VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY

UNIVERSITY OF INFORMATION TECHNOLOGY

Faculty of Computer Networks and Communications

Final Project Report

Nguyen Dang Quynh Nhu 22520150
Tran Van Chien 22520156
Le Minh Quan 22521181
Dang Duc Tai 22521270

GVHD: Phan The Duy

Group ID: G8 - Topic ID: CK14
Project Title: SCALM: Detecting Bad Practices in Smart Contracts Through LLMs
Subject: *NT522-Machine Learning in Cybersecurity*

June 25, 2025

Abstract

The increasing use of economically incentivized smart contracts is hindered by ongoing security vulnerabilities, leading to considerable financial losses due to their immutability after deployment. This paper introduces an innovative framework that combines finely-tuned large language models (LLMs) with Retrieval Augmented Generation (RAG) to improve the accuracy and clarity of smart contract vulnerability detection. By fine-tuning an open-source LLM and utilizing RAG, our model effectively integrates domain-specific external knowledge during inference, which significantly enhances threat identification. In evaluations on two public benchmarks, SolidiFI-Benchmark and Smart Bugs Curated, our model surpasses zero-shot LLMs by as much as 5% in F1-score, achieving an F1-score of up to 86% and a recall of 97%. These findings illustrate the effective collaboration between RAG and fine-tuning in delivering accurate and comprehensible assessments of smart contract security.

Keywords: Large Language Models, Finetuning, RAG, Smart Contract, Blockchain

Acknowledgements

I would like to thank my supervisor for his invaluable guidance, encouragement, and insightful feedback during the completion of this project. His expertise and support have been instrumental in shaping the direction and quality of my work. Thank you for providing the necessary facilities and resources that made this work possible.

Contents

List of Figures

Chapter 1

Introduction

Blockchain technology has made significant strides across various applications, including supply chain management, healthcare, and the pharmaceutical sector. Since smart contracts in blockchain are invariably linked to cryptocurrencies valued in the millions, any vulnerabilities within these contracts can lead to substantial financial losses Zhu et al. (2024). For example, a logical flaw in the smart contract of Euler Finance led to a miscalculation of health factors during collateral donations, culminating in a loss of $197 million in March 2023. In a similar vein, the Ronin Network suffered a loss of $615 million in 2022 due to compromised private keys and insufficient access control in its validator bridge. The primary factor contributing to these occurrences is the immutable nature of smart contracts once they are deployed on the blockchain, which means that any bugs become permanent and accessible to all, including potential attackers globally.

Therefore, the development of vulnerability detection tools is of paramount importance. Recently, a wide array of vulnerability detection tools for smart contracts has emerged to bolster blockchain security, including program analysis-based methods (e.g., Slither Feist et al. (2019)), fuzz testing-based approaches (e.g., ContractFuzzer Zhu et al. (2024)), and symbolic execution-based strategies (e.g., Mythril Jiang et al. (2018)). Nevertheless, these tools frequently fall short in identifying critical vulnerabilities, such as absent logic checks or oracle manipulation. Instead, they often concentrate on issues that are less frequently exploited in actual attacks, such as integer overflows. This shortcoming arises from their dependence on manually established rules and patterns, which limits their capacity to detect novel or intricate vulnerabilities Wu et al. (2024).

Consequently, instead of manually gathering characteristics of smart contracts, developers have turned to automated methods such as RAG-enhanced LLMs and fine-tuned LLMs. While conventional tools like Slither and Mythril generally apply fixed rules to identify known defects, they struggle with more complex or novel issues. Fine-tuned LLMs offer greater flexibility in detection by learning from smart contract data; however, they may hallucinate and lack transparency Wu et al. (2024). RAG-enhanced LLMs enhance accuracy and reduce hallucinations by retrieving real-world context (such as audit reports) to tackle these challenges Wu et al. (2024). Nonetheless, their implementation is more intricate, necessitating efficient retrieval mechanisms. Thus, to overcome the shortcomings of small-scale vulnerability databases and the dependence on manually defined patterns, we introduce RAG-SmartVuln, which investigates an improved vulnerability detection framework by integrating the fine-tuning of an open-source LLM within the smart contract auditing domain, supplemented by Retrieval-Augmented Generation (RAG). RAG-SmartVuln is distinct from traditional rule-based systems. Rather than depending on predefined patterns or expert oversight, our system harnesses the capabilities of large language models, which are subsequently fine-tuned on a domain-specific corpus of vulnerable smart contracts and their audits. Unlike previous LLM-based approaches that either depend on static prompts or necessitate extensive fine-tuning, we incorporate structured retrieval to direct model reasoning with domain-specific knowledge.

To assess the efficacy of our proposed methodology, we conduct experiments on two prominent datasets related to smart contract vulnerabilities: SolidiFI-Benchmark and SmartBugs-Curated. The findings indicate that our method is effective in the specialized area of identifying vulnerabilities in smart contracts.

In conclusion, the principal contributions of this paper are as follows:

- We fine-tune two open-source large language models (LLMs) utilizing domain-specific knowledge of Solidity, focusing on the task of detecting vulnerabilities within smart contracts.
- We develop a vulnerability knowledge base to facilitate retrieval-augmented generation (RAG). This base is founded on the premise that if a model can differentiate between a pair of vulnerable and non-vulnerable code snippets, despite minimal lexical variations (for instance, changes involving only a few tokens), it is considered to possess an advanced capability for understanding and reasoning about smart contract vulnerabilities.
- We evaluate our approach using the SolidiFI-Benchmark and SmartBugs Curated datasets. The results demonstrate outstanding performance across various weaknesses in smart contracts, achieving high precision.

