above, studies such as [42] were also analysed. There an LLM is trained through a data set and hyper parametrization without involving the creation of a tool itself. The study uses DistilBERT, the distilled version of BERT, that retains much of its performance while being smaller and faster and has been successfully applied in domains that require efficient and effective text analysis. From the study is possible to conclude that DistilBERT performs better and consequently achieves a better accuracy when the dataset provided is balanced.

Conclusion

The studies analysed showed that all LLMs approaches were faster and more effective than traditional tools or manual audits. It is possible to extract from them that the number of false positives also reduces with these tools.

Although all the studies presented included performance metrics, it is not possible to determine which approach is most effective, as the measures used were not always the same. Furthermore, there is no guarantee that the contracts analysed are the same, nor that the data used to train the LLMS are the same. Table 5 summarises the tools addressed as well as their respective LLM.

Tool	LLM
LLMSmartSec	GPT-4
FELLMVP	Gemma-7b
GPTScan	GPT-3.5-turbo
Vulnerability Detection using LLM and Graph Structural Analysis	GPT-3.5-turbo

Table 5 - Tools and associated LLM.

As stated in article [43], pre-trained models on datasets containing Solidity source code (such as StarCoderBase-1b, StarCoderBase-3b and StarCoderBase-7b) are likely to better understand its specific syntax and semantics, potentially improving vulnerability detection. Besides that, in the same article can be seen that the parameterisation has a big influence on the efficiency of the LLM.

4 Design

After the review of the literature and the conclusions obtained through this. This chapter describes the decisions that were considered to conduct the comparison analysis. Starting by the selection of the security vulnerabilities, the tools to be used (static analysis tools and LLM), the presentation of the dataset and the strategy used to define the prompt. At the end is presented the solution architecture and the performance metrics that will be evaluated.

4.1 Selection of security vulnerabilities

Due to the high number of vulnerabilities addressed by [26] and the lack of resources and computing power, only two vulnerabilities will be analysed. According to data from Web3HackHub, in 2024, 149 incidents were documented, resulting in losses of $1.42 billion.[44]. Of the documented incidents, only a few can be detected by analysing the source code.

Thus, two of the most frequently reported vulnerabilities that can be detected through source code analysis were selected. **Access Control vulnerability** the most reported incident, 37 times, and generated losses of $953.2M. **Reentrancy Attack** reported 6 times, also generated significant losses of $35.7M.

Reentrancy, one of the most common vulnerabilities, occurs when an external function calls another untrusted contract, and an attacker gains control of it: they can make a recursive call back to the original function, unexpectedly repeating transactions that would have otherwise not run, and eventually consume all the gas [45]. There are several ways to prevent and reduce the impact of this kind of vulnerability, for example, using mutex/reentrancy guard and setting a gas limit.

An access control vulnerability happens when an attacker gains access to functions or variables that cannot be accessed. Functions without modifier restrictions indicate that anyone has access and can manipulate them, and the core functions can be manipulated by malicious attack performers, leading to vulnerable smart contracts. To prevent this vulnerability, all the accesses

given to a function must be carefully selected by following the principle that the privileges granted should always be the minimum required.

4.2 Selection of Tools

Since one of the main objectives of this study is to understand whether LLMS are more effective than static analysis tools for detecting security vulnerabilities in Solidity code, the first step is to choose which tools will be used to conduct an analysis that allows to provide an accurate answer. The selection of these tools is based on the criteria defined in the chapters 4.2.1 and 4.2.2.

4.2.1 Static Analysis Tools

Since most existing datasets with vulnerabilities contain files with source code to facilitate and accelerate the analysis process, only tools capable of analysing source code will be considered. This way, tools that analyse EVM bytecode are excluded. Constantly updating tools is a very important factor when conducting this type of analysis, especially in an area such as this that is constantly evolving. Therefore, the tools selected for analysis should be maintained.

Considering that the analysis will be restricted to the reentrancy and Access control vulnerability as stated in chapter 4.1, the tools must be able to detect at least those vulnerabilities. Finally, although tools such as EthLint and Solhint were considered at an early stage because they are also static analysis tools, those tools are more focused on syntactic analysis and not so much on detecting security vulnerabilities. Therefore, they should also be excluded. Based on these assumptions, the inclusion and exclusion criteria presented in Table 6 were created.

Inclusion	**Exclusion**
Tools able to analyse source code	Tools able to analyse EVM bytecode
Tools maintained	Tools like Lints or style checks
Tools focus on security vulnerabilities	Tools deprecated
Tools able to detect at least reentrancy and access control vulnerability	

Table 6 - Criteria for the selection of Static Analysis Tools.

The criteria were duly applied to each of the static analysis tools found in the literature review, and the detailed analysis can be seen in Table 7.

Static Analysis Tools	Source Code/ EVM Bytecode	Maintained	Reentrancy & Access Control Detection	Lints or Style checks
Slither	*Source Code*	*Yes*	*Yes*	*No*
Solhint	Source Code	Yes	No	Yes
SmartCheck	Source Code	No	Yes	No
solscan	Source Code	No	Yes	No
EtherSolve	EVM Bytecode	No	Just Reentrancy	No
Mythril	EVM Bytecode	Yes	Yes	No
Oyente	EVM Bytecode	No	Just Reentrancy	No
Osiris	EVM Bytecode	No	No	No
MadMax	EVM Bytecode	No	No	No
Rattle	EVM Bytecode	No	Yes	No
SmartAce	Source Code	No	No	No
SmartDagger	EVM Bytecode	-	-	
SmartFast	Solidity source	No	Yes	No
SmartPulse	Solidity source	No	No	No
SoMo	Solidity source	Yes	No	No
EthLint	Solidity source	Yes	No	Yes
Pluto	Solidity source	No	Just Reentrancy	No
RA	EVM bytecode	No	Just Reentrancy	No

Table 7 - Application of criteria selection of Static Analysis Tools.

As shown in Table 7, only Slither [46] met all selection criteria.

4.2.2 LLM

Unlike the selection of static analysis tools, the strategy to select the LLM to be used follows slightly different principles, seeking to explore LLMS that have not yet been studied. So, the LLM models presented in the literature review (GPT-4, Gemma-7b, DistilBERT and GPT-3.5-turbo) are automatically excluded. Another important aspect to be considered is whether their use is free of additional costs.

In line with the conclusion extracted from [43], which states to the fact of LLMS pre-trained on datasets with solidity code are likely to better understand its specific syntax and semantics, the LLM selected for the analysis was codellama [47]. This model, in addition to meeting the previously defined criteria, stands out for being specialised in code generation and discussion.

4.3 Dataset

The dataset SB Curated is a “dataset for research in automated reasoning and testing of smart contracts written in Solidity” [48] that contains nine types of vulnerabilities categorised and organised. As stated by the creators of the dataset, it was created “specifically to measure the accuracy of automated analysis tools” [48].

Only the solidity contracts related to the reentrancy and access control vulnerabilities were considered for the analysis done in this project. The remaining files were discarded because they were not considered relevant to the analysis.

After the filtering process, the dataset was composed of 32 solidity files (58 smart contracts) with reentrancy vulnerability and 18 solidity files (25 smart contracts) with access control vulnerability.

In order to have a control group with some smart contracts without vulnerabilities, the repository [49] was analysed, which contains the audit report of the smart contracts and projects audited by the company InterFi. The vulnerabilities in the audit report are generally identified in a Table, - take Table 8 as an example - where they are classified by severity and their status.

Status	Critical 🔴	Major 🟠	Medium 🟡	Minor 🟢	Unknown 🟤
Open	0	0	0	0	0
Acknowledged	0	0	1	2	1
Resolved	0	0	0	0	0

Table 8 - Example of vulnerability classification in audit reports of InterFi.

The audit reports were briefly analysed, and seven projects (60 smart contracts) were selected. The criteria used to select them were: no open vulnerabilities (regardless of severity level) and, if they had acknowledged or resolved vulnerabilities, these could not be classified as major or critical. The audit reports for the selected contracts can be found in the *interfi_analysis* folder of the project repository [50].

Table 9 summarises the composition of the dataset, divided into the three categories: Reentrancy, Access Control, and Non-Vulnerable, highlighting the amount (SCs) presented in each category.

Categories	**Solidity Files**
Reentrancy	32 (58 SCs)
Access Control	18 (25 SCs)
Non-Vulnerable	7 (60 SCs)
Total	57 (143 SCs)

Table 9 - Dataset Composition.

4.4 Prompt Design Strategy

The increased use of artificial intelligence technologies has led to the emergence of new areas such as prompt engineering. Prompt engineering is used to develop and optimise prompts (the input provided to the model) to efficiently use LLMS [51]. It also helps to understand the capabilities and the limitations of the LLM.

