University of Padova

Department of Civil, Environmental and Architectural Engineering

Master Thesis in Mathematical Engineering

The Role of Large Language Models in Detecting and Patching Smart Contract Vulnerabilities

Supervisor
Prof. Vinod Puthuvath
University of Padova

Co-supervisor
Prof. Mauro Conti
University of Padova

Master Candidate
Leonardo Reyes Cardenas

Student ID
2023620

Academic Year
2023-2024

I am truly grateful for the opportunity to work on this thesis under the guidance of Prof. Vinod Puthuvath and Prof. Mauro Conti. Their invaluable insights and feedback have been fundamental not only in shaping this work, but also in helping me understand and appreciate the nuances of scientific research.

I also sincerely thank my family and friends for their constant support and encouragement throughout this journey. I will always cherish the skills I've developed and the friendships I've built across the globe.

Abstract

Smart contracts are self-executing programs that run on blockchain platforms like Ethereum, enabling decentralized applications and services. While they offer significant advantages in terms of automation and trustlessness, smart contracts are susceptible to vulnerabilities that can lead to financial losses and security breaches. Traditional methods for detecting and patching these vulnerabilities often require extensive manual effort and specialized expertise.

This thesis explores the use of Large Language Models to automate the detection and remediation of vulnerabilities in Solidity smart contracts. We leveraged a substantial labelled dataset of smart contracts and performed data cleaning and preprocessing to extract individual functions. Using prompt engineering techniques, we guided the LLMs to analyze both integral smart contracts and individual functions, identify potential vulnerabilities, and generate patched versions of the code. The models were evaluated based on their ability to accurately detect vulnerabilities and produce effective patches that preserve the original functionality of the smart contracts.

The findings suggest that despite some limitations, these models hold promise for enhancing smart contract security. This approach could significantly reduce the time and expertise required to secure smart contracts, making blockchain technology more robust and accessible. However, further research is needed to address challenges related to model accuracy, reliability, and the handling of complex vulnerabilities.

Contents

Listing of figures

Listing of tables

Listing of acronyms

AI ..... Artificial Intelligence

API ..... Application Programming Interface

CPU ..... Central Processing Unit

CSV ..... Comma-Separated Values

DApp ..... Decentralized Application

DAO ..... Decentralized Autonomous Organization

DLT ..... Distributed Ledger Technology

DoS ..... Denial of Service

ETH ..... Ether, Ethereum's cryptocurrency

EVM ..... Ethereum Virtual Machine

FN ..... False Negative

FP ..... False Positive

GPU ..... Graphics Processing Unit

GPT ..... Generative Pre-trained Transformer

HTTP ..... Hypertext Transfer Protocol

I/O ..... Input/Output

ICO ..... Initial Coin Offering

IDE ..... Integrated Development Environment

JSON ..... JavaScript Object Notation

LLM ..... Large Language Model

ML ..... Machine Learning

NLP . . . . . Natural Language Processing

OS . . . . . Operating System

OWASP . . . . . Open Web Application Security Project

P2P . . . . . Peer-to-Peer

RAM . . . . . Random Access Memory

REST . . . . . Representational State Transfer

SC . . . . . Smart Contract

SDK . . . . . Software Development Kit

TP . . . . . True Positive

VM . . . . . Virtual Machine

XML . . . . . eXtensible Markup Language

1 Introduction

Distributed Ledger Technologies (DLTs) encompass a family of technologies designed to enable the decentralized recording, sharing, and synchronization of data across a network of participants. Unlike traditional centralized databases, which rely on a single authoritative entity for managing and storing data, DLTs distribute data among multiple nodes, each one typically holds a complete copy of the ledger. Updates to the ledger are achieved through consensus mechanisms, which ensure that network participants collectively agree and validate transactions without the need for a central authority. Blockchain, a specific and well-known type of DLT, organizes data into sequential blocks that are cryptographically linked to form a chain. This structure guarantees data integrity and immutability, as altering the contents of a block would require not only modifying all subsequent blocks but also taking over the consensus of the network. Every transaction recorded on a blockchain is visible to authorized participants, depending on whether the blockchain is public or private, enabling traceability and audit trails making it particularly appealing for various applications. Additionally, the immutability of blockchain ensures that once data is recorded, it is highly resistant to tampering or deletion, which is critical for maintaining trust in sensitive applications.

This technological foundation has enabled blockchain to be adopted across diverse domains. In finance, it underpins cryptocurrencies like Bitcoin, providing secure and decentralized digital currency systems. In supply chain management, blockchain facilitates end-to-end traceability of goods, offering a transparent and tamper-proof record of transactions. In healthcare, it is

used to securely manage and share patient records while preserving privacy and enabling interoperability. However, one of the most transformative innovations enabled by blockchain technology is the advent of smart contracts, which are self-executing agreements with their terms directly encoded into software. First implemented on the Ethereum blockchain [1], where they are written in the Solidity programming language, the number of smart contracts deployed have grown massively since their conception.

These contracts operate autonomously, automatically enforcing and executing their stipulated actions when predefined conditions are met. By eliminating the need for intermediaries or third-party enforcement, smart contracts significantly streamline processes, reducing both transaction costs and the potential for human error. This capability not only accelerates the execution of agreements but also enhances trust, as the transparent and immutable nature of the blockchain ensures that the contract operates exactly as programmed without the risk of tampering or dispute over its terms. Blockchain platforms like Ethereum, which supports Turing-complete programming languages, allow for the creation of highly complex, conditional agreements that go beyond simple transactional functions. For instance, smart contracts can manage multi-party workflows, execute sophisticated financial instruments, or govern decentralized autonomous organizations (DAOs). Despite their transformative potential, smart contracts have introduced significant security challenges [2], as their immutable and transparent nature makes them uniquely vulnerable to exploitation. A single flaw in the code of a deployed smart contract can have irreversible consequences, leading to substantial financial losses and undermining trust in blockchain systems.

The DAO, a decentralized autonomous organization built on Ethereum in 2016 [3], aimed to create a new business model for organizing both commercial and non-profit enterprises through collective decision-making and investment. It was launched after one of the largest crowdfunding campaigns in history. The DAO had no conventional management structure or board of directors, relying instead on open-source smart contract code to govern its operations. However, in June 2016, attackers exploited a flaw in this code, diverting one-third of The DAO's funds (approximately 50 million USD worth of Ether) into a subsidiary account, due to a reentrancy vulnerability in the smart contract code [3]. This vulnerability allowed attackers to repeatedly call a withdrawal function before the contract's balance was updated, effectively draining funds. The incident led to a controversial hard fork of the Ethereum blockchain, restoring the stolen funds to their original contract but splitting the blockchain into two ver-

sions: “Ethereum” and “Ethereum Classic”. By September 2016, The DAO’s value token was delisted from major exchanges, rendering the project effectively defunct. The DAO is not an isolated case. Security flaws in smart contracts continue to result in significant losses across the blockchain ecosystem. For instance, on August 7th 2024, the DeFi protocol Nexera was hacked [4], with 1.5 million USD worth of NXRA tokens siphoned through an exploit. Attackers took ownership of a proxy contract, upgraded it, and used administrative functions to withdraw funds. Such incidents are not anomalies but part of a broader pattern of vulnerabilities in DeFi protocols and smart contracts, which collectively have resulted in losses amounting to billions of dollars.

The immutable nature of blockchain transactions amplifies the consequences of these vulnerabilities. Once a smart contract is deployed, its code cannot be altered without achieving consensus across the network. This immutability makes pre-deployment security audits indispensable. Without these safeguards, the very features that make blockchain technology revolutionary become liabilities, undermining the trust they are designed to inspire. Identifying vulnerabilities requires either manual auditing by skilled professionals (a time-consuming and resource-intensive process) or the use of specialized automated tools, which are limited in their capabilities and unable to detect all potential flaws.

Smart contract developers need to understand potential vulnerabilities and take strong security measures to reduce risks. Manual code reviews, while detailed, can be slow and prone to mistakes, especially for smart contracts with complex logic. Similarly, static analysis tools, which automate the search for known issues, often struggle to detect new or more subtle security problems. As smart contracts become more important in many industries, there is a growing need for better and faster ways to analyze their security. Language models offer a promising new way to improve vulnerability detection by using advancements in Natural Language Processing (NLP) and Machine Learning (ML). These models can read code, find patterns that suggest vulnerabilities, and make predictions based on what they’ve learned. Recent developments in Large Language Models (LLMs) have created powerful tools that are now being explored for use in security analysis. However, LLMs face challenges, such as requiring large and diverse datasets, risks of biased predictions, and difficulties in understanding how the models make decisions. Overcoming these challenges is essential to fully realize their potential in detecting vulnerabilities. This study aims to evaluate the effectiveness of LLMs can in smart contract security. By comparing their performance with traditional methods using standard datasets,

the research will highlight the strengths and weaknesses of LLM-based vulnerability analysis for blockchain applications. Following are the key research questions about LLMs accuracy, reliability, and potential as an alternative or complement to existing methods.

- **RQ1: How effective are LLMs in detecting smart contract vulnerabilities?**
  We assessed the performance of three state-of-the-art LLMs: GPT-3.5-turbo, GPT-01-mini, and Gemini-1.5-flash-8b in identifying vulnerabilities in Solidity smart contract code. Utilizing the widely adopted ScrawlD dataset (`https://github.com/sujeetc/ScrawlD`), we conducted extensive experiments to evaluate the detection capabilities of these models across different types of vulnerabilities.

- **RQ2: How do the patches generated by LLMs perform against traditional vulnerability detection tools?**
  Beyond detection, we explored the capability of LLMs to generate patches for identified vulnerabilities, and assessed the effectiveness of these remediations using traditional vulnerability detection tools (Mythril, SmartCheck, Osiris, and Oyente).

- **RQ3: How does character obfuscation affect the vulnerability detection capabilities of LLMs and conventional detection tools?**
  To evaluate the robustness of LLMs against character-level obfuscation techniques, we performed additional experiments where variables in the smart contracts were obfuscated. This assessment focused on GPT-3.5-turbo and Gemini-1.5-flash-8b. Our study indicates that character obfuscation has a minimal impact on LLM vulnerability detection performance.