The strategy used to design the prompt for this analysis mixes three techniques presented by the Prompt Engineering Guide [51]

- **Role Playing:** This technique consists of attributing a specific role to the LLM model. By doing that, it is possible to improve the contextual understanding of the remaining prompt and prepare the model to perform a specific task.
- **Structured output:** Providing a clear structure/format of the information that needs to be answered will help to analyse data more quickly and easily.
- **Zero shot approach:** Since the codellama model was pre-trained with Solidity code, no vulnerability examples were provided in the prompt in order to use only the pre-trained data and not distort it with examples that may be very similar to the test data.

The combination of these three techniques has made it possible to develop a more optimised and consistent prompt that will make better use of the Codellama model.

4.5 Solution Architecture

The analysis conducted for the tools previously selected (Slither and Codellama), it is represented by the architecture in Figure 8. The first step of the analysis was to filter the selected dataset (SB Curated) by removing all the files not related to the selected vulnerabilities: Reentrancy and Access control, and adding the non-vulnerable smart contracts.

Subsequently, the second phase is divided into two analyses: Static Analysis and LLM Analysis. Static Analysis begins with the definition of the solidity compiler, and then the smart contract is analysed using Slither. LLM analysis starts with the refinement of the SB Curated Filtered. This refinement is needed since most of the contracts have comments that suggest the vulnerability present. The smart contract refined will be used by the run_ollma.py to create the final prompt that will enable the request to the codellama.

At the end, all the analysis details for each tool are stored in Comma-Separated Values (CSV) files that will be used to collect all the information needed to calculate the performance metrics mentioned in chapter 4.6

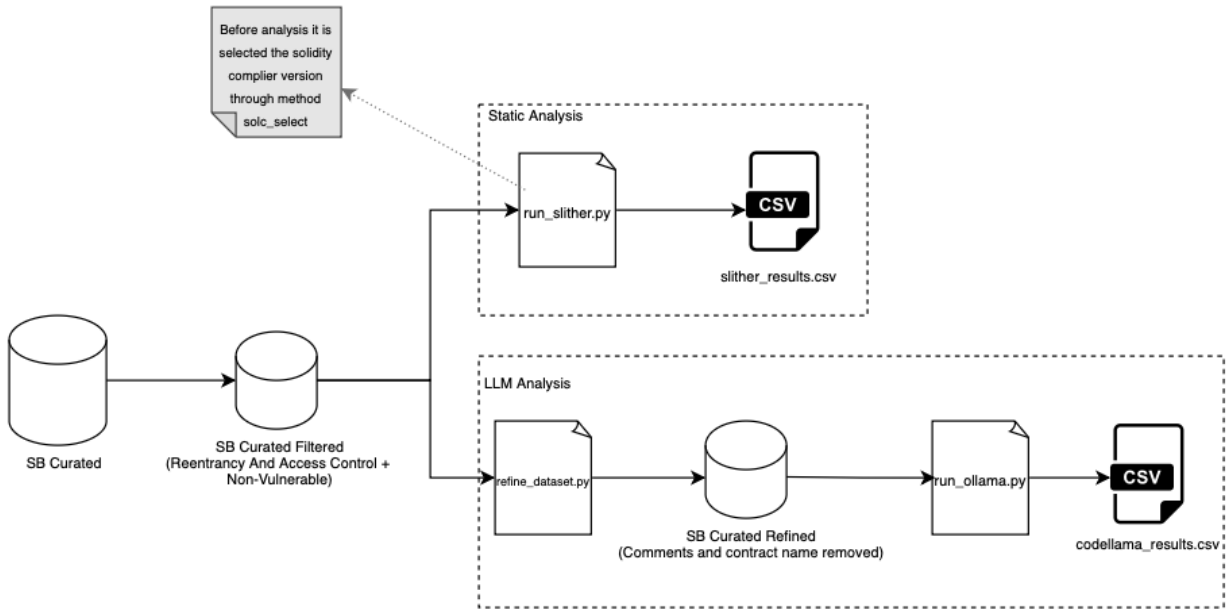

Figure 8 - Solution Architecture.

All the implementation details related to the solution architecture presented Figure 8 are explained in chapter 5.

4.6 Performance Metrics

In order to draw conclusions from the analysis, it is necessary to evaluate the performance of the tools involved. For that, a GQM (Goal Question Metric) approach was defined.

GQM approach is an analysis technique introduced by [52] to enhance software products and the development process. Nowadays, it is widely used to measure and evaluate tools or projects. Starting with the definition of the Goal, the approach is divided into three different stages: [53]

- **Goal**: describes the purpose and the object to be measured. To define a clear goal, it is necessary to understand *why* the measure is needed.
- **Question(s)**: The main idea behind them is to explore and characterise the goal. They should be operationally focused, helping to evaluate the progress against the goal.
- **Metric(s)**: are used to provide clear information that will enable to answer the questions previously defined. The metrics can be used to address multiple questions.

As previously mentioned in chapter 1.3, the goal for the current work is to evaluate and compare the effectiveness of an LLM and a static analysis tool for detecting security vulnerabilities in Solidity code. Based on that, the following questions were defined:

- Q1: Are LLMS able to detect security vulnerabilities in Solidity code?
- Q2: Are the LLMS more effective than the static analysis tools to detect security vulnerabilities in Solidity code?

Once well-defined the questions, the choice of the metrics began by firstly understanding if the metrics are costly to the computer or impractical to measure. To answer the question Q1, the metrics used were *recall* and *precision*.

Recall

The recall metric is used to understand what is the percentage of positive results that were correctly classified as positives [54]. In the current analysis, this metric will provide the percentage of vulnerabilities that were properly identified. That way is possible to have evidence about whether the LLM model can detect security vulnerabilities in Solidity code.

To calculate recall, it was used the following formula:

$$Recall = \frac{TruePositives}{TruePositives \ + \ FalseNegatives}$$

Precision

Besides the recall metric, to be able to properly answer Q1, it is also necessary to consider precision. This metric is used to identify from the positive results which of them are truly positives [54].In the context of the analysis conducted with this metric, it is possible to see from the vulnerabilities detected how many were truly identified as vulnerabilities.

In this case, a high precision indicates that the tools produce fewer false alarms. To measure the precision, the following formula was used:

$$Precision = \frac{TruePositives}{TruePositives \ + \ FalsePositives}$$

Unlike Q1, Q2 has a more comparative character, and for that, the metrics used were: *F1-score*, *accuracy* and *execution time*.

F1-Score

F1-score is typically seen as “harmonic mean of precision and recall” [54], and it aims to balance the false positives and negatives. This metric will show which tool achieves the best trade-off between detecting more vulnerabilities and being more precise to detect them (raising fewer false alarms).

To calculate the F1-score, the following formula was used:

$$F1 \ score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

Accuracy

The accuracy metric indicates the amount of correct classifications, whether positive or negative [54]. It provides a brief overview of how often the results provided by the tool are correct. After having this value for the static analysis tool and for the LLM model, it is possible to understand which one is more accurate.

To calculate the accuracy, the following formula was used:

$$accuracy = \frac{TruePositives + TrueNegatives}{TruePositives\ +\ TrueNegatives + FalsePositives + FalseNegatives}$$

Execution Time

Another metric that will enable the comparison between a static code analysis tool and an LLM is the tool's execution time, i.e., the time it takes to analyse whether the code has vulnerabilities or not.

Since all the executions will be carried out using the command line, to measure the execution time, it was considered the time required for the entire instruction process.

After defining the goal, questions and metrics, Figure 9 illustrates the GQM map.

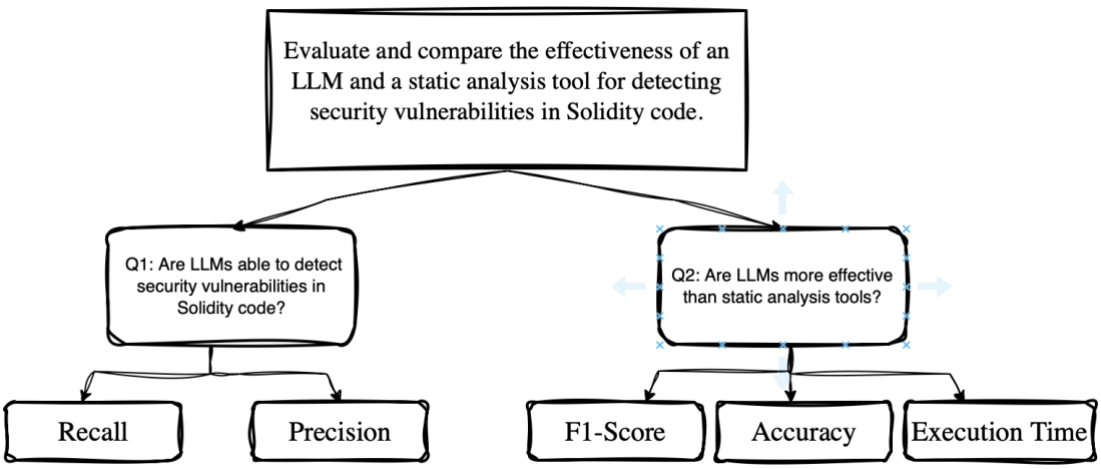

Figure 9 - GQM mapping.

5 Implementation

After the design phase, with all the elements involved in the analysis duly selected, the next chapter will focus on implementation, explaining the technical details of the analysis. This chapter can be divided into three main phases: exploratory, development, and testing. The chapter concludes with an explanation of how the results were extracted and presents the values obtained for each of the performance metrics.

5.1 Exploratory Phase

Before proceeding with the actual implementation of the analysis, a preliminary learning phase is necessary. This phase focuses on installing and learning how the selected tools behave and what their limitations are.

The static code analysis tool was installed using Python, following the official documentation found in the respective repository. Although the tool is available in a Docker image, the strategy adopted was to install it locally, so in the next phase would be easier to access it programmatically. For the LLM, the approach was slightly different, and the model was installed on OLLAMA[55], which is a tool that allows open-source LLMs to be used on a private computer. Interaction with this tool can be done through a chatbot by selecting the LLM model or programmatically by providing the respective prompt.

Once the tools were installed, a few tests were carried out with only one vulnerability of each type (Reentrancy and Access Control) to verify that the tools behaved as expected. The test results were positive, meaning that the tools were able to detect security vulnerabilities without any issue.

After obtaining an overview of how these tools worked for a single smart contract, the alternative of directly analysing a project was explored, concluding that only Slither was suitable for that, while the LLM (codellama) require the provision of contract by contract.

The analysis was conducted on a desktop with the following specifications: MacOS Sequoia version 15.6.1 (24G90), with 16 GB of RAM, and an Apple M1 processor

5.2 Development Phase

After understanding how the tools work, the development phase began. Due to the high number of contracts to be analysed, it was necessary to automate their analysis using some Python scripts.

The decision to choose Python as a programming language was based on the fact that it was a language previously used by the intervener in other contexts, such as academic and professional. In addition to being a language commonly used for process automation due to its flexibility, it is also a language that is easy to integrate with the selected tools. All the work produced, as well as supportive files, can be found at the project repository [50].

The development phase initially was focused on creating scripts to analyse the dataset with the static analysis tool, and only at a later stage was the analysis performed using LLM.

5.2.1 Slither

As mentioned earlier in chapter 4.2.1 the tool selected for the analysis was slither. Slither is a widely used Python-based tool that can detect nearly 100 types of security vulnerabilities and code optimisations. It receives as initial input the Solidity Abstract Syntax Tree (AST) generated by the Solidity compiler from the contract source code, which means that all contracts are pre-compiled before being analysed [56]. Therefore, before running the tool, it is necessary to be on the same solidity version as the smart contract, and since the selected dataset has multiple smart contracts on multiple versions of Solidity, to ensure that all contracts are suitable for analysis, it was necessary to add extra logic through the `set_solc_version` function, which will basically check if the solidity compiler version is the same as the one in the smart contract and, if not, will install it and use It.

After setting the correct version of Solidity, the analysis is performed by calling *slither* tool as a subprocess. When a vulnerability is found, the result of the process is 255 and is printed to stderr. Once finished, the analysis is automatically printed to CSV file for further analysis.

Although mentioned in the development phase that *slither* allows the analysis of an entire project, since it is necessary to compile all contracts before analysing them, the approach adopted was to analyse file by file.

5.2.2 CodeLLAMA

As mentioned in chapter 4.3, the dataset used to conduct the analysis has smart contracts properly organised and categorised, which, when used in an LLM, can lead to unbalanced results.

Therefore, before starting the implementation, it was necessary to refine the contracts by removing the additional information that is important for categorisation, namely comments. Besides that, the name of the contract was also replaced to not provide details about the possible vulnerability in the contract. This process was done using the Python script refine_dataset.py stored in [50].

Figure 10 shows an example of a refinement of the smart contract. In the left side is shown the original contract from SB Curated dataset, and on the right can be seen the smart contract after refinement.

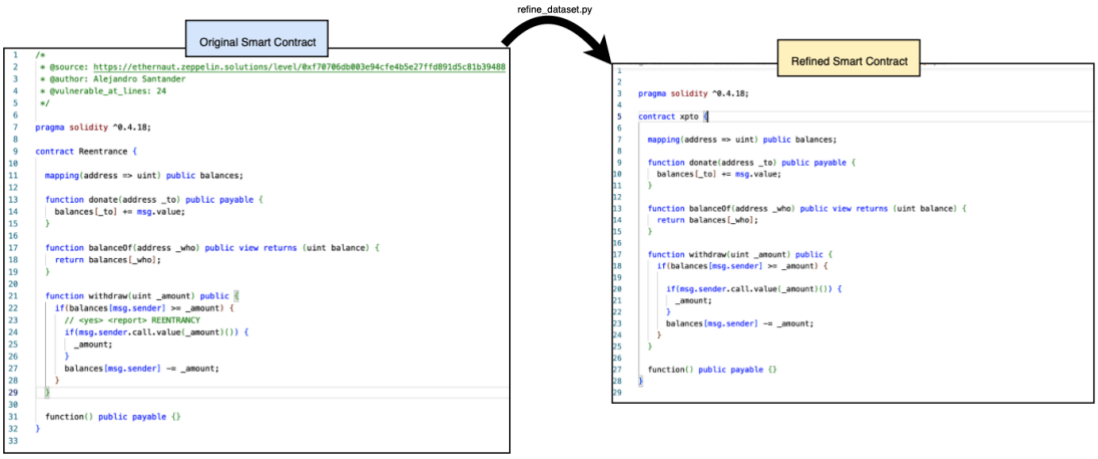

Figure 10 - Refinement of a smart contract.

After refining the dataset, it was necessary to define a common prompt for all requests. According to Berryman and Ziegler “concise and crisp prompts are generally more effective—plus, they use less computational power and are processed more quickly” [51, p. 123].

The wording of the prompts must be carefully considered and analysed in detail because, as with humans, a poorly worded question can lead to completely wrong answers and miscommunication. The prompt organisation is also an important aspect to avoid misunderstanding that could lead to wrong answers.

[51] suggests a prompt organised into three different sections. Starting with the introduction, it will set the context for the following two sections, allowing the model to start thinking about the problem from the start. The second section includes the context itself, providing all the necessary details. Finally, after giving the context, in the third section is the refocus, where it is necessary to end the prompt with a statement of the problem to be addressed. The refocus phase is essential because it lets users determine the scope that will define the boundaries of the generated analysis.

Figure 11 shows the prompt template created based on the assumptions previously mentioned and the prompt engineering techniques detailed in chapter 4.4. The prompt begins with the introduction (1), where the problem to be solved is presented. Next comes the context (2),

which in this case will be the smart contract source code. Finally, the refocus (3) defines how the answer should be given.

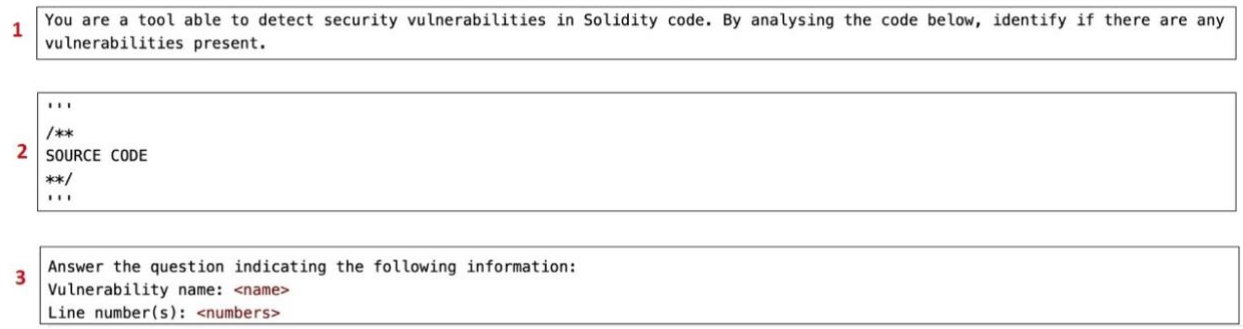

Figure 11 - Prompt template.

Once the dataset was refined and the prompt template created, the script for the analysis was implemented, allowing the *codellama* model to be used programmatically. The methodology adopted was very similar to the one previously presented in the *slither* tool.

The script receives as input the path to the Solidity file, which is parsed and used to build the respective prompt. After that, the request to the *codellama* model is done through *ollama,* and its answer is stored in a CSV file for further analysis. To facilitate that analysis, all answers are properly identified with the name of the file analysed.

5.3 Testing Phase

Due to the fact that the scripts produced were designed to perform an analysis and, at an early stage, did not have clear requirements for their implementation, the Test-Last Development (TLD) approach was adopted. TLD, also known as the traditional approach, consists in performing the tests once the code is finished.