Organization: This project report is structured into five chapters. Chapter 2 presents the background, including blockchain, smart contracts, and their vulnerabilities. Chapter 3 offers a literature review focusing on vulnerability detection using machine learning and Large Language Models (LLMs). Chapter 4 outlines the methodology employed for this research. The experiments and their results are detailed in Chapter 5, while Chapter 6 concludes the study and provides recommendations for future research.

2
Background

Smart contracts are a cornerstone of blockchain technology, enabling automated and secure execution of agreements between parties without intermediaries. In this chapter, we explore the mechanics of smart contracts, highlight common vulnerabilities, introduce the Large Language Models (LLMs) used in this study, and describe traditional tools for vulnerability detection.

2.1 Smart Contracts

A smart contract is a self-executing program stored and executed on a blockchain. It operates under predefined rules and conditions, automatically enforcing or executing certain actions when triggered. Smart contracts eliminate the need for intermediaries, ensuring operations are transparent, immutable, and tamper-proof. When users interact with a smart contract, they do so through blockchain transactions. These transactions invoke the contract's functions, which can handle various tasks, such as transferring cryptocurrency, managing digital assets, or implementing complex workflows. To illustrate, Figure 2.1 is an example of a basic Solidity smart contract for a banking application. This contract allows users to deposit Ether, withdraw funds, and check their balances. More specifically, the deposit function allows users to send Ether to the contract, which is recorded in the balances mapping associated with their address. The withdraw function lets users withdraw Ether from their balance, provided they have sufficient funds. It checks the user's balance before proceeding and transfers the requested amount

back to the user. The `getBalance` function is a read-only function that enables users to view their current balance in the contract. Although this example provides basic functionality, it also highlights the importance of careful implementation. If not carefully designed, smart contracts can be vulnerable to issues such as reentrancy attacks, where malicious actors repeatedly call a function to manipulate balances, or access control problems, where unauthorized users gain privileges they should not have.

As we mentioned previously, the convenience of these tools have made decentralized applications (dApps) extremely popular. Just forcusing on the DeFi (Decentralized Finance) applications, the global smart contracts market size was reported to be 1.71 billion USD in 2023, and it is expected to be worth 12.55 billion USD by 2032 [5]. This growing trend is represented in Figure 2.2, and along with the considerations on the software vulnerabilities, it becomes clear how smart contract security will be of upmost importance for the coming years. In the next section, we explore smart contracts' vulnerabilities in detail.

2.2 Smart Contracs' Vulnerabilities

Smart contracts are prone to a range of vulnerabilities that can result in significant financial losses. Below, we detail the Open Worldwide Application Security Project (OWASP) Top 10 smart contract vulnerabilities studied in this work:

- **SC01 - Reentrancy Attacks:** Reentrancy attacks exploit vulnerabilities in smart contracts where an external call is made before the contract's state is updated. This allows a potentially malicious external contract to repeatedly invoke the original function, often enabling unauthorized actions such as multiple withdrawals, which can result in draining the contract's funds. An example of this is reported in Figure 2.3, the vulnerability lies in the external call to `createTokenProxy` being made before the contract's internal state is updated. Since the balance of `msg.sender` and the `totalSupply` are only updated later in the function, a malicious actor could repeatedly exploit the same state to withdraw funds multiple times, allowing the attacker to drain more funds than they are entitled to.
- **SC02 - Integer Overflow and Underflow:** The Ethereum Virtual Machine (EVM) uses fixed-size integers, limiting the range of representable values. Overflow occurs when a value exceeds the maximum limit of its type, while underflow happens when it drops below the minimum. For unsigned integers, underflow causes a wraparound to the maximum value, whereas for signed integers, it cycles to the highest positive value. An example of this is reported in Figure 2.4, the vulnerability lies in the possibility of integer overflow, particularly in the `extendLockTime` function. Here, "`additionalSeconds`"

is added to "`withdrawalUnlockTime[msg.sender]`" without any checks to ensure that the result stays within the valid range of a `uint256`. If a user passes a very large value for "`additionalSeconds`", the arithmetic operation could overflow, causing the `withdrawalUnlockTime` to wrap around to a much smaller number or even zero. This effectively bypasses the intended lock mechanism, and an attacker could artificially shorten their lock time, allowing them to withdraw funds without respecting the intended delay.

- **SC03 - Timestamp Dependence:** Smart contracts frequently rely on *block.timestamp* for time-sensitive operations. However, miners can slightly manipulate this value, potentially exploiting it to influence contract behavior and gain an advantage. An example of this is reported in Figure 2.5, the vulnerability stems from the use of *block.timestamp* in the `rollDice` function to determine the outcome of the dice roll. As we mentioned, miners can be slightly adjust this value within a range of approximately 15 seconds. This means that a malicious miner could alter the timestamp to meet specific conditions, allowing them to unfairly win the game and withdraw the contract's entire balance.

- **SC04 - Access Control Vulnerabilities:** These flaws occur when contracts fail to enforce proper restrictions on user permissions, allowing unauthorized entities to access or modify contract data and functions, compromising security. An example of this is reported in Figure 2.6, the vulnerability arises from the `burn` function. Specifically, the function is declared as public and does not include any checks to ensure that only authorized users can call it. This means that any user can call the `burn` function and arbitrarily destroy tokens. Attackers could maliciously reduce the token balances of legitimate users or disrupt the token's overall supply, undermining trust and destabilization the ecosystem. Moreover, the contract's governance or critical functions, such as pausing or upgrading, could be permanently compromised.

- **SC05 - Front-running Attacks:** Front-running involves exploiting knowledge of pending transactions. Attackers monitor the *mempool* and submit their own transactions with higher gas fees to prioritize execution, potentially disrupting operations and causing financial losses. An example of this is reported in Figure 2.7, here the vulnerability lies in the absence of slippage protection in the `swapBNBForSSToken` function. Slippage refers to the difference between the expected price of a token trade and the actual price at which it is executed. Since the function does not include a minimum output amount check, an attacker can exploit this by front-running the transaction. Front-running occurs when an attacker observes a large swap transaction and submits their own transaction with a higher gas fee to ensure it is processed first. By doing so, the attacker can execute a trade that alters the market conditions just before the victim's transaction is processed. As a result, the victim receives significantly fewer tokens than expected or pays a much higher price for them.

- **SC06 - Denial of Service (DoS) Attacks:** DoS attacks target vulnerabilities within

smart contracts to deplete critical resources like gas, computational power, or storage. This disruption can render contracts inoperable and result in financial or operational harm. An example of this is reported in Figure 2.8, the vulnerability here lies in the logic of the `claimThrone` function, where the contract attempts to send Ether to the current king before updating the king and balance variables. Specifically, if the current king's address points to a malicious contract with a fallback function that intentionally reverts the transaction, the entire `claimThrone` function will fail. This prevents new users from successfully claiming the throne, locking the contract in a state where no one else can become the new king.

- **SC07 - Logic Errors:** Logic errors, or business logic vulnerabilities, are flaws where the implementation deviates from the intended behavior of the smart contract. These errors are often subtle and difficult to detect but can lead to exploitable or unintended outcomes. An example of this is reported in Figure 2.9, the vulnerability arises from a logic error in the withdraw function, it reduces the user's balance but fails to update the `totalLendingPool` variable to reflect the withdrawal. This inconsistency introduces can lead to the `totalLendingPool` value no longer accurately represent the total amount of funds held by the contract. This could mislead other users or external systems interacting with the contract.

- **SC08 - Insecure Randomness:** Due to the deterministic nature of blockchain networks, generating truly random values is challenging. Predictable or manipulable randomness can allow attackers to exploit contracts, undermining fairness and security. An example of this is reported in Figure 2.10, the random number is derived using block-specific attributes such as *block.timestamp*, *block.difficulty*, and `msg.sender`, combined with the `keccak256` hashing function. While the resulting value may appear random, these block properties are not truly unpredictable. Miners, who have control over certain block parameters can manipulate them to produce favorable outcomes. This insecurity is particularly problematic in applications where fairness and unpredictability are critical, such as lotteries, gambling games, or prize distributions.

- **SC09 - Gas Limit Vulnerabilities:** Gas limits restrict the computational resources available for a transaction. Contracts using resource-intensive operations, such as loops over large data structures, risk exceeding these limits, causing transaction failure and potential vulnerabilities in design. An example of this is reported in Figure 2.11, here the vulnerability arises from a gas limit in the transfer function. Specifically, the function includes a for-loop that iterates "amount" times to decrement the sender's balance and increment the recipient's balance. This approach becomes problematic if "amount" is very large, as the number of iterations required may exceed the Ethereum block gas limit. When this happens, the transaction will fail, leaving the funds effectively inaccessible and rendering the contract unusable for such transfers.

- **SC10 - Unchecked External Calls:** If a smart contract fails to verify the outcome of an external function call, it may proceed with operations under the false assumption of success. This oversight can lead to integrity issues and unintended behavior within the contract. An example of this is reported in Figure 2.12, here the vulnerability lies in the forward function, specifically the `callee.delegatecall` statement. "`delegatecall`" does not automatically revert the transaction if it fails; instead, it returns a boolean value indicating success or failure. By not checking this return value, the contract incorrectly assumes that the external call was successful, even if it silently failed. This can result in inconsistent contract states or incomplete operations.

2.3 Large Language Models

Large Language Models (LLMs) represent a significant advancement in the field of artificial intelligence, specifically in natural language processing (NLP) [6]. These models are trained on vast amounts of textual data, enabling them to understand and generate human-like language with remarkable precision. At their core, LLMs rely on transformer architectures, which utilize self-attention mechanisms to process and contextualize information across large sequences of text. This architecture allows the model to capture relationships between words, phrases, and even entire paragraphs, enabling nuanced understanding and contextual reasoning. Furthermore, LLMs can be fine-tuned to perform a variety of tasks, ranging from natural language understanding to more specialized applications.

In the context of vulnerability detection, LLMs offer unique advantages. First, their ability to process large amounts of text, including structured data like code [7], makes them ideal for analyzing smart contracts. Smart contracts often contain complex logic, and vulnerabilities can manifest in subtle ways that require a deep understanding of both the programming language and the intended functionality. LLMs might have an edge in this area, because of they might better grasp the semantics of code, recognizing patterns associated with vulnerabilities, and even suggest remediation strategies.