Although this approach may provide less test coverage, it speeds up the development phase and is more suitable for small projects, as it avoids the overhead produced by other approaches like Test-Driven Design (TDD) [57]. TDD is a commonly used approach for testing that implies all tests are written before development.

Once the scripts were finalised, manual tests were first carried out to understand if the script produced the expected results. For these tests, some of the files in the dataset were used and the generated output was verified. In addition to manual testing, unit tests were developed to ensure that the scripts are properly protected against invalid inputs.

The unit testing is focused on testing individual units of source code by guarantying that the units can work independently. The unit tests were developed using the framework in Python *unittest.* An example of these unit tests is provided in the Source Code 4 and represents a test

case of the RunOllama method. The objective of this class is to test that when the file provided as input does not exist in the directory provided, the method ends with a system exit code 1.

```
1  class TestRunOllama(unittest.TestCase):
2      def test_nonexistent_file_exits_with_code_1(self):
3          with self.assertRaises(SystemExit) as cm:
4              run_ollama("nonexistent_file.sol")
5          self.assertEqual(cm.exception.code, 1)
```

Source Code 4 - Unit Test runOllama.

Besides the test previously presented, the other methods were also tested following the same approach to test the possible outcome of the method. All the unit tests can be found in the package tests of the project repository [50].

5.4 Analysis and Results

After completing all implementation phases, the results analysis phase began. As previously described, the results obtained from each tool were stored in a CSV under the folder *results* of the repository.

This analysis is stored independently, meaning that each tool has its own CSV file. These files have a slight difference in how they are organised. While *slither_analysis_results.csv* has the header in Table 10, the *codellama_analysis_results.csv* does not contains the *Result* column. The Result column is used for Slither, because the return code may vary depending on the result of the analysis. For example, if the tool is unable to analyse the file, the return code is 1, and a message is added indicating that the file was not analysed due to a tool error. Since the execution of codellama does not have different return codes, this column was not added to the CSV file.

ContractName	Result	Analysis	Execution Time

Table 10 - Table header of slither_analysis_results.csv file.

The first column, *ContractName*, stores the name of the Solidity file being analysed. The *Analysis* column contains the output generated by the tool, and *Execution Time* is the time the analysis takes.

Upon completion of the analysis by the tools, the results classification began. In order to be able to calculate the performance metrics presented in Chapter 4.6, the results were classified into four subcategories: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).

Table 11 illustrates how the results were classified based on the relation between vulnerabilities in the Solidity file and the result obtained by the tool.

Solidity file	Tool Result	Category
Vulnerable	Detected	True Positive (TP)
Vulnerable	Not detected	False Negative (FN)
Not vulnerable	Not detected	True Negative (TN)
Not vulnerable	Detected	False Positive (FP)

Table 11 - Classification of TP, FP, TN, FP.

The classification was done based on the two vulnerabilities mentioned in chapter 4.1 (Access Control and Reentrancy), for the contracts without vulnerabilities the analysis preformed was also solely done for these two vulnerabilities - all the classification details can be found at Annex C and Annex D. In addition to the results classification, the vulnerability present in each solidity file is also marked with an 'x'. and, to avoid mixing them with categorised files, these were also identified with the word 'detected'.

Due to an error during analysis with Slither, the files reentrancy_bonus.sol and reentrancy_cross_function.sol were excluded from the analysis, and they are marked as '-' in the Annex C and Annex D.

Table 12 summarizes the total amount of True Positives, True Negatives, False Positives and False Negatives for Slither and Codellama.

	Slither	Codellama
True Positives	38	36
True Negatives	5	4
False Positives	2	3
False Negatives	9	11

Table 12 - Total of TP, TN, FP and FN for Slither and Codellama.

Having all the results gathered, it is possible to calculate some of the performance metrics (recall, precision F1-Score and accuracy) needed for the evaluation by applying the formulas presented in chapter 4.6. Besides that, the other performance metric -the execution time - was calculated using the average of the times shown in the Annex C and Annex D.

Table 13 shows all performance metrics for both tools Slither and Codellama

Performance Metric	**Slither**	**Codellama**
Recall	0,8085	0,7659
Precision	0,95	0,9231
F1-score	0,8736	0,8372
Accuracy	0,7963	0,7407
Execution Time (seconds)	2,1694	56,3293

Table 13 - Performance metrics for Slither and codellama.

6 Conclusion

This chapter details all the conclusions drawn from the analysis carried out during this thesis. It firstly introduces what has been achieved, and next it describes the eventual future work that can be performed

6.1 Achievements

The comparison of Slither and Codellama offers valuable insights regarding the efficacy of both tools in detecting security vulnerabilities in Solidity code, and the results show that despite both approaches performing well, Slither achieves better results across all performance metrics. It achieves a recall of 0.81, compared to 0.77 for Codellama. Although this is not a significant difference, it demonstrates that Slither is a more effective tool at detecting security vulnerabilities within the dataset.

The results about accuracy show that Slither achieved an accuracy of 0.95, while Codellama achieved an accuracy of 0.92. This difference means that Slither generates fewer false positives, which shows that its detections are more reliable and consequently can reduce the time spent on the investigation of non-critical issues.

The F1 score, which balances the relationship between recall and precision, confirms Slither's overall superiority. With a score of 0.87 compared to 0.84 for Codellama, Slither exhibits a better balance between correctly identifying vulnerabilities and minimising the number of false positives. Besides that, the precision values show 0.80 for Slither and 0.74 for Codellama. These values provide further evidence of Slither's superior performance. As the percentage of contracts classified correctly, regardless of whether they are vulnerable or not, the higher accuracy on Slither shows that this tool is more consistent and trustworthy.

The main difference between these two tools is their execution time, with Slither achieving an average analysis time of 2.17 seconds and Codellama achieving 56.33 seconds. This time

difference can be explained by the fact that Slither is a tool designed specifically to analyse smart contracts, while the LLM model is capable of performing various tasks.

These results indicate that both Slither and Codellama can detect security vulnerabilities in Solidity code, but Slither emerges as the more reliable, efficient and faster tool. Its higher recall ensures greater coverage of vulnerabilities, and its precision reduces unnecessary extra work by not providing misleading results. On the other hand, although Codellama's performance is slightly weaker, it is still competitive and confirms the potential of large language model-based approaches in detecting security vulnerabilities in Solidity code.

In summary, Slither is currently the most mature and reliable option for detecting security vulnerabilities in smart contracts. However, the emergence of Codellama as a competitive alternative demonstrates the growing potential of LLM approaches, which crucial role in future research and practical applications.

6.2 Future Work

The detection of security vulnerabilities through LLMS is a constantly changing area, and although this study has demonstrated that these tools can be used to detect security vulnerabilities in solidity, there are still several opportunities for future exploration.

Starting with the vulnerabilities analysed, as mentioned earlier in this study, only two vulnerabilities (Reentrancy and Access Control) were considered. Extending this research to more vulnerabilities leads to a more comprehensive and inclusive analysis, which will provide a better overview of how the tools behave for others security vulnerabilities, such as, for example, Overflows and Denial of Services.

In addition to that, some works can be done related to the LLM itself. Nowadays, the results obtained from an LLM are impacted by many factors such as prompt techniques. As mentioned, the prompt techniques used for this study were role playing, structured output, and zero-shot approach. In future work, some experiments can be conducted with other techniques, such as few-shot training and chain-of-thought, to understand whether the performance of the LLM is improved by the different techniques and adapt the prompt if needed.

To conduct this study a pre-trained LLM was selected, and no fine-tuning was performed. In future, fine-tuning an LLM can improve its performance by creating and LLM specifically for detecting securtity vulnerabilities in solidity code.

Furthermore, to combine the benefits of both Slither and Codellama, the implementation of a hybrid tool, in which the code of smart contracts is analysed by both tools and the results are combined, is also a possible approach that can possibly enable a faster and more in-depth analysis.

References

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

[2] IBM, 'What Are Smart Contracts on Blockchain?', IBM. Accessed: Aug. 06, 2025. [Online]. Available: https://www.ibm.com/think/topics/smart-contracts

[3] Z. Zheng *et al.*, 'An overview on smart contracts: Challenges, advances and platforms', *Future Generation Computer Systems*, vol. 105, pp. 475–491, Apr. 2020, doi: 10.1016/j.future.2019.12.019.

[4] A. Bouichou, S. Mezroui, and A. E. Oualkadi, 'An overview of Ethereum and Solidity vulnerabilities', in *2020 International Symposium on Advanced Electrical and Communication Technologies (ISAECT)*, Marrakech, Morocco: IEEE, Nov. 2020, pp. 1–7. doi: 10.1109/ISAECT50560.2020.9523638.

[5] S. Palladino, 'The Parity Wallet Hack Explained', OpenZeppelin blog. Accessed: Aug. 06, 2025. [Online]. Available: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[6] M. Krichen, 'Strengthening the Security of Smart Contracts through the Power of Artificial Intelligence', *Computers*, vol. 12, no. 5, p. 107, May 2023, doi: 10.3390/computers12050107.