Unlike traditional rule-based analysis tools, LLMs can adapt to new and unseen scenarios. This flexibility is particularly valuable in the domain of blockchain and smart contract development, where the landscape is continually evolving, and new vulnerabilities emerge as the technology advances. In this study, three distinct LLMs were evaluated for their performance in detecting vulnerabilities in smart contracts:

- **GPT-3.5-Turbo** Developed by OpenAI and released in November 2022 [8]; it is characterised by a 16,385-tokens context window, 4,096 max output tokens and knowledge cutoff in September 2021.
- **GPT-o1-mini** Developed by OpenAI and released in September 2024[8]; it is characterised by a 128,000-tokens context window, 65,536 max output tokens and knowledge cutoff in October 2023.
- **Gemini 1.5 Flash 8B** Developed by Google and released in October 2024[9]; it is characterised by a 1,048,576-tokens context window, 8,192 max output tokens and unknown knowledge cutoff.

2.4 Traditional Vulnerability Detection Tools

Several analysis tools have been developed and employed for software security. The tools we employ in this study — Slither, Mythril, SmartCheck, Oyente, and Osiris — are widely recognized and extensively used for smart contract security analysis. These frameworks leverage techniques like static analysis, symbolic execution, and taint tracking to identify vulnerabilities in Ethereum smart contracts. Notably, Sujeet et al.[10], the creators of the dataset we rely upon for our analysis, have utilized these tools in their research, underscoring their relevance and reliability in the field.

- **Slither:** Slither is a fast and comprehensive static analysis tool for Solidity and Vyper smart contracts, designed to identify vulnerabilities, improve code comprehension, and enable custom analyses. Developed in Python, Slither runs a suite of vulnerability detectors, provides detailed contract insights through built-in printers, and offers a Python API for advanced customizations. It works by converting Solidity smart contracts into an intermediate format called SlithIR. SlithIR uses a simplified instruction set and Static Single Assignment (SSA) form, which makes it easier to analyze while keeping the important meaning of the original Solidity code that might be lost when converting to bytecode. Slither enables common program analysis methods, such as dataflow analysis and taint tracking, to be applied effectively. Slither is widely used for enhancing smart contract security and integrating into CI workflows.
  Link to GitHub: `https://github.com/crytic/slither`
- **Mythril:** Mythril is a versatile security analysis tool for Ethereum Virtual Machine (EVM) bytecode, used to detect vulnerabilities in smart contracts deployed on Ethereum and other EVM-compatible blockchains such as Hedera, Quorum, Vechain, Rootstock, and

Tron. It employs symbolic execution, SMT (Satisfiability Modulo Theories) solving, and taint analysis to identify security issues, including unprotected self-destructs, reentrancy attacks, and integer overflows.
Link to GitHub: https://github.com/ConsenSys/mythril

- **SmartCheck:** SmartCheck is a static analysis tool designed to detect vulnerabilities and code issues in Solidity smart contracts. By translating Solidity source code into an XML-based intermediate representation, SmartCheck enables efficient and systematic analysis through predefined XPath patterns. This approach facilitates the detection of a wide range of issues, from basic coding errors to more complex vulnerabilities, reflecting the current state of knowledge on Solidity security.
  Link to GitHub: https://github.com/smartdec/smartcheck

- **Oyente:** Oyente is a symbolic execution tool designed to detect vulnerabilities in Ethereum smart contracts by analyzing their behavior on the Ethereum Virtual Machine (EVM). By modeling the execution of smart contracts symbolically, Oyente can identify potential vulnerabilities that could be exploited.
  Link to GitHub: https://github.com/enzymefinance/oyente

- **Osiris:** Osiris is a framework built on top of Oyente, designed specifically to detect vulnerabilities related to integer bugs in Ethereum smart contracts, such as overflows, underflows, and improper arithmetic operations. Osiris combines symbolic execution with taint analysis to achieve this, symbolic execution enables the tool to explore potential execution paths in the contract, while taint analysis tracks how untrusted data propagates through the code.
  Link to GitHub: https://github.com/christoftorres/Osiris

Each tool offers distinct advantages and limitations, making their combined use a robust approach for vulnerability detection. By comparing LLM outputs with traditional tools, this study provides a holistic evaluation of smart contract security.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract SimpleBank {
    mapping(address => uint256) private balances;

    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        payable(msg.sender).transfer(amount);
    }

    function getBalance() external view returns (uint256) {
        return balances[msg.sender];
    }
}
```

Figure 2.1: Example of simple smart contract representing a banking application.

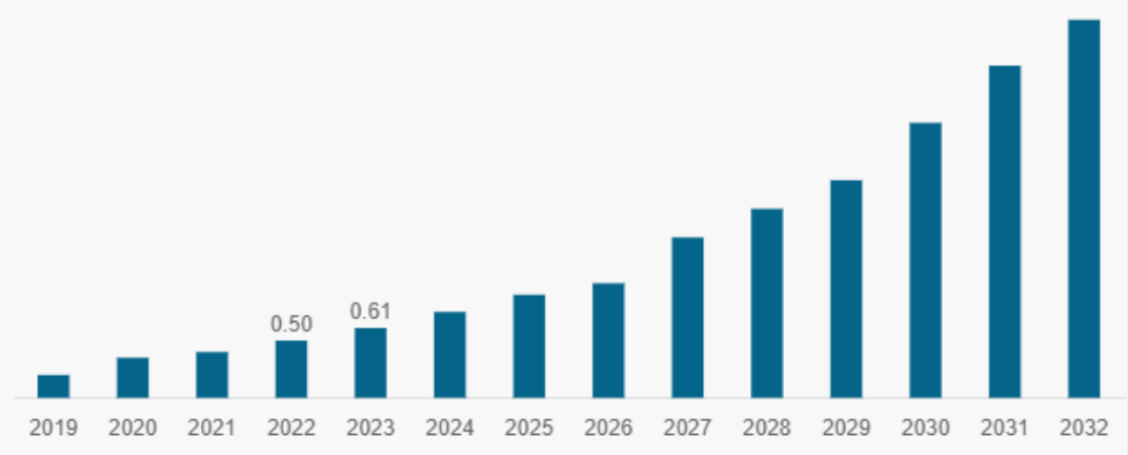

Figure 2.2: North America projected smart contracts' market size 2019-2032.

```
SC01
function splitDAO(uint _proposalID, address _newCurator) noEther
onlyTokenholders returns (bool _success) {
    ...

    uint fundsToBeMoved = (balances[msg.sender] *
p.splitData[0].splitBalance) / p.splitData[0].totalSupply;
    //Since the balance is never updated the attacker can pass this
modifier several times
    if
(p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMoved)(msg.sen
der) == false) throw;

    ...

    // Burn DAO Tokens
    // Funds are transferred before the balance is updated

    Transfer(msg.sender, 0, balances[msg.sender]);
    withdrawRewardFor(msg.sender); // be nice, and get his rewards
    // Only now after the funds are transferred is the balance
updated
    totalSupply -= balances[msg.sender];
    paidOut[msg.sender] = 0;
    return true;
}
```

Figure 2.3: Example of SC01 - Reentrancy Attacks.

```
SC02
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.0;

contract TimeWrapVault {
    mapping(address => uint256) public accountBalances;
    mapping(address => uint256) public withdrawalUnlockTime;

    // Allows anyone to deposit ETH and set a lock time
    function depositFunds() external payable {
        accountBalances[msg.sender] += msg.value;
        withdrawalUnlockTime[msg.sender] = block.timestamp + 1 weeks;
    }

    // Vulnerable to overflow, the user could pass a very large value
causing overflow
    function extendLockTime(uint256 _additionalSeconds) public {
        withdrawalUnlockTime[msg.sender] += _additionalSeconds;
    }

    // Allows withdrawals only if the current time is greater than
the lock time
    function releaseFunds() public {
        require(accountBalances[msg.sender] > 0, "Insufficient
funds");
        require(block.timestamp > withdrawalUnlockTime[msg.sender],
"Lock time not expired");

        uint256 withdrawalAmount = accountBalances[msg.sender];
        accountBalances[msg.sender] = 0;

        (bool successfulWithdrawal,) = msg.sender.call{value:
withdrawalAmount}("");
        require(successfulWithdrawal, "Failed to send Ether");
    }
}
```

Figure 2.4: Example of SC02 - Integer Overflow and Underflow.

```
SC03
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract DiceRoll {
    uint256 public lastBlockTime;

    constructor() payable {}

    function rollDice() external payable {
        require(msg.value == 5 ether, "Must send 5 ether to play");
// Player must send 5 ether to play
        require(block.timestamp != lastBlockTime, "Only 1 transaction
per block allowed"); // Ensures only 1 transaction per block

        lastBlockTime = block.timestamp;

        // Player wins if the last digit of the block timestamp is
less than 5
        if (block.timestamp % 10 < 5) {
            (bool sent,) = msg.sender.call{value:
address(this).balance}("");
            require(sent, "Failed to send Ether");
        }
    }
}
```

Figure 2.5: Example of SC03 - Timestamp Dependence.

```
SC04
function burn(address account, uint256 amount) public { //No proper
access control is implemented for the burn function
        _burn(account, amount);
    }
}
```

Figure 2.6: Example of SC04 - Access Control Vulnerabilities.

```
SC05
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract VulnerableSwap {
    address public pancakeRouter;
    address public ssToken;

    constructor(address _pancakeRouter, address _ssToken) {
        pancakeRouter = _pancakeRouter;
        ssToken = _ssToken;
    }

    function swapBNBForSSToken(uint256 amount) private {
        address[] memory path = new address[](2);
        path[0] = IPancakeRouter02(pancakeRouter).WETH();
        path[1] = ssToken;


IPancakeRouter02(pancakeRouter).swapExactETHForTokensSupportingFeeOnT
ransferTokens{
            value: amount
        }(0, path, address(this), block.timestamp);
    }
}
```

Figure 2.7: Example of SC05 - Front-running Attacks.

```
SC06
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

contract VulnerableKingOfEther {
    address public king;
    uint256 public balance;

    function claimThrone() external payable {
        require(msg.value > balance, "Need to pay more to become the
king");

        (bool sent,) = king.call{value: balance}("");
        require(sent, "Failed to send Ether");  // if the current
king's fallback function reverts, it will prevent others from
becoming the new king, causing a Denial of Service.

        balance = msg.value;
        king = msg.sender;
    }
}
```

Figure 2.8: Example of SC06 - Denial of Service (DoS) Attacks.