[7] P. Tantikul and S. Ngamsuriyaroj, 'Exploring Vulnerabilities in Solidity Smart Contract', in *Proceedings of the 6th International Conference on Information Systems Security and Privacy*, Valletta, Malta: SCITEPRESS - Science and Technology Publications, 2020, pp. 317–324. doi: 10.5220/0008909803170324.

[8] A. Singh, R. M. Parizi, Q. Zhang, K.-K. R. Choo, and A. Dehghantanha, 'Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities', *Computers & Security*, vol. 88, p. 101654, Jan. 2020, doi: 10.1016/j.cose.2019.101654.

[9] C. GPT, 'ChatGPT'. Accessed: Aug. 06, 2025. [Online]. Available: https://chatgpt.com

[10] G. Gemini, 'Google Gemini', Gemini. Accessed: Aug. 06, 2025. [Online]. Available: https://gemini.google.com

[11] M. Ortu, G. Ibba, C. Conversano, R. Tonelli, and G. Destefanis, 'Identifying and Fixing Vulnerable Patterns in Ethereum Smart Contracts: A Comparative Study of Fine-Tuning and Prompt Engineering Using Large Language Models', 2023. doi: 10.2139/ssrn.4530467.

[12] IEEE, 'IEEE Code of Ethics', 2020, [Online]. Available: www.ieee.org

[13] IEEE Author Center, 'IEEE Author Ethics Guidelines'. IEEE. [Online]. Available: https://pspb.ieee.org/images/files/files/opsmanual.pdf

[14] A. Bahga and V. K. Madisetti, ‘Blockchain Platform for Industrial Internet of Things’, *JSEA*, vol. 09, pp. 533–546, 2016, doi: 10.4236/jsea.2016.910036.

[15] A. S. Rajasekaran, M. Azees, and F. Al-Turjman, ‘A comprehensive survey on blockchain technology’, *Sustainable Energy Technologies and Assessments*, vol. 52, p. 102039, Aug. 2022, doi: 10.1016/j.seta.2022.102039.

[16] V. Buterin, ‘A next generation smart contract & decentralized application platform’, *Ethereum White Paper*, 2014, [Online]. Available: https://ethereum.org/en/whitepaper/

[17] T. M, ‘“Turing Completeness” and Early Computing’, Medium. Accessed: Mar. 22, 2025. [Online]. Available: https://medium.com/@tmlkm/turing-completeness-and-early-computing-b41cd41c83f8

[18] Bitstamp, ‘What is Turing complete?’, Bitstamp Trusted Crypto Exchange. Accessed: Mar. 30, 2025. [Online]. Available: https://www.bitstamp.net/learn/blockchain/what-is-turing-complete/

[19] W. Zou *et al.*, ‘Smart Contract Development: Challenges and Opportunities’, *IIEEE Trans. Software Eng.*, vol. 47, no. 10, pp. 2084–2106, Oct. 2021, doi: 10.1109/TSE.2019.2942301.

[20] QuickNode, ‘How does Ethereum Virtual Machine (EVM) work? A deep dive into EVM Architecture and Opcodes | QuickNode Guides’, QuickNode. Accessed: Mar. 30, 2025. [Online]. Available: https://www.quicknode.com/guides/ethereum-development/smart-contracts/a-dive-into-evm-architecture-and-opcodes

[21] DefiLlama, ‘DefiLlama’, DefiLlama. Accessed: Mar. 31, 2025. [Online]. Available: https://defillama.com/languages

[22] Rust, ‘Rust Programming Language’. Accessed: Mar. 31, 2025. [Online]. Available: https://www.rust-lang.org/

[23] Vyper, ‘Vyper’, Vyper. Accessed: Mar. 31, 2025. [Online]. Available: https://vyperlang.org

[24] Solidity, ‘Solidity — Solidity 0.8.29 documentation’. Accessed: Apr. 02, 2025. [Online]. Available: https://docs.soliditylang.org/en/v0.8.29/

[25] Solidity, ‘Contracts — Solidity 0.8.29 documentation’. Accessed: Jun. 16, 2025. [Online]. Available: https://docs.soliditylang.org/en/v0.8.29/contracts.html#function-modifiers

[26] SWC, ‘Smart Contract Weakness Classification (SWC)’. Accessed: Jul. 21, 2025. [Online]. Available: https://swcregistry.io/

[27] ‘Static Code Analysis Tools: A Systematic Literature Review’, in *DAAAM Proceedings*, DAAAM International Vienna, 2020, pp. 0565–0573. doi: 10.2507/31st.daaam.proceedings.078.

[28] L. Xin and C. Wandong, ‘A program vulnerabilities detection frame by static code analysis and model checking’, in *2011 IEEE 3rd International Conference on Communication Software and Networks*, Xi’an, China: IEEE, May 2011, pp. 130–134. doi: 10.1109/iccsn.2011.6013559.

[29] D. De Silva, P. Samarasekara, and R. Hettiarachchi, 'A Comparative Analysis of Static and Dynamic Code Analysis Techniques', May 12, 2023, *Institute of Electrical and Electronics Engineers (IEEE)*. doi: 10.36227/techrxiv.22810664.v1.

[30] J. Alammar and M. Grootendorst, *Hands-on large language models: language understanding and generation*, First edition. Sebastopol, CA: O'Reilly Media, Inc, 2024.

[31] J. McCarthy, 'What is AI? / Basic Questions'. Accessed: Jul. 30, 2025. [Online]. Available: http://jmc.stanford.edu/artificial-intelligence/what-is-ai/index.html

[32] B. Goertzel and C. Pennachin, Eds., *Artificial general intelligence*. in Cognitive technologies. New York: Springer, 2007.

[33] C. Stryker and J. Holdsworth, 'What Is NLP (Natural Language Processing)?' Accessed: Aug. 01, 2025. [Online]. Available: https://www.ibm.com/think/topics/natural-language-processing

[34] A. Carrera-Rivera, W. Ochoa, F. Larrinaga, and G. Lasa, 'How-to conduct a systematic literature review: A quick guide for computer science research', *MethodsX*, vol. 9, pp. 1–12, 2022, doi: 10.1016/j.mex.2022.101895.

[35] K. Petersen, S. Vakkalanka, and L. Kuzniarz, 'Guidelines for conducting systematic mapping studies in software engineering: An update', *Information and Software Technology*, vol. 64, pp. 1–18, Aug. 2015, doi: 10.1016/j.infsof.2015.03.007.

[36] Y. Sun *et al.*, 'GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis', in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Lisbon Portugal: ACM, Apr. 2024, pp. 2048–2060. doi: 10.1145/3597503.3639117.

[37] S. Hwang and S. Ryu, 'Gap between theory and practice: an empirical study of security patches in solidity', in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Seoul South Korea: ACM, Jun. 2020, pp. 542–553. doi: 10.1145/3377811.3380424.

[38] V. Mothukuri, R. M. Parizi, and J. L. Massa, 'LLMSmartSec: Smart Contract Security Auditing with LLM and Annotated Control Flow Graph', in *2024 IEEE International Conference on Blockchain (Blockchain)*, Copenhagen, Denmark: IEEE, Aug. 2024, pp. 434–441. doi: 10.1109/Blockchain62396.2024.00064.

[39] code-423n4, '2024-02-wise-lending/contracts at main · code-423n4/2024-02-wise-lending', GitHub. Accessed: Aug. 22, 2025. [Online]. Available: https://github.com/code-423n4/2024-02-wise-lending/tree/main/contracts

[40] Y. Luo, W. Xu, K. Andersson, M. S. Hossain, and D. Xu, 'FELLMVP: An Ensemble LLM Framework for Classifying Smart Contract Vulnerabilities', in *2024 IEEE International Conference on Blockchain (Blockchain)*, Copenhagen, Denmark: IEEE, Aug. 2024, pp. 89–96. doi: 10.1109/Blockchain62396.2024.00021.

[41] R.-Y. Choi, Y. Song, M. Jang, T. Kim, J. Ahn, and D.-H. Im, 'Smart Contract Vulnerability Detection Using Large Language Models and Graph Structural Analysis', *CMC*, vol. 83, no. 1, pp. 785–801, 2025, doi: 10.32604/cmc.2025.061185.

[42] J. Kim, T.-T.-H. Le, S. Lee, and H. Kim, 'Ethereum Smart Contracts Vulnerabilities Detection Leveraging Fine-Tuning DistilBERT', in *2024 International Conference on Platform Technology and Service (PlatCon)*, Jeju, Republic of Korea: IEEE, Aug. 2024, pp. 133–138. doi: 10.1109/PlatCon63925.2024.10830749.

[43] F. Sikder, Y. Lei, and Y. Ji, 'Efficient Adaptation of Large Language Models for Smart Contract Vulnerability Detection', in *Proceedings of the 21st International Conference on Predictive Models and Data Analytics in Software Engineering*, Trondheim Norway: ACM, Jun. 2025, pp. 65–74. doi: 10.1145/3727582.3728688.

[44] Credshields, 'SolidityScan's HackHub: Learn about Hacks & Prevent Attacks'. Accessed: Aug. 23, 2025. [Online]. Available: https://solidityscan.com/web3hackhub?year=2024

[45] N. Fatima Samreen and M. H. Alalfi, 'Reentrancy Vulnerability Identification in Ethereum Smart Contracts', in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, London, ON, Canada: IEEE, Feb. 2020, pp. 22–29. doi: 10.1109/IWBOSE50093.2020.9050260.

[46] crytic, 'Usage · crytic/slither Wiki · GitHub'. Accessed: Aug. 23, 2025. [Online]. Available: https://github.com/crytic/slither/wiki/Usage#detector-selection

[47] B. Rozière *et al.*, 'Code Llama: Open Foundation Models for Code', Jan. 31, 2024, *arXiv*: arXiv:2308.12950. doi: 10.48550/arXiv.2308.12950.

[48] Smartbugs-curated, *smartbugs/smartbugs-curated*. (Aug. 23, 2025). Solidity. SmartBugs. Accessed: Aug. 23, 2025. [Online]. Available: https://github.com/smartbugs/smartbugs-curated

[49] I. Network, *interfinetwork/project-delivery-data*. (Sep. 03, 2025). Solidity. Accessed: Sep. 11, 2025. [Online]. Available: https://github.com/interfinetwork/project-delivery-data

[50] franciscosilva-1999, *franciscosilva-1999/SecurrityVulnerabilitiesinSolidity*. (Aug. 29, 2025). Solidity. Accessed: Aug. 30, 2025. [Online]. Available: https://github.com/franciscosilva-1999/SecurrityVulnerabilitiesinSolidity

[51] J. Berryman and A. Ziegler, *Prompt engineering for LLMs: the art and science of building large language model-based applications*, First Edition. Sebastapol, CA: O'Reilly Media, 2025.

[52] V. R. Basili and D. M. Weiss, 'A Methodology for Collecting Valid Software Engineering Data', *IIEEE Trans. Software Eng.*, vol. SE-10, no. 6, pp. 728–738, Nov. 1984, doi: 10.1109/TSE.1984.5010301.

[53] C. Ciceri *et al.*, *Software Architecture Metrics*. O'Reilly Media, Inc., 2022.

[54] 'Classification: Accuracy, recall, precision, and related metrics | Machine Learning', Google for Developers. Accessed: Aug. 29, 2025. [Online]. Available:

https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall

[55] Ollama, ‘Ollama’. Accessed: Aug. 24, 2025. [Online]. Available: https://ollama.com

[56] J. Feist, G. Grieco, and A. 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)*, May 2019. doi: 10.1109/WETSEB.2019.00008.

[57] M. A. Yahya, ‘Mastering the Art of Test-Driven Development: Building Better Software, One Test at a Time’, Medium. Accessed: Sep. 01, 2025. [Online]. Available: https://medium.com/@mochammadagusyahya/mastering-the-art-of-test-driven-development-building-better-software-one-test-at-a-time-41f5d4c1db3c

Annexes

Annex A. Smart Contract Weakness Classification

Annex A presents the description of each vulnerability by the Smart Contract Weaknesses Classification as well as a possible way to prevent them[26].

SWC ID	Title	Description	Prevention
SWC-100	Function Default Visibility	By default, the functions are defined as public, and this could lead to a vulnerability if a developer forgot to set the visibility, and a malicious user is able to make unauthorized or unintended state changes.	The visibility of the function must be specified being external, public, internal or private.
SWC-101	Integer Overflow and Underflow	This happens when an arithmetic operation reaches the maximum or minimum size of a type.	Must be used a Safe math library for arithmetic operations consistently throughout the smart contract system.
SWC-102	Outdated Compiler Version	An outdated compiler version can be problematic especially if there are publicly disclosed issues that affect the current compiler version.	Must be used a recent version of the Solidity compiler.
SWC-103	Floating Pragma	Pragma must be locked to ensure that contracts do not accidentally get deployed using, an outdated compiler version that might introduce bugs that affect the contract system negatively. They must be complied and tested using the same version.	The pragma version must be checked to check the known bugs for the compiler version used.

SWC ID	Title	Description	Prevention
SWC-104	Unchecked Call Return Value	When the return value of a message call is not checked the execution will resume even if the called contract throws an exception. If the call fails accidentally or an attacker forces the call to fail, this may cause unexpected behaviour in the subsequent program logic.	When low-level call methods are used, it is necessary to make sure to handle the possibility that the call will fail by checking the return value.
SWC-105	Unprotected Ether Withdrawal	This vulnerability can be caused by wrongly naming a function intended to be a constructor, the constructor code ends up in the runtime byte code and can be called by anyone to re-initialize the contract.	Implement controls so withdrawals can only be triggered by authorized parties or according to the specs of the smart contract system.
SWC-106	Unprotected SELFDESTRUCT Instruction	Malicious parties can self-destruct the contract if access controls are missing or are insufficient.	The self-destruct functionality must be removed unless it is absolutely required. If there is a valid use-case, it is recommended to implement a multisig scheme so that multiple parties must approve the self-destruct action.
SWC-107	Reentrancy	It is one of the major dangers of calling external contracts is that they can take over the control flow, by calling contract before the first invocation of the function is finished. This may cause the different invocations of the function to interact in undesirable ways.	Guarantee that all internal state changes are performed before the call being executed. The usage of a Reentrancy lock is another way to prevent this vulnerability.
SWC-108	State Variable Default Visibility	Variable visibility must be defined to make easier to catch incorrect assumptions about who can access the variable.	Must be defined the visibility for all state variables.
SWC-109	Uninitialized Storage Pointer	Uninitialized local storage variables can point to unexpected storage locations in the contract, which can lead to intentional or unintentional vulnerabilities.	If the contract requires a storage object as in many situations this is actually not the case. If a local variable is sufficient, mark the storage location of the variable explicitly with the memory attribute. If a storage variable is needed, then initialise it upon

SWC ID	Title	Description	Prevention
		This vulnerability is only applicable for versions below than 0.5.0.	declaration and additionally specify the storage location storage.
SWC-110	Assert Violation	When the code is working properly, it should not be able to fail the assert statement. This can happen due to a bug on the code or a bad configuration of the function.	Must be consider whether the condition checked in the assert() is actually an invariant. If not, the assert() statement with a require() statement.
SWC-111	Use of Deprecated Solidity Functions	Using deprecated functions can leads to a reduction of the code quality. With new major versions of the Solidity compiler, deprecated functions and operators may result in side effects and compile error.	Solidity provides alternatives for the deprecated functions. Those deprecated ones must be replaced by the alternatives.
SWC-112	Delegatecall to Untrusted Callee	Delegatecall allows a smart contract to dynamically load code from a different address at runtime. Calling into untrusted contracts is very dangerous, as the code at the target address can change any storage values of the caller and has full control over the caller's balance.	Must make sure to never call delegatecall into untrusted contracts. If the target address is derived from user input, ensure to check it against a whitelist of trusted contracts.
SWC-113	Denial of Service with Failed Call	External calls can fail accidentally or deliberately, which can cause a Denial-of-Service conditions.	Must be avoid combining multiple calls in a single transaction, especially when calls are executed as part of a loop and must be always assume that external calls can fail Implement the contract logic to handle failed calls.
SWC-114	Transaction Order Dependence	When transactions are sent to the Ethereum network, they are forwarded to each node for processing. Thus, a person who is running an Ethereum node can tell which transactions are going to occur before they are finalized. A race condition vulnerability occurs when code depends on the order of the transactions submitted to it.	Must be added a field to the inputs of approve, which is the expected current value, and to have approve revert.

SWC ID	Title	Description	Prevention
SWC-115	Authorization through tx.origin	tx.origin is a global variable in Solidity which returns the address of the account that sent the transaction. Using the variable for authorization could make a contract vulnerable if an authorized account calls into a malicious contract.	Must be used msg.sender instead of tx.origin.
SWC-116	Block values as a proxy for time (Time dependence)	SCs often need access to time values to perform certain types of functionalities. Values such as block.timestamp, and block.number are not safe for most purposes because they are not constant and are subject to change.	Must be used *oracles* as an alternative to the *block*.
SWC-117	Signature Malleability	It is often assumed that the signature is unique, but signatures can be altered without the possession of the private key and still be valid. A malicious user can slightly modify the parameters and create other valid signatures. A system that performs signature verification on contract level might be susceptible to attacks if the signature is part of the signed message hash.	A signature should never be included into a signed message hash to check if previously messages have been processed by the contract.
SWC-118	Incorrect Constructor Name	Constructors are special functions that are called only once during the contract creation that often perform critical, privileged actions such as setting the owner of the contract. However, its name must be exactly the same as the name of the smart contract otherwise it can lead .to security issues, for example when smart contract code is re-used with a different name, but the name of the constructor function is not changed accordingly. This vulnerability is only applicable for versions before than 0.4.22.	The contract must be updated to a recent version (after 0.4.22) of the Solidity compiler and changed to the new constructor declaration.