```
SC07
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract LendingPlatform {
    mapping(address => uint256) public userBalances;
    uint256 public totalLendingPool;

    function deposit() public payable {
        userBalances[msg.sender] += msg.value;
        totalLendingPool += msg.value;
    }

    function withdraw(uint256 amount) public {
        require(userBalances[msg.sender] >= amount, "Insufficient
balance");

        // Faulty calculation: Incorrectly reducing the user's
balance without updating the total lending pool
        userBalances[msg.sender] -= amount;

        // This should update the total lending pool, but it's
omitted here.

        payable(msg.sender).transfer(amount);
    }
}
```

Figure 2.9: Example of SC07 - Logic Errors.

```
SC08
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

contract InsecureRandomNumber {
    constructor() payable {}

    function guess(uint256 _guess) public {
        uint256 answer = uint256(
            keccak256(
                abi.encodePacked(block.timestamp, block.difficulty,
msg.sender) // Using insecure mechanisms for random number generation
            )
        );

        if (_guess == answer) {
            (bool sent,) = msg.sender.call{value: 1 ether}("");
            require(sent, "Failed to send Ether");
        }
    }
}
```

Figure 2.10: Example of SC08 - Insecure Randomness.

```
SC09
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract TokenTransfer {
    mapping(address => uint256) public balances;

    function transfer(address _to, uint256 _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient
balance");

        for (uint256 i = 0; i < _amount; i++) {  // The loop iterates
_amount times, which can be very inefficient and can potentially
exceed the block gas limit if _amount is too large.
            balances[msg.sender]--;
            balances[_to]++;
        }
    }
}
```

Figure 2.11: Example of SC09 - Gas Limit Vulnerabilities.

```
SC10
// SPDX-License-Identifier: MIT
pragma solidity ^0.4.24;

contract Proxy {
    address public owner;