SWC ID	Title	Description	Prevention
SWC-119	Shadowing State Variables	Solidity allows for ambiguous naming of state variables when inheritance is used which can lead in security issues when applicable in complex contracts.	The storage variable layouts for the contract systems carefully must be reviewed and must be removed any ambiguities. The compiler warnings must be checked as they can flag the issue within a single contract.
SWC-120	Weak Sources of Randomness from Chain Attributes	The randomness in Ethereum is very challenging. Depending on the criteria that it is used to generate a random condition it can be manipulated.	External sources of randomness via oracles must be used.
SWC-121	Missing Protection against Signature Replay Attacks	Sometimes it is necessary to perform signature verification in smart contracts to achieve better usability or to save gas cost. All secure implementation needs to protect against Signature Replay Attacks by for example keeping track of all processed message hashes and only allowing new message hashes to be processed. A malicious user could attack a contract without such a control and get message hash that was sent by another user processed multiple times.	Store every message hash that has been processed by the smart contract and when new messages are received check against the already existing ones and only proceed with the business logic if it's a new message hash. Include the address of the contract that processes the message. This ensures that the message can only be used in a single contract.
SWC-122	Lack of Proper Signature Verification	It is a common pattern for smart contract systems to allow users to sign messages off-chain instead of directly requesting users to do an on-chain transaction because of the flexibility and increased transferability that this provides. Smart contract systems that process signed messages must implement their own logic to recover the authenticity from the signed messages before they process them further. Otherwise, they can lead to vulnerabilities especially in scenarios where proxies can be used to relay transactions.	Only methods that require proper signature verification through *ecrecover()* must be used.

SWC ID	Title	Description	Prevention
SWC-123	Requirement Violation	require() construct is meant to validate external inputs of a function. This function can be violated due to a bug in the contract that provided the external input, or the condition used to express the requirement is too strong.	If the required logical condition is too strong, it should be weakened to allow all valid external inputs. Otherwise, the bug must be in the contract that provided the external input and the code must be updated to ensure that all inputs provided are accepted.
SWC-124	Write to Arbitrary Storage Location	The contract is responsible for ensuring that only authorized user or contract accounts may write to sensitive storage locations. If an attacker can write to arbitrary storage locations of a contract, the authorization checks may easily be circumvented. This can allow an attacker to corrupt the storage; for instance, by overwriting a field that stores the address of the contract owner.	Must be written in one data structure that cannot inadvertently overwrite entries of another data structure.
SWC-125	Incorrect Inheritance Order	Solidity supports multiple inheritance, meaning that one contract can inherit several contracts. Since that, if two or more base contracts define the same function, the inheritance in the main contract must be defined correctly to know which function must be used.	The inheritance must be specified in the correct order. The rule of thumb is to inherit contracts from more /general/ to more /specific/.
SWC-126	Insufficient Gas Griefing	Insufficient gas griefing attacks can be performed on contracts which accept data and use it in a sub-call on another contract. If the sub-call fails, either the whole transaction is reverted, or execution is continued. In the case of a relayer contract, the user who executes the transaction, the 'forwarder', can effectively censor transactions by	Only allow trusted users to relay transactions and require that the forwarder provides enough gas.

SWC ID	Title	Description	Prevention
		using just enough gas to execute the transaction, but not enough for the sub-call to succeed.	
SWC-127	Arbitrary Jump with Function Type Variable	Solidity can support several function types, however sometimes it can lead in security vulnerabilities, for example if the developer uses assembly instructions, such as mstore or assign operator, an attacker can be able to point a function type variable to any code instruction, violating required validations and required state changes.	The use of assembly should be minimal. The assignment of the arbitrary values to function type variables must not be allowed.
SWC-128	~~DoS~~ With Block Gas Limit	When smart contracts are deployed or functions inside them are called, the execution of these actions always requires a certain amount of gas (based on how much computation is needed to complete them). The Ethereum network specifies a block gas limit and the sum of all transactions included in a block cannot exceed the threshold. If this limit is reached it can lead in security vulnerabilities such as DoS.	When it is expected a large array that grow over time looping across the entire data structure should be avoided. If it is real needed, then must be taken multiple blocks and therefore require multiple transactions.
SWC-129	Typographical Error	A typographical error can occur for example when the intent of a defined operation is to sum a number to a variable (+=) but it has accidentally been defined in a wrong way (=+), introducing a typo which happens to be a valid operator. Instead of calculating the sum it initializes the variable again provoking and unexpected behaviour.	Must be performed pre-condition checks on any math operation or using a vetted library for arithmetic calculations such as SafeMath.
SWC-130	Right-To-Left-Override control character (U+202E)	The usage of Right-To-Left-Override Unicode character can be used by malicious user to force	This character must be not used in the source code of the smart contract.

SWC ID	Title	Description	Prevention
		RTL text rendering and confuse users as to the real intent of a contract.	
SWC-131	Presence of unused variables	The unused variables cannot lead to security vulnerabilities; however, they must be avoided because they cause an increase in computations, indicate bugs or malformed data structures and they are generally a sign of poor code quality and cause code noise and decrease readability of the code.	All unused variables must be removed from the code base.
SWC-132	Unexpected Ether balance	Contracts can behave erroneously when they strictly assume a specific Ether balance. They must be configured otherwise; in the worst-case scenario this could lead to DoS conditions that might render the contract unusable.	Strict equality checks for the Ether balance in a contract must be avoided.
SWC-133	Hash Collisions with Multiple Variable Length Arguments	Using abi.encodePacked() with multiple variable length arguments can, in certain situations, lead to a hash collision. Since it packs all elements in order regardless of whether they're part of an array, you can move elements between arrays and, so long as all elements are in the same order, it will return the same encoding. In a signature verification situation, an attacker could exploit this by modifying the position of elements in a previous function call to effectively bypass authorization.	Alternatively, abi.encode() must be usedinstead.
SWC-134	Message call with hardcoded gas amount	The transfer() and send() functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions	The use of transfer() and send() must be avoided and do not otherwise specify a fixed amount of gas when performing calls. Use .call.value(...)("") instead.

SWC ID	Title	Description	Prevention
		may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs.	
SWC-135	Code With No Effects	It is possible to write code that does not produce the intended effects. Currently, the solidity compiler will not return a warning for effect-free code. This can lead to the introduction of "dead" code that does not properly perform an intended. Even if it is not a security vulnerability it can lead to one.	Unit tests must be written to ensure that the code behaves as expected.
SWC-136	Unencrypted Private Data On-Chain	It is a common misconception that private type variables cannot be read. Even if your contract is not published, attackers can look at contract transactions to determine values stored in the state of the contract. For this reason, it's important that unencrypted private data is not stored in the contract code or state.	Every private data must be stored off-chain or carefully encrypted.

Annex B. Relevant Paper's Result

Search Engine	Search String	Number of papers before exclusion criteria	Number of Papers selected
IEEE Xplore	((("Abstract":solidity) AND ("Abstract":security) AND ("Abstract":vulnerabilities) AND ("Abstract":"static analysis" OR "LLM")))	10	7
ACM Library	[Abstract: solidity] AND [Abstract: security] AND [Abstract: vulnerabilities] AND [[Abstract: static analysis] OR [Abstract: llm]]	13	5
Science Direct	solidity AND security AND vulnerabilities AND ("static analysis" OR LLM)	162	15

Annex C. Slither Results

File Name	Reentrancy	Access control	TP	TN	FP	FN	ExecutionTime
0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f.sol	x		1	0	0	0	2,2535
0x23a91059fdc9579a9fbd0edc5f2ea0bfdb70deb4.sol	x		1	0	0	0	2,2545
0x4320e6f8c05b27ab4707cd1f6d5ce6f3e4b3a5a1.sol	x		1	0	0	0	2,2359
0x4e73b32ed6c35f570686b89848e5f39f20ecc106.sol	x		1	0	0	0	2,2392
0x561eac93c92360949ab1f1403323e6db345cbf31.sol	x		1	0	0	0	2,2388
0x627fa62ccbb1c1b04ffaecd72a53e37fc0e17839.sol	x		1	0	0	0	2,2542
0x7541b76cb60f4c60af330c208b0623b7f54bf615.sol	x		1	0	0	0	2,273
0x7a8721a9d64c74da899424c1b52acbf58ddc9782.sol	x		1	0	0	0	2,241
0x7b368c4e805c3870b6c49a3f1f49f69af8662cf3.sol	x		1	0	0	0	2,2895
0x8c7777c45481dba411450c228cb692ac3d550344.sol	x		1	0	0	0	2,2343
0x93c32845fae42c83a70e5f06214c8433665c2ab5.sol	x		1	0	0	0	2,277
0x941d225236464a25eb18076df7da6a91d0f95e9e.sol	x		1	0	0	0	2,2494
0x96edbe868531bd23a6c05e9d0c424ea64fb1b78b.sol	x		1	0	0	0	2,2489
0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8.sol	x		1	0	0	0	2,2361
0xb5e1b1ee15c6fa0e48fce100125569d430f1bd12.sol	x		1	0	0	0	2,2329
0xb93430ce38ac4a6bb47fb1fc085ea669353fd89e.sol	x		1	0	0	0	2,2529