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

    function forward(address callee, bytes _data) public {
        require(callee.delegatecall(_data));
    }
}
```

Figure 2.12: Example of SC10 - Unchecked External Calls.

3

Related works

This chapter provides a comprehensive review of the existing literature on the detection and mitigation of vulnerabilities in smart contracts, with the goal of contextualizing the current project within the broader landscape of smart contract security research. Several surveys have sought to consolidate knowledge in this domain, offering detailed analyses of current methodologies while identifying critical gaps in existing tools and techniques.

Chu et al. [11] offers an in-depth examination of vulnerabilities at three distinct levels: the programming language (primarily Solidity), the virtual machine (Ethereum Virtual Machine), and the underlying blockchain infrastructure. This work underscores the inherent limitations of current approaches in addressing the complexities of real-world smart contract behavior, and calls for more holistic solutions capable of tackling issues across these levels.

The insufficiency of existing detection tools is a recurring theme in the literature. This limitation is particularly concerning given the dynamic nature of blockchain environments, where innovations can quickly render existing safeguards obsolete.

A key takeaway from these surveys is the urgent need for scalable and adaptive solutions. As smart contracts proliferate across various sectors, tools capable of managing this expansion must evolve in tandem. Current methods, though effective in controlled or limited scenarios, struggle with the scalability required for real-world applications. Furthermore, there is a noticeable gap in research addressing cross-platform vulnerabilities, as the majority of studies remain narrowly focused on Ethereum, despite the rise of other blockchain platforms like Binance Smart Chain and Solana.

The field of smart contract security has seen significant advancements through the adoption of various detection methodologies, including static and dynamic analysis techniques, machine learning-based approaches, and the emerging application of Large Language Models (LLMs) in code security. Each of these methodologies offers unique strengths and has contributed to the growing body of research on identifying and mitigating vulnerabilities in blockchain-based smart contracts.

3.1 Machine Learning Approach

The application of machine learning (ML) has introduced a paradigm shift in smart contract security, enabling the detection of patterns and anomalies in contract code that traditional methods often miss. Sendner et al. [12], propose a machine learning-based framework named ESCORT. This deep learning model employs a common feature extractor to analyze bytecode semantics, enabling the simultaneous detection of multiple vulnerabilities. Notably, ESCORT is equipped to adapt to emerging threats through transfer learning, requiring minimal additional data for retraining. Such adaptability is vital in the rapidly evolving blockchain ecosystem, where new vulnerabilities can arise unpredictably.

The work of Ivanov [13] provides a cornerstone for understanding the breadth of challenges in this domain. Ivanov's study meticulously catalogs 133 distinct vulnerabilities, creating a comprehensive taxonomy that serves as a foundation for future research. However, despite the rigor in classification, the study stops short of empirical testing, leaving a critical gap in its applicability to real-world scenarios. Complementing this effort, He [14] conducted an extensive evaluation of 27 detection tools used to identify vulnerabilities in smart contracts. This experimental approach offered invaluable benchmarks but was constrained by a limited variety of smart contracts, highlighting the need for broader testing frameworks.

Sun [15] proposed an innovative framework based on the BERT model, combining active and semi-supervised learning techniques. While this method demonstrates remarkable accuracy, its high computational costs pose a barrier to widespread adoption. Another noteworthy contribution comes from Zhang [16], who developed SVScanner, a tool that integrates dual-feature recognition with performance optimization to detect vulnerabilities. The approach, while robust, admits to gaps in identifying more subtle flaws, underscoring the ongoing challenge of comprehensive vulnerability detection.

These studies collectively reveal both commonalities and unique strengths. Many researchers

emphasize structured approaches like taxonomies and evaluation frameworks, which provide foundational tools for categorizing and assessing vulnerabilities. In parallel, advances in AI push the boundaries of what is achievable in terms of detection precision and scalability, as shown by the most recent works of Wang et al. [17] who developed "ContractWard", a model designed to effectively and efficiently detect six types of vulnerabilities in smart contracts by extracting static characteristics. Also, Waqas et al. [18] managed to validate a novel automated bug localization model that leverages a hybrid meta-heuristic algorithm: HGW-SFO, enhancing bug localization performance by optimizing bug feature extraction and prediction.

We can also see a variety of different techniques and models being developed and trained with the purpose of vulnerability detection in smart contracts. Semantic graph and residual graph convolutional networks are being employed, as shown by the work of Chen et al. [19]. This proposed approach leverages graph-based learning and attention mechanisms to effectively analyze smart contracts. Other researchers, like Yuan et al. [20] experimented with novel entropy embedding techniques, which combine token embedding, and the utilization of Bug Injection frameworks to automatically generate specific types of buggy code for fine-tuning of the models. Although, many challenges remain to be faced - such as the AI models' difficulties in detecting non-code-level vulnerabilities, such as those related to transaction records and traffic - these complementary efforts highlight the multifaceted nature of the problem and the necessity for integrated solutions. Despite the progress made, the field remains ripe with opportunities for further exploration. Empirical validation of proposed methods across a diverse range of smart contracts is essential to bridge the gap between theoretical models and practical application.

3.2 Large Language Models Approach

The integration of large language models (LLMs) in smart contract security could represent a transformative advancement in the field, particularly in the context of detecting vulnerabilities. One of the standout advantages of LLMs lies in their semantic understanding potential and their ability to generalize across various programming languages and blockchain platforms. Traditional vulnerability detection approaches, including static analysis and machine learning models, often suffer from narrow applicability and require extensive retraining to adapt to different ecosystems. LLMs, however, can operate effectively across diverse contexts by leveraging their pre-trained understanding of syntax, semantics, and code structures.

Jie et al. [21] highlight the specific utility of transformer-based LLMs in identifying vulnerabilities within Solidity, the primary programming language for Ethereum smart contracts.

Their study demonstrates how LLMs excel at recognizing complex, context-dependent relationships, such as variable misuse, hidden dependencies, and logical flaws, that could lead to severe security breaches. By capturing these intricate patterns, LLMs enhance the scope and precision of vulnerability detection, surpassing the capabilities of many existing tools.

Building on this foundation, Boi et al. [22] present a novel approach to vulnerability detection, leveraging LLMs as a comprehensive tool for analyzing smart contracts. Their proposed method utilizes the advanced natural language processing capabilities of LLMs to detect a wide range of vulnerabilities in a single analysis. By training the LLM on a diverse dataset comprising known vulnerabilities and secure coding practices, the model is equipped to identify subtle and complex flaws that traditional static analysis tools often overlook. This approach significantly advances the detection process, demonstrating its effectiveness, achieving high levels of accuracy and satisfaction in identifying vulnerabilities across different contract types and scenarios.

Moreover, LLMs offer additional capabilities that make them invaluable to both developers and security auditors. Tasks such as code completion, vulnerability explanation, and remediation suggestion are some of the potential strengths of LLMs which need to be further investigated. Therefore, the application of LLMs in this domain represents a significant leap forward in the use of artificial intelligence for blockchain security. By integrating advanced natural language processing with code analysis, LLMs could significantly enhance the reliability and safety of decentralized applications, with the potential to redefine the standards for smart contract security, paving the way for a more secure and resilient blockchain ecosystem.

4 Methodology

This chapter outlines the comprehensive methodology employed in the project to automate vulnerability detection and patch generation in smart contracts and their functions using LLMs. The methodology is divided into several phases, each designed to ensure the accuracy, efficiency, and effectiveness of the process.

4.1 Data Collection and Preparation

The foundation of this project lies in the utilization of a comprehensive dataset of Ethereum smart contracts written in Solidity, the predominant programming language for Ethereum smart contracts. This labelled dataset was curated by Sujeet et al.[10], and it is widely recognized for its detailed annotations of vulnerabilities. We turned to Etherscan, a well-established Ethereum blockchain explorer, in order to extract the real-world examples referenced in the ScrawlD dataset. In total, 9,252 smart contracts were collected, spanning a wide array of functionalities. The diversity within this dataset was crucial for evaluating the performance of detection models across a broad spectrum of use cases.

Data Preprocessing: The raw data required significant refinement to ensure relevance. To this end, a rigorous cleaning process was implemented. We made sure there were no duplicates or redundant entries, minimizing potential bias in subsequent analyses. In addition, to optimize the interactions with LLMs and minimize the token usage, all unnecessary whitespaces, line breaks, and comments were stripped from the code to reduce file size and focus exclusively on

the core logic. This not only made the dataset more compact but also eliminated extraneous elements that could interfere with model performance (for example: snippets of code which were made into comments, but remained inside the smart contract, were eliminated).

Data annotations: The subsequent phase focused on annotating the dataset with known vulnerabilities. This was a critical step that involved leveraging the methodology outlined by Sujeet et al.[10], which relies on a consensus-based threshold system to ensure the reliability of the labels. Under this system, a vulnerability is only reported if at least half of the supporting tools agree on its presence at the same location. For example, a "Reentrancy Attack" vulnerability, supported by three tools, would be flagged as valid only if at least two tools identified it at a specific location. This approach minimizes false positives and ensures that only well-substantiated vulnerabilities are recorded. The original dataset relied on the Smart Contract Weakness Classification (SWC) registry, a widely accepted framework at the time. However, the SWC registry has not been comprehensively updated since 2020. To address these shortcomings, our project employed the OWASP Top 10 Vulnerabilities for Smart Contracts taxonomy. This framework offers a more comprehensive and up-to-date classification system, better aligned with current best practices in security auditing. By mapping vulnerabilities from the SWC registry to OWASP categories, we ensured that our dataset remains relevant and consistent with contemporary security standards. The labeling process revealed that 5,706 smart contracts contained at least one vulnerability that met the threshold criteria. To enable a more detailed and granular understanding of vulnerabilities at the functional level, we created a set of 3,921 individual functions, extracted from the previously created dataset. These functions were annotated with their specific vulnerabilities, providing a granular view of how vulnerabilities manifest within the operational components of smart contracts. This additional layer of detail allows for deeper insights into the mechanics of vulnerabilities and offers a valuable resource for developing and testing detection models.

The specific OWASP vulnerabilities flagged in the smart contracts' dataset and in the functions' dataset are reported in Table 4.1. This preparation phase offers a dual-layer approach to vulnerability detection. By ensuring alignment with modern security frameworks and maintaining rigorous standards of data quality, this project establishes a robust foundation for evaluating the effectiveness of detection models. The diversity and granularity of the dataset enhances its utility for benchmarking.

Vuln. Type	Smart Contracts	Functions
SC01	654	939
SC02	3956	2333
SC03	145	284
SC04	283	0
SC05	1106	506
SC06	0	0
SC07	1696	0
SC08	0	0
SC09	0	0
SC10	18	28

Table 4.1: Vulnerability labels present in the whole smart contracts dataset and in the functions dataset.

4.2 Large Language Models

We already mentioned in Chapter 2 that the chosen LLMs to carry out our experiments would be: GPT-3.5-Turbo, GPT-01-mini and Gemini 1.5 Flash 8B. *GPT-3.5-turbo* emerged as an ideal candidate for this project, primarily due to its balance of advanced capabilities, adaptability, and cost-effectiveness. As part of OpenAI's Generative Pre-trained Transformer (GPT) series, GPT-3.5 builds on its predecessors' achievements by significantly enhancing its language comprehension and generative abilities. A key strength of GPT-3.5-turbo lies in its sheer scale. With over 200 billion parameters—an increase over GPT-3's 175 billion—the model is equipped to handle the intricacies of smart contract code. This larger parameter count enables GPT-3.5 to process complex language patterns and contextual relationships, making it adept at identifying vulnerabilities that depend on logic or interactions within the code. These capabilities allow the model to tackle tasks it was not explicitly trained on, such as detecting vulnerabilities in Solidity code, with minimal input or guidance. Cost-efficiency was another critical factor in selecting GPT-3.5-turbo. Compared to its successor GPT-4, GPT-3.5-turbo offers significantly lower operational costs while maintaining sufficient performance for tasks like vulnerability detection and code analysis. Given the scale of the project, which involved analyzing thousands of smart contracts and generating detailed assessments, the affordability of GPT-3.5-turbo was a major advantage. This low cost enabled the extensive testing and evaluation of the model's capabilities without incurring prohibitive expenses, making it a practical choice for large-scale experimentation.

GPT-o1-Mini emerged as a compelling choice as well for this project due to its balanced design, offering a practical trade-off between cost, performance, and efficiency. Released in September 2024, o1-Mini is part of the o1 family, designed for enhanced performance in resource-constrained environments. While its counterpart, o1-Preview, is built for superior reasoning and excels in complex problem-solving, o1-Mini offers a lightweight alternative, optimized for cost-effectiveness and high-throughput tasks. This balance made it particularly suitable for the large-scale analysis required in this project, where thousands of smart contracts were evaluated for vulnerabilities. With a parameter count of 100 billion—significantly smaller than o1-Preview's 300 billion—o1-Mini operates on a lightweight transformer architecture. Despite its smaller size, it incorporates advanced attention mechanisms and hierarchical reinforcement learning, enabling it to focus on critical aspects of input data. One of the key considerations in selecting o1-Mini was cost. Although it was more expensive to operate than GPT-3.5 Turbo, it remained significantly cheaper than o1-Preview, whose premium pricing placed it out of reach for a project of this scale. While o1-Preview would have been an excellent candidate due to its claimed reasoning capabilities, its high operational costs rendered it impractical. In contrast, o1-Mini provided a viable middle ground—offering advanced analytical capabilities at a fraction of the cost of its larger sibling.

Gemini 1.5 Flash-8B was the last but not least important candidate model used for this project, offering an balance of power and efficiency for complex computational tasks. As a variant of the Flash model, the 1.5 Flash-8B is optimized with 8 billion parameters, significantly enhancing its ability to process large datasets and handle intricate analytical requirements. This made it particularly suited for analyzing the diverse and complex dataset of Solidity smart contracts used in the project. Similar to the standard Flash model, the Flash-8B variant brings improved processing capabilities that are specifically designed for higher-order reasoning tasks. This level of token capacity and reasoning power allowed the model to process extensive sections of smart contract code, identifying vulnerabilities while maintaining a low-latency experience. These features were critical in scenarios requiring quick turnarounds without sacrificing analytical depth, such as batch analysis of large datasets.

4.3 Vulnerability Detection Phase

To evaluate the performance of large language models (LLMs) in detecting vulnerabilities in smart contracts, the we relied on a carefully designed prompt for vulnerability detection. This was foundational to understanding the capabilities of models like GPT-3.5, o1-mini, and Gemini-1.5-flash-8b in identifying potential security flaws. The prompt served to standardize the task for the models, ensuring that their analyses could be compared consistently across various types of vulnerabilities and contracts.

The vulnerability detection prompt reported in Figure 4.2 was structured to establish a clear context for the models. It introduced the OWASP Top 10 Vulnerabilities for Smart Contracts, each vulnerability was explicitly named and assigned a unique code, such as SC01 for Reentrancy Attacks or SC02 for Integer Overflow and Underflow, and so on. By presenting these categories upfront, the prompt directed the models' focus to specific, constrained and well-defined answers. The prompt then instructed the models to analyze the provided Solidity code and identify vulnerabilities using the OWASP codes. This design ensured clarity and consistency in the outputs, which were essential for automated parsing and subsequent analysis. For instance, if a model identified SC01 and SC03 vulnerabilities in a contract, its response could be directly compared against the dataset's known labels, allowing us to evaluate its accuracy.

4.4 Patch Generation Phase

The second phase of the interaction with the LLMs focused on evaluating their ability to generate secure patches for vulnerabilities identified in smart contracts. After vulnerabilities were detected using the first prompt, a second prompt, reported in Figure 4.3, was introduced to guide the models in generating patches. This prompt reiterated the vulnerabilities correctly identified in the contract, using the OWASP taxonomy codes (e.g., SC01, SC02). It provided explicit instructions to fix the listed vulnerabilities while preserving the contract's original functionality. The prompt emphasized the importance of retaining the intended behavior of the code, ensuring that the patches did not compromise the operational logic or introduce new errors. This focus on maintaining functional integrity was particularly important given the high-stakes nature of blockchain applications, where even minor deviations in behavior could result in significant operational consequences. The models were instructed to output only the revised Solidity code, avoiding any additional comments or explanations. This streamlined format ensured that the generated patches could be directly integrated into further analysis. For

example, if a model identified SC01 (Reentrancy Attacks) and SC02 (Integer Overflow and Underflow) as vulnerabilities, the prompt specifically asked it to address these issues while retaining the contract's original operations. The models' ability to understand and apply security best practices, such as adding mutexes to prevent reentrancy or using SafeMath libraries to handle integer operations, was critical to the success of this phase.

The patches generated by the models were then evaluated using traditional vulnerability detection tools: Slither, Mythril, SmartCheck, Osiris, and Oyente. These tools, where chosen specifically for the review of the generated patches because they are the same tools used originally used to create labeled dataset. Therefore they are the best suited candidates to verify whether the vulnerabilities originally detected in the smart contracts were effectively mitigated in the new patched versions. For instance, if a patched contract no longer triggered alerts for "Reentrancy Attacks" in Slither, this indicated that the model successfully addressed the issue. Conversely, if the tools still flagged the same vulnerabilities, it suggested that the model's patch was either incomplete or ineffective.

4.5 Performance Metrics

A critical aspect of this evaluation is the use of standard classification metrics to keep track of and measure the models' performance, both in the vulnerability detection phase and in the assessment of the patches generated by the LLMs. True Positives (TP) indicated instances where the model correctly identified a vulnerability present in the contract. False Positives (FP) occurred when the model identified a vulnerability that was not actually present, while False Negatives (FN) represented cases where the model failed to identify a vulnerability that was indeed present. These metrics formed the basis for calculating precision, recall, and F1-score, of the detection phase, providing a comprehensive view of each model's effectiveness.

Precision:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Recall:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

F1-Score:

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Precision measured the proportion of correctly identified vulnerabilities among all detec-

tions, reflecting the model's accuracy in its predictions. Recall, on the other hand, assessed the model's ability to identify all actual vulnerabilities, highlighting its thoroughness. The F1-score balanced these two metrics, offering a single measure to evaluate the overall performance of each model. These metrics were calculated for each OWASP vulnerability code, such as SC01 and SC02, to understand the specific strengths and weaknesses of the models in detecting particular types of vulnerabilities.

4.6 Character Obfuscation

To evaluate the robustness of LLMs against character obfuscation techniques, lastly we decided to focus on the detection of vulnerabilities in obfuscated smart contracts. The objective was to understand whether GPT-3.5-turbo and Gemini-1.5-flash-8b rely primarily on the semantic clarity of code (such as variable names) or if they can infer vulnerabilities purely based on the structural and logical aspects of the code. This investigation formed a crucial part of evaluating the practical resilience of LLMs in scenarios where adversarial techniques, such as code obfuscation, are applied. Variable obfuscation is a technique commonly employed to make code more challenging for both humans and automated systems to understand. In this experiment, the obfuscation targeted variable names in Solidity smart contracts. Variable names, while not affecting the functionality of the code, play a vital role in conveying meaning and intent. By systematically altering these names, the experiment sought to evaluate the extent to which LLMs depend on such semantic clues for vulnerability detection.

The obfuscation process was automated using a Python script specifically designed for this task. We employed a character substitution strategy to obscure variable names in a manner that maintained their syntactical validity while reducing their semantic clarity. For instance, the script transformed specific characters in variable names according to a predefined mapping: "l" was replaced with "1", "o" with "0", "i" with "l", "s" with "5", and "g" with "9". If the resulting variable name began with a digit (a syntax error in Solidity) it was prefixed with an underscore to ensure correctness. The script utilized a regular expression to identify variable declarations in Solidity smart contracts. It detected variable types (e.g., uint, address, or mapping) along with any associated access modifiers such as public or private, and then replaced the variable names using the obfuscation rules. Every occurrence of the identified variable was replaced consistently throughout the contract.

Vuln. Type	Overlapping TP
SC01	50
SC02	50
SC03	33
SC04	50
SC05	4
SC07	50
SC10	11

Table 4.2: A subset of true positives shared between GPT-3.5-turbo and Gemini 1.5 Flash-8B, which were later obfuscated.

The experimental design included the selection of up to 50 smart contracts for each OWASP vulnerability type (e.g., SC01, SC02, etc.), ensuring diversity across different categories. These contracts were chosen based on their prior correct labeling by both GPT-3.5-turbo and Gemini-1.5-flash-8b in earlier experiments. This selection provided a reliable baseline for comparison, as the models had already demonstrated their ability to accurately detect vulnerabilities in these contracts. The selected contracts were obfuscated using the described script, resulting in a new dataset, the composition of which is reported in Table 4.2. Subsequently, the obfuscated contracts were subjected to the same vulnerability detection prompts described in Section 4.3. Both LLMs were tasked with analyzing the obfuscated code and identifying vulnerabilities, and we kept track of the True Positives (TP) and False Negatives (FN), which allowed us to calculate the new recall to measure the models' performance on the obfuscated dataset. Additionally, we evaluated the effectiveness of traditional detection tools in labeling character obfuscated smart contracts with vulnerabilities.

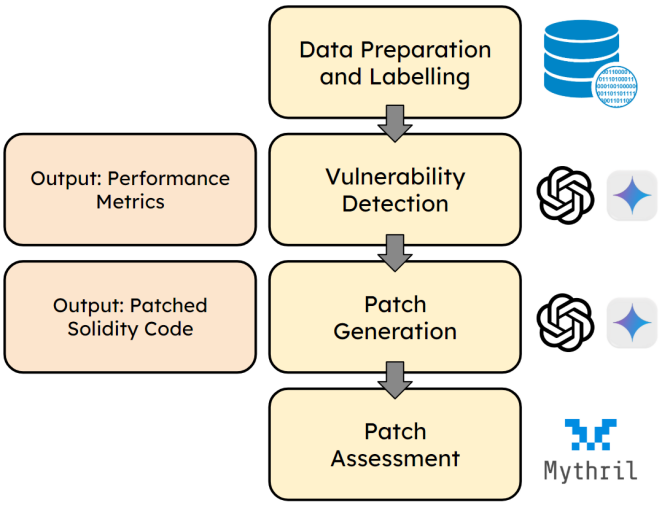

Figure 4.1: Architectural diagram for the complete vulnerability detection and patch generation process.