File Name	Reentrancy	Access control	TP	TN	FP	FN	ExecutionTime
0xbaf51e761510c1a11bf48dd87c0307ac8a8c8a4f.sol	x		1	0	0	0	2,2349
0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888.sol	x		1	0	0	0	2,2368
0xcead721ef5b11f1a7b530171aab69b16c5e66b6e.sol	x		1	0	0	0	2,2669
0xf015c35649c82f5467c9c74b7f28ee67665aad68.sol	x		1	0	0	0	2,295
arbitrary_location_write_simple.sol		x	1	0	0	0	2,2673
ElonCEO.sol	detected		0	0	1	0	3,1829
etherbank.sol	x		1	0	0	0	0,7938
etherstore.sol	x		1	0	0	0	0,8292
FibonacciBalance.sol		x	1	0	0	0	2,3104
GodMode.sol			0	1	0	0	3,8083
incorrect_constructor_name1.sol		x	0	0	0	1	2,3097
incorrect_constructor_name2.sol		x	0	0	0	1	2,2684
incorrect_constructor_name3.sol		x	0	0	0	1	2,2644
koalacoin.sol			0	1	0	0	2,965
mapping_write.sol		x	0	0	0	1	2,27
Metacons.sol			0	1	0	0	3,3029
modifier_reentrancy.sol	x		1	0	0	0	2,2839
multiowned_vulnerable.sol		x	0	0	0	1	2,4044
mycontract.sol		x	1	0	0	0	2,3093

File Name	Reentrancy	Access control	TP	TN	FP	FN	ExecutionTime
parity_wallet_bug_1.sol		x	1	0	0	0	0,9588
parity_wallet_bug_2.sol		x	1	0	0	0	0,9384
phishable.sol		x	1	0	0	0	2,2242
proxy.sol		x	1	0	0	0	2,2508
reentrance.sol	x		1	0	0	0	2,2169
reentrancy_bonus.sol	x		-	-	-	-	-
reentrancy_cross_function.sol	x		-	-	-	-	-
reentrancy_dao.sol	x		1	0	0	0	2,2306
reentrancy_insecure.sol	x		1	0	0	0	2,2675
reentrancy_simple.sol	x		1	0	0	0	0,8441
rubixi.sol		x	0	0	0	1	0,9126
SantaPepe.sol			0	1	0	0	3,1174
ShinChan.sol			0	1	0	0	1,0656
simple_dao.sol	x		1	0	0	0	0,8158
simple_suicide.sol		x	1	0	0	0	0,7957
spank_chain_payment.sol	x		1	0	0	0	4,6853
unprotected0.sol		x	1	0	0	0	0,8357
wallet_02_refund_nosub.sol		x	0	0	0	1	2,2898
wallet_03_wrong_constructor.sol		x	0	0	0	1	2,3003

File Name	Reentrancy	Access control	TP	TN	FP	FN	ExecutionTime
wallet_04_confused_sign.sol		x	0	0	0	1	2,283
zaibot.sol	detected		0	0	1	0	3,5029

Annex D. Codellama Results

File Name	Reentrancy	Acess control	TP	TN	FP	FN	Execution Time
0x01f8c4e3fa3edeb29e514cba738d87ce8c091d3f_refined.sol	x		1	0	0	0	42,0816
0x23a91059fdc9579a9fbd0edc5f2ea0bfdb70deb4_refined.sol	x		0	0	0	1	18,6381
0x4320e6f8c05b27ab4707cd1f6d5ce6f3e4b3a5a1_refined.sol	x		1	0	0	0	37,3257
0x4e73b32ed6c35f570686b89848e5f39f20ecc106_refined.sol	x		1	0	0	0	39,4814
0x561eac93c92360949ab1f1403323e6db345cbf31_refined.sol	x		1	0	0	0	34,8601
0x627fa62ccbb1c1b04ffaecd72a53e37fc0e17839_refined.sol	x		0	0	0	1	40,4562
0x7541b76cb60f4c60af330c208b0623b7f54bf615_refined.sol	x		1	0	0	0	43,5914
0x7a8721a9d64c74da899424c1b52acbf58ddc9782_refined.sol	x		0	0	0	1	28,0644
0x7b368c4e805c3870b6c49a3f1f49f69af8662cf3_refined.sol	x		1	0	0	0	47,1403
0x8c7777c45481dba411450c228cb692ac3d550344_refined.sol	x		1	0	0	0	48,5585
0x93c32845fae42c83a70e5f06214c8433665c2ab5_refined.sol	x		1	0	0	0	56,621
0x941d225236464a25eb18076df7da6a91d0f95e9e_refined.sol	x		0	0	0	1	57,5721
0x96edbe868531bd23a6c05e9d0c424ea64fb1b78b_refined.sol	x		1	0	0	0	63,1584
0xaae1f51cf3339f18b6d3f3bdc75a5facd744b0b8_refined.sol	x		1	0	0	0	43,1106
0xb5e1b1ee15c6fa0e48fce100125569d430f1bd12_refined.sol	x		1	0	0	0	12,2078
0xb93430ce38ac4a6bb47fb1fc085ea669353fd89e_refined.sol	x		1	0	0	0	56,2257
0xbaf51e761510c1a11bf48dd87c0307ac8a8c8a4f_refined.sol	x		1	0	0	0	43,0989
0xbe4041d55db380c5ae9d4a9b9703f1ed4e7e3888_refined.sol	x		1	0	0	0	42,4799
0xcead721ef5b11f1a7b530171aab69b16c5e66b6e_refined.sol	x		1	0	0	0	45,7508
0xf015c35649c82f5467c9c74b7f28ee67665aad68_refined.sol	x		1	0	0	0	23,5206
arbitrary_location_write_simple_refined.sol		x	1	0	0	0	41,6412
ElonCEO_refined.sol			0	1	0	0	48,9581
etherbank_refined.sol	x		1	0	0	0	52,596

File Name	Reentrancy	Acess control	TP	TN	FP	FN	Execution Time
etherstore_refined.sol	x		1	0	0	0	36,78
FibonacciBalance_refined.sol		x	1	0	0	0	905,269
GodMode_refined.sol			0	1	0	0	74,25
incorrect_constructor_name1_refined.sol		x	1	0	0	0	30,5362
incorrect_constructor_name2_refined.sol		x	1	0	0	0	29,1268
incorrect_constructor_name3_refined.sol		x	0	0	0	1	12,4091
koalacoin_refined.sol			0	1	0	0	35,9964
mapping_write_refined.sol		x	0	0	0	1	40,2121
Metacons_refined.sol	detected		0	0	1	0	38,0687
modifier_reentrancy_refined.sol	x		1	0	0	0	37,5744
multiowned_vulnerable_refined.sol		x	1	0	0	0	32,4882
mycontract_refined.sol		x	1	0	0	0	13,6151
parity_wallet_bug_1_refined.sol		x	0	0	0	1	94,289
parity_wallet_bug_2_refined.sol		x	1	0	0	0	67,4542
phishable_refined.sol		x	0	0	0	1	36,6646
proxy_refined.sol		x	1	0	0	0	28,8295
reentrance_refined.sol	x		1	0	0	0	5,9975
reentrancy_bonus_refined.sol	x		-	-	-	-	-
reentrancy_cross_function_refined.sol	x		-	-	-	-	-
reentrancy_dao_refined.sol	x		1	0	0	0	20,6626
reentrancy_insecure_refined.sol	x		1	0	0	0	25,7052
reentrancy_simple_refined.sol	x		1	0	0	0	20,634
rubixi_refined.sol		x	1	0	0	0	77,8543
SantaPepe_refined.sol	detected		0	0	1	0	48,814
ShinChan_refined.sol	detected		0	0	1	0	61,9179

File Name	Reentrancy	Acess control	TP	TN	FP	FN	Execution Time
simple_dao_refined.sol	x		1	0	0	0	18,2717
simple_suicide_refined.sol		x	0	0	0	1	21,4479
spank_chain_payment_refined.sol	x		0	0	0	1	97,6033
unprotected0_refined.sol		x	1	0	0	0	11,8854
wallet_02_refund_nosub_refined.sol		x	0	0	0	1	25,7221
wallet_03_wrong_constructor_refined.sol		x	1	0	0	0	40,8968
wallet_04_confused_sign_refined.sol		x	1	0	0	0	37,3073
zaibot_refined.sol			0	1	0	0	46,3617