```
You are a security analyst specializing in smart contract
vulnerabilities, particularly the OWASP Top 10 for smart contracts:

{list of OWASP vulnerabilities}

Please analyze the following Solidity smart contract and identify if
any vulnerabilities from the list above are present. Provide your
analysis mentioning the vulnerability codes (e.g. SC01, SC02, etc.).

Smart Contract:
{contract_code}
```

Figure 4.2: Structure of prompt used for the vulnerability detection phase.

```
You are a smart contract developer and auditor. The following
Solidity smart contract has been analyzed and found to contain the
following vulnerabilities:

{list of vulnerabilities correctly detected}

Please provide a revised version of the contract where these
vulnerabilities are fixed. Ensure that the patched contract maintains
the same functionality as the original.

Original Smart Contract:
{contract_code}
```

Figure 4.3: Structure of prompt used for the patch generation phase.

5 Experiments and Results

In this chapter, we present the experiments conducted to evaluate the effectiveness of using LLMs, specifically GPT-3.5-turbo, GPT-01-mini and Gemini 1.5 Flash 8B, for vulnerability detection and patch generation in Solidity smart contract functions.

5.1 Experiment 1

The **(RQ1)** aimed to evaluate the effectiveness of the three models mentioned above in detecting vulnerabilities. This analysis was performed using two granularities: detection at the whole contract level and detection at the individual function level, using the two different datasets prepared for this purpose. Metrics such as precision and recall were calculated to gauge the model performance for each type of vulnerability in the OWASP, offering insight into their strengths and weaknesses.

5.1.1 Smart Contract Analysis

The analysis of precision and recall metrics, illustrated in Figures Figure 5.1 and Figure 5.2, reveals distinct performance patterns among the LLMs evaluated in different types of vulnerability at the entire contract level. The three models—GPT-3.5-turbo, GPT-01-mini, and Gemini-1.5-flash-8b—demonstrated strong precision in detecting SC02 vulnerabilities (Integer Overflow and Underflow), with scores exceeding 0.70. Specifically, GPT-01-mini achieved the high-

Vuln. Type	GPT-3.5	GPT o1-mini	Gemini
SC01	0.15	0.08	0.12
SC02	0.45	0.27	0.51
SC03	0.11	0.15	0.09
SC04	0.09	0.13	0.06
SC05	0.24	0.02	0.15
SC07	0.44	0.06	0.39
SC10	0.01	0.02	0.01

Table 5.1: F1 scores for GPT 3.5-turbo, GPT o1-mini and Gemini 1.5 Flash 8B in vulnerability detection of whole Solidity smart contracts.

est precision at 0.86, followed closely by GPT-3.5-turbo at 0.72 and Gemini at 0.80. This indicates that these models are particularly effective in identifying integer-related vulnerabilities. However, their performance was considerably weaker for other vulnerability types. For SC01 (Reentrancy Attacks) and SC03 (Timestamp Dependence), all models showed low precision, with scores falling below 0.15. GPT-3.5-turbo and Gemini recorded precision scores of 0.11 and 0.08 for SC01, respectively, while GPT-01-mini had a slightly higher precision of 0.12 but still within the low-performance range. This suggests that models struggle to accurately detect these types of vulnerabilities, potentially leading to a higher rate of false positives. In terms of recall, which measures the models' ability to identify all actual vulnerabilities, GPT-3.5-turbo excelled in detecting SC04 (Access Control Vulnerabilities) and SC10 (Unchecked External Calls), achieving high recall scores of 0.99 and 0.89, respectively. This indicates a strong capability to comprehensively detect these vulnerabilities. In contrast, its recall for SC01 was significantly lower at 0.27. GPT-01-mini showed poor recall in most vulnerabilities, with scores between 0.02 and 0.39, indicating difficulty in identifying most of the actual vulnerabilities. Gemini recall was moderate, performing better for SC10 with a score of 0.61 but remaining low for SC01 at 0.22.

Vuln. Type	GPT-3.5	GPT o1-mini	Gemini
SC01	0.19	0.32	0.15
SC02	0.69	0.46	0.54
SC03	0.23	0.21	0.28
SC05	0.06	0.05	0.11
SC10	0.02	0.04	0.02

Table 5.2: F1 scores for GPT 3.5-turbo, GPT o1-mini and Gemini 1.5 Flash 8B in vulnerability detection of specific Solidity functions.

5.1.2 Function Analysis

At the function level, the precision metrics, illustrated in Figures Figure 5.3 and Figure 5.4, reveal notable strengths and weaknesses among the evaluated Large Language Models (LLMs). GPT-3.5-turbo exhibited better precision in detecting the SC02 vulnerability, achieving a high score of 0.80, and showed commendable performance for SC01 with a precision of 0.39. This indicates a strong capability to accurately identify certain vulnerabilities when analyzing individual functions. However, its precision drastically dropped for SC10, scoring a mere 0.01, highlighting a significant challenge in detecting this particular issue. GPT-01-mini outperformed the other models in precision for SC01 and SC02, with impressive scores of 0.61 and 0.88 respectively, demonstrating robust effectiveness in pinpointing these vulnerabilities at the function level. Despite this, its performance was less remarkable for other vulnerabilities, indicating areas where improvement is needed. Gemini 1.5 Flash 8B displayed moderate precision in general. It performed best for SC02 with a precision of 0.71 but lagged behind for SC01, scoring only 0.26. In terms of recall, Gemini showed balanced results for SC02 (0.44) and achieved a relatively high score for SC10 (0.64). Nonetheless, it struggled significantly with SC05, attaining a low recall of 0.09, which suggests difficulty in consistently identifying this vulnerability.

5.1.3 Observation

In general, the analysis revealed that the performance of LLMs in vulnerability detection varies widely depending on the type of vulnerability and the granularity of the analysis. SC02 (Inte-

ger Overflow and Underflow) emerged as the vulnerability where all three models consistently performed best across both whole contract and function-level analyses. However, other vulnerabilities, such as SC10 (Unchecked External Calls), saw significant disparities in precision and recall, indicating that certain vulnerabilities are inherently more challenging for these models to detect. This variability may underscore the importance of tailoring LLM-based solutions to specific security needs, as their effectiveness could vary depending on the context.

5.2 Experiment 2

The **(RQ2)** aimed to explore the remediation capabilities of LLMs by assessing such fixes against traditional vulnerability detection tools. We evaluated the patched smart contracts and functions using traditional vulnerability detection tools: Slither, Mythril, SmartCheck, Osiris, and Oyente. These tools, previously employed by the authors in [10]. to label their dataset, provided a robust framework for understanding how well the patches mitigated the identified vulnerabilities. Our analysis focused on two levels: the performance of the tools on the entire patched contracts and on the specific patched functions.

Table 5.3 presents the results of the tools' assessments on whole patched smart contracts. For SC01 (Reentrancy Attacks), all 122 patched contracts were flagged by Slither, but only 13% surpassed the threshold of two tools confirming the presence of the vulnerability, suggesting that the generated patches effectively mitigated the issue in most cases. SC02 (Integer Overflow and Underflow) showed remarkable consistency, with 100% of the 666 patched contracts being flagged by at least two tools, demonstrating a failure to fully address this type of vulnerability. SC03 (Timestamp Dependence) and SC10 (Unchecked External Calls) had perfect detection consistency across tools, with all patched contracts meeting or exceeding their respective thresholds. In contrast, SC05 (Front-running Attacks) and SC07 (Logic Errors) exhibited lower rates of effective patching, with only 43% and 22% of patched contracts surpassing the threshold, respectively.

Table 5.4 shifts the focus to the specific functions that were analysed and patched. This granularity allowed us to evaluate whether targeting individual functions improved the effectiveness of the patches. For SC01, while 90 patched functions were analyzed, only 9% met the threshold for vulnerability detection by at least two tools, indicating better performance at the function level compared to the entire contract. SC02 remained a significant challenge, with 97% of the 712 patched functions flagged as vulnerable, underscoring the difficulty of generating robust patches for integer overflow and underflow vulnerabilities. SC03 and SC10 maintained their

Vuln. Type	Patched Contr.	Slither	Smartcheck	Mythril	Oyente	Osiris	Threshold
SC01	122	122	0	0	3	15	16 (13%)
SC02	666	666	0	9	637	362	666 (100%)
SC03	33	33	0	0	33	0	33 (100%)
SC04	37	37	37	2	0	0	37 (100%)
SC05	168	0	0	4	71	0	72 (43%)
SC07	269	38	0	24	0	0	60 (22%)
SC10	16	16	16	0	0	0	16 (100%)
	1311	**912**	**53**	**39**	**744**	**377**	
		69.56%	**4.04%**	**2.97%**	**56.75%**	**28.76%**	

Table 5.3: Vulnerability detection on whole Solidity smart contracts that were patched by GPT-3.5-turbo.

strong performance, with 100% of the patched functions meeting their thresholds. However, SC05, which targets front-running attacks, showed a diminished effectiveness at the function level, with only 45% of patched functions surpassing the threshold. These results highlight several important observations about the effectiveness of LLM-generated patches.

First, the ability of LLMs to address vulnerabilities varies significantly across different types of issues. While vulnerabilities such as SC03 and SC10 were consistently mitigated in both whole contracts and specific functions, others, like SC02 and SC07, remained challenging, with persistent detections by traditional tools, in particular Slither flagged the most vulnerabilities across the scope.

Second, the function-level analysis revealed that targeting specific functions for patching could, in some cases, improve the mitigation of vulnerabilities compared to patching the entire contract. This was particularly evident for SC01, where the relative performance at the function level outperformed that at the contract level. However, this approach was not universally beneficial, as demonstrated by SC05, where function-level patching did not yield substantial improvements.

Lastly, the reliance on traditional tools for evaluating patches provided a critical benchmark for the effectiveness of LLMs in generating secure code. While these tools were instrumental in identifying vulnerabilities, the results also highlighted their limitations. This variability points to the ongoing need for advancements in both automated detection tools and LLM methodologies to enhance smart contract security.

Vuln. Type	Patched Contr.	Slither	Smartcheck	Mythril	Oyente	Osiris	Threshold
SC01	90	90	0	0	0	8	8 (9%)
SC02	712	712	0	24	598	283	691 (97%)
SC03	35	35	0	0	35	0	35 (100%)
SC05	11	0	0	0	5	0	5 (45%)
SC10	18	18	18	0	0	0	18 (100%)
	866	**855**	**18**	**24**	**638**	**291**	
		98.73%	**2.08%**	**2.77%**	**73.67%**	**33.60%**	

Table 5.4: Vulnerability detection on specific Solidity functions that were patched by GPT-3.5-turbo.

5.3 Experiment 3

The **(RQ3)** aimed to explore how variable obfuscation affects the vulnerability detection capabilities of LLMs, we conducted a series of experiments using obfuscated smart contracts. This last experiment specifically evaluated the performance of GPT-3.5-turbo and Gemini-1.5-flash-8b. The objective was to understand the resilience of these models to character obfuscation techniques, which are designed to obscure variable names and reduce semantic clarity without altering the syntactic correctness or functionality of the contracts.

Variable obfuscation was implemented using a systematic process that replaced clear and descriptive variable names with obfuscated alternatives. For instance, a variable such as "`balance`" could be transformed into "`b4l4nc3`" by substituting certain characters with visually similar alternatives or numeric representations. This process forces the models to rely on deeper structural and logical patterns rather than the semantic meaning of variable names, creating a challenging environment to test their vulnerability detection capabilities. The obfuscated contracts were evaluated against the same set of OWASP vulnerability types as in the previous experiments. The effectiveness of the models was measured using recall, allowing us to assess the completeness of each model's detection capabilities under obfuscation.

The results summarized in Figure 5.5 revealed notable differences between the two models. GPT-3.5-turbo exhibited strong performance for certain vulnerabilities, such as SC04 (Access Control Vulnerabilities), where it achieved a perfect recall score of 1.00, and SC07 (Logic Errors), where it scored 0.82. However, it struggled significantly with SC10 (Unchecked External Calls), achieving a recall of only 0.45, indicating difficulty in identifying external call vulnerabilities. Additionally, its performance for SC05 (Front-running Attacks) was limited, with a recall score of 0.50, suggesting sensitivity to semantic obfuscation in this category.

Gemini-1.5-flash-8b, on the other hand, demonstrated greater robustness overall. It achieved perfect recall for SC10 (Unchecked External Calls) and SC04 (Access Control Vulnerabilities), indicating its ability to maintain detection accuracy for these categories despite obfuscation. For SC01 (Reentrancy Attacks), it achieved a recall of 0.90, significantly outperforming GPT-3.5-turbo, which scored 0.58. However, Gemini also exhibited vulnerabilities, particularly for SC03 (Timestamp Dependence), where its recall dropped to 0.52, suggesting reliance on specific patterns that were disrupted by obfuscation.

The impact of obfuscation varied by vulnerability type. Vulnerabilities like SC04, which involve access control mechanisms, were relatively immune to obfuscation, as both models maintained perfect recall scores. In contrast, SC03 and SC05, which involve timestamp dependence and front-running attacks, respectively, were more challenging for both models, highlighting areas where semantic understanding plays a significant role in detection.

But in order to actually assess the robustness of LLMs, we'd need to compare it to the performance of the traditional tools, when asked to detect the vulnerabilities in the obfuscated set. The results in Table 5.5 offer a clear view of how a consistent pattern emerges. Slither is confirmed to be the dominant performer among the tools. For multiple vulnerability types—such as SC01, SC02, SC03, and SC04 Slither successfully detects a substantial portion of the vulnerabilities in the obfuscated contracts, demonstrating a robust capacity to recognize these issues even when code has been rendered less readable. Similarly, Slither detects all SC03 cases and a notable fraction of SC07 instances in the character poisoned smart contract dataset, underscoring its adaptability and thoroughness under challenging conditions. In contrast, tools like Mythril, Oyente, and Osiris appear largely ineffective at identifying vulnerabilities in the obfuscated dataset. They report zero detections for most vulnerability categories, suggesting that these tools rely on patterns lost through obfuscation. SmartCheck, however, stands out as a partial exception. While it fails to detect many vulnerability types, it collaborates effectively with Slither in identifying SC04 and SC10 vulnerabilities.

Overall, these results highlight a significant variance in the resilience and adaptability of traditional detection tools when faced with character obfuscated code. Slither consistently emerges as the most robust tool, retaining much of its detection capability across multiple vulnerability types. SmartCheck can also prove effective in certain scenarios. On the other hand, Mythril, Oyente, and Osiris struggle, often failing to identify known vulnerabilities. These findings underscore the nuanced effects of obfuscation on LLM-based vulnerability detection. While both Gemini-1.5-flash-8b and GPT-3.5-turbo demonstrated demonstrate good overall resilience to obfuscation (greater than that of traditional tools), the experiments still reveal that obfuscation

Vuln. Type	OBF Dataset	Slither	Smartcheck	Mythril	Oyente	Osiris
SC01	50	37	0	0	0	0
SC02	50	50	0	0	0	0
SC03	33	33	0	0	0	0
SC04	50	19	32	1	0	0
SC05	4	0	0	0	0	0
SC07	50	15	0	0	0	0
SC10	11	11	11	0	0	0
	248	**165**	**43**	**1**	**0**	**0**
		66.53%	**17.34%**	**0.40%**	**0%**	**0%**

Table 5.5: Vulnerability detection on obfuscated whole Solidity smart contracts.

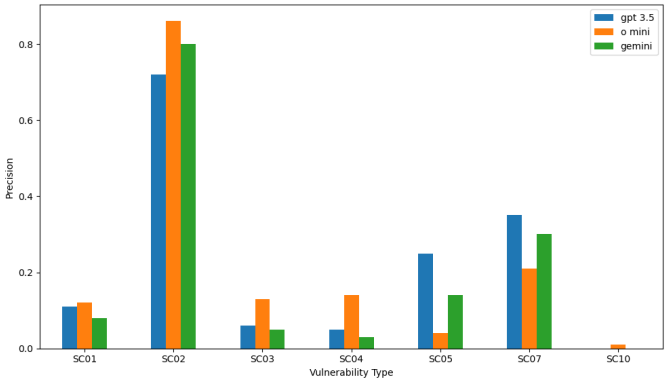

Figure 5.1: LLMs' vulnerability detection performance metrics when analyzing whole Solidity smart contracts.

disrupts the semantic cues relied upon by these models to varying degrees.

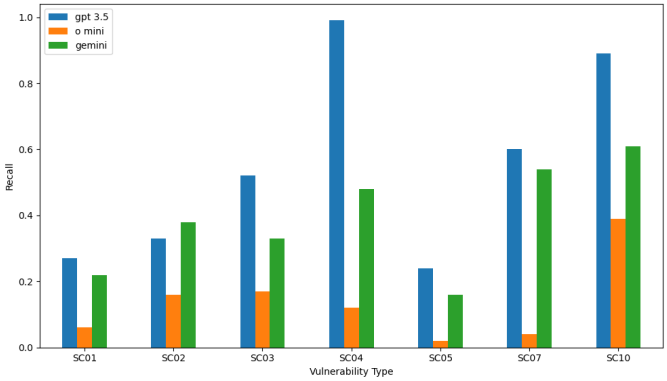

Figure 5.2: LLMs' vulnerability detection performance metrics when analyzing whole Solidity smart contracts.

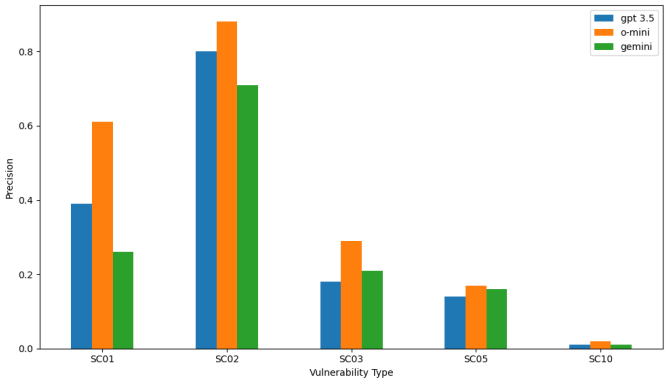

Figure 5.3: LLMs' vulnerability detection performance metrics when analyzing specific Solidity functions.

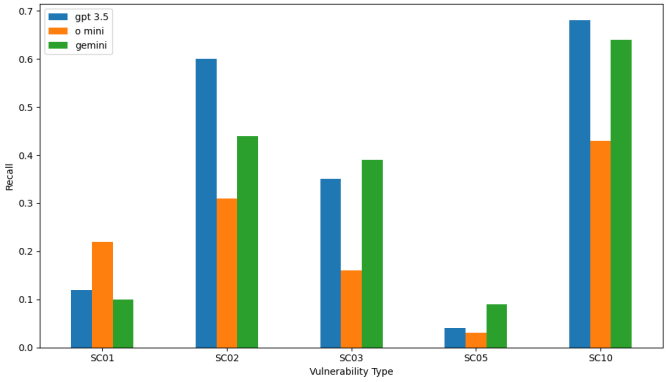

Figure 5.4: LLMs' vulnerability detection performance metrics when analyzing specific Solidity functions.

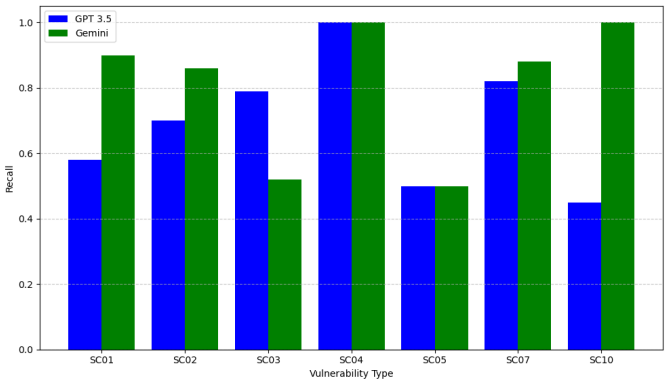

Figure 5.5: LLMs' vulnerability detection performance metrics when analyzing whole Solidity smart contracts whose variables have been obfuscated.

6 Conclusion

This study set out to explore the effectiveness of Large Language Models in detecting and mitigating vulnerabilities in Solidity-based smart contracts, a critical component of blockchain technology. Given the immutable and transparent nature of blockchain transactions, ensuring the security of smart contracts is paramount. We focused on three research questions to guide our investigation.

Firstly, we assessed how effective LLMs are in detecting smart contract vulnerabilities. Our experiments involved evaluating the performance of GPT-3.5-turbo, Gemini-1.5-flash-8b, and o1-mini using a robust dataset derived from the ScrawlD [10] repository. The findings revealed that all LLMs demonstrated great detection capabilities with regards to SC02, SC04 and SC10 vulnerabilities. However, we should highlight that o1-mini performed poorly across most categories, especially with regards to its recall, indicating limitations in its applicability for vulnerability detection in smart contracts.

Secondly, we explored the effectiveness of LLM-generated patches when assessed by traditional vulnerability detection tools such as Slither, Mythril, SmartCheck, Osiris, and Oyente. The results were promising, showing that LLMs have significant potential in generating patches that remediate specific vulnerabilities. In several cases, vulnerabilities that were previously detected by traditional tools were no longer identified after applying the LLM-generated patches. This demonstrates the practical utility of LLMs not only in detecting but also in addressing security issues within smart contracts.

Thirdly, we investigated the impact of variable obfuscation on the vulnerability detection ca-

pabilities of LLMs. By obfuscating variable names in a subset of smart contracts, we tested the robustness of GPT-3.5-turbo and Gemini-1.5-flash-8b against code obfuscation techniques. The experiments revealed that obfuscation did hinder the models' performance to some extent. GPT-3.5-turbo was notably affected, with decreased recall in certain vulnerability categories, indicating that it can be misled by the absence of semantic cues provided by meaningful variable names. Gemini-1.5-flash-8b showed greater resilience but still experienced reduced effectiveness in detecting specific vulnerabilities like timestamp dependence (SC03). Both LLMs however showed greater robustness than the traditional detection tools, which in most cases failed to detect the vulnerabilities they themselves had previously flagged, in the original non-obfuscated set.

Our study highlights the nuanced capabilities of LLMs in the realm of smart contract security. While they exhibit strong potential in both detecting and patching vulnerabilities, their performance varies across different types of vulnerabilities and can be influenced by code obfuscation. These findings suggest that LLMs can be valuable tools for developers and auditors, enhancing the security of smart contracts through automated analysis and remediation.

Building on the insights gained from this research, future studies could focus on evaluating newer models such as LLaMA 3 and other advanced LLMs. Investigating the effects of fine-tuning these models on domain-specific datasets could potentially enhance their performance, as suggested by the most recent experiments such as Detect Llama [23] by Ince et al. Fine-tuning has been shown to improve model accuracy and robustness in various contexts, and applying this technique could address some of the limitations observed with existing LLMs. Additionally, expanding the scope of obfuscation techniques and exploring adversarial examples could further assess the resilience of LLMs, contributing to the development of more secure smart contract analysis tools.

By continuing to explore and refine the application of LLMs in blockchain security, we can advance toward more reliable and secure decentralized systems, ultimately fostering greater trust and adoption of blockchain technologies.

References

[1] Zerocap. (2023) Ethereum smart contracts: How they changed crypto. Accessed: 2024-12-06. [Online]. Available: https://zerocap.com/insights/snippets/ethereum-smart-contracts-changed-crypto/

[2] J. Howell. (2022, October) Smart contract security guide. Accessed: 2024-12-06. [Online]. Available: https://101blockchains.com/smart-contract-security-guide/

[3] Z. Pratap. (2022, August) Reentrancy attacks and the dao hack. Accessed: 2024-12-06. [Online]. Available: https://blog.chain.link/reentrancy-attacks-the-dao-hack/

[4] CoinTelegraph. (2024) Defi protocol nexera hacked for 1.5m usd via smart contract exploit. Accessed: December 3rd, 2024. [Online]. Available: https://cointelegraph.com/news/nexera-hack-1-5-million-nxra-token

[5] Fortune Business Insights. (2024, November) Smart contracts market size. Accessed: 2024-12-06. [Online]. Available: https://www.fortunebusinessinsights.com/smart-contracts-market-108635

[6] S. Vaniukov, "Nlp vs. llm: A comprehensive guide to understanding key differences," February 2024, accessed: 2024-12-06. [Online]. Available: https://medium.com/@vaniukov.s/nlp-vs-llm-a-comprehensive-guide-to-understanding-key-differences-0358f6571910

[7] W. Yeadon, A. Peach, and C. Testrow, "A comparison of human, gpt-3.5, and gpt-4 performance in a university-level coding course," *Scientific Reports*, vol. 14, p. Article number: 23285, October 2024, accessed: 2024-12-06. [Online]. Available: https://www.nature.com/articles/s41598-024-73634-y

[8] OpenAI. (2024) Models. Accessed: 2024-12-06. [Online]. Available: https://platform.openai.com/docs/models/gp

[9] PromptHub. (2024) Gemini 1.5 flash-8b model card. Accessed: 2024-12-06. [Online]. Available: https://prompthub.org/models/gemini-1.5-flash-8b

[10] S. K. Chavhan Sujeet Yashavant and A. Karkare, "Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities," *arXiv preprint arXiv:2202.11409*, 2022.

[11] H. Chu, "A survey on smart contract vulnerabilities: Data analysis, tools, and methods," *arXiv preprint arXiv:2302.04567*, 2023.

[12] C. Sendner, "Smarter contracts: Detecting vulnerabilities in ethereum smart contracts using escort," *Journal of Blockchain Technology*, 2023.

[13] N. Ivanov, "Security threat mitigation for smart contracts," *ACM Computing Surveys*, July 2023.

[14] D. He, "Detection of vulnerabilities of blockchain smart contracts," *IEEE Internet of Things Journal*, July 2023.

[15] X. Sun, "Assbert: Active and semi-supervised bert for smart contracts," *Journal of Information Security and Applications*, January 2023.

[16] H. Zhang, "Svscanner: Detecting smart contract vulnerabilities," *Journal of Information Security and Applications*, April 2023.

[17] W. Wang, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, January 2020.

[18] A. Waqas, "Automated software bug localization enabled by meta-heuristic-based convolutional neural network and improved deep neural network," *Expert Systems with Applications*, December 2023.

[19] D. Chen, "Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention," *Journal of Systems and Software*, August 2023.

[20] D. Yuan, X. Wang, Y. Li, and T. Zhang, "Optimizing smart contract vulnerability detection via multi-modality code and entropy embedding," *Journal of Systems and Software*, vol. 202, p. 111699, 2023.

[21] W. Jie, "A novel extended multimodal ai framework towards smart contract security," *AI and Blockchain Advances*, 2023.

[22] B. Boi, C. Esposito, and S. Lee, "Smart contract vulnerability detection: The role of large language model (llm)," *ACM SIGAPP Applied Computing Review*, vol. 24, no. 2, pp. 19–29, August 2024.

[23] P. Ince, X. Luo, J. Yu, J. K. Liu, and X. Du, "Detect llama: Finding vulnerabilities in smart contracts using large language models," *arXiv preprint arXiv:2407.08969v1*, 2024, cs.CR, 12 Jul 2024.