1.1 Background & Related Work

1.1.1 Background

Large Language Models

Large Language Models (LLMs), such as GPT, BERT, and T5, are deep neural networks trained on massive corpora of text to perform a wide range of natural language processing (NLP) tasks Vaswani et al. (2017); Devlin et al. (2018); Raffel et al. (2020). By utilizing the Transformer architecture Vaswani et al. (2017), these models are able to capture semantic context and long-range relationships across text sequences. LLMs have recently shown cutting-edge performance in a number of applications, including code comprehension, summarization, question answering, and translation Chen et al. (2021). They are ideal for jobs involving reasoning, categorization, and code analysis, such as identifying vulnerabilities in smart contracts, because of their capacity to generalize from vast amounts of data Touvron et al. (2023); OpenAI (2023).

Retrieval-Augmented Generation (RAG)

A hybrid method called Retrieval-Augmented Generation (RAG) blends information retrieval methods with generative models. RAG architectures supplement the generation process by retrieving pertinent external documents from a knowledge base or vector store Lewis et al. (2020). Usually, dense embeddings and similarity search algorithms or vector databases (like FAISS or Pinecone) are used to accomplish this retrieval Johnson et al. (2019). RAG improves accuracy, lessens hallucinations, and allows for more domain-specific or current replies by integrating external context into the generation process Izacard and Grave (2021). This is especially helpful in situations where contextual accuracy and factual accuracy are crucial, like smart contract auditing.

Agentic AI

Agentic AI refers to systems that can act autonomously to pursue goals, make decisions, and interact with their environment. Unlike passive models, these agents can initiate actions, adapt to changes, and learn from experience. They are used in areas like autonomous vehicles, personal assistants, and reinforcement learning. As their capabilities grow, understanding their behavior and ensuring safe, aligned operation becomes increasingly important.

Smart Contract Weakness Classification

The process of locating and classifying vulnerabilities in smart contract code, usually implemented in Solidity, is known as "smart contract weakness classification." Reentrancy, integer overflows, access control problems, and unhandled exceptions are examples of common vulnerabilities Atzei et al. (2017). To find known vulnerability patterns, tools like Mythril, Slither, and Oyente use static and symbolic analysis Tsankov et al. (2018); Feist et al. (2019). However, these technologies frequently lack contextual awareness or have significant false positive rates. Deep learning and machine learning techniques, particularly those that use LLMs, have become strong substitutes that allow classification based on behavioral patterns and code semantics Tann et al. (2023); Zhou et al. (2023). Such models are trained and assessed using benchmark datasets such as curated vulnerability corpora or SmartBugs Durieux et al. (2020).

1.1.2 Related Work

The ubiquity of ChatGPT by OpenAI in late 2022 has increased, and we have seen a growing interest in using Large Language Models for various use cases. Therefore, several notable approaches have since emerged: For instance, Hu et al.'s GPTLens Hu et al. (2023) uses GPT-4 with a two-step auditor-critic process and open-ended prompts to rank vulnerabilities by correctness, severity, and profitability. Although it does not employ Retrieval-Augmented Generation (RAG), it achieves explainability through careful prompt engineering. Similarly, B. Boi's VulnHunt-GPT Boi et al. (2024) leverages GPT-3 and enhances prompting by incorporating vulnerability-specific data from the Top-10 DASP dataset. While not fully RAG-based, this integration of external information helps reduce hallucinations. In contrast, Du's VulRAG Du et al. (2024) explicitly uses RAG to detect bugs in C code by retrieving CVE-based data and transforming it into audit reports via LLMs, improving vulnerability detection through grounded, real-world knowledge.

To our best knowledge, the integration of RAG with fine-tuned open-source LLMs for Solidity vulnerability detection remains an underexplored direction, and our work seeks to address this gap.

1.2 Problem statement

Smart contracts, though central to decentralized applications, are often plagued by bad practices that can lead to vulnerabilities, inefficiencies, or unintended behaviors. Existing static analysis tools struggle to detect such practices comprehensively, often requiring expert intervention or manual effort. These limitations pose significant risks to the security and reliability of blockchain-based systems. Therefore, there is a pressing need for an automated, intelligent approach that can effectively identify bad practices in smart contracts. This paper addresses this gap by leveraging large language models (LLMs) to analyze smart contract code and detect potentially harmful coding patterns through a novel framework called SCALM.

1.3 Aims and objectives

Aims: The aim of this project is to develop an intelligent system, SCALM, capable of detecting bad practices in smart contracts by leveraging fine-tuned large language models (LLMs), retrieval-augmented generation (RAG), and agentic AI for autonomous knowledge base construction. Collect and curate a comprehensive dataset of smart contract weaknesses (e.g., SWC Registry) using an agentic AI agent capable of autonomously crawling, extracting, and updating relevant sources.

Objectives:

1. Design and implement a structured knowledge base to organize smart contract vulnerabilities and bad practices, enabling efficient retrieval and contextual analysis.

2. Apply RAG techniques to integrate external domain knowledge into the LLM's reasoning process, improving the model's understanding of smart contract contexts.

3. Fine-tune a large language model on annotated smart contract data to enable accurate detection and explanation of bad practices and anti-patterns in Solidity code.

4. Evaluate the performance of SCALM against baseline tools in detecting bad practices using metrics such as precision, recall, and F1-score.

5. Demonstrate SCALM's effectiveness in real-world scenarios by testing it on open-source smart contracts and analyzing its detection capability and usability.

Chapter 2

Literature Review

2.1 Overview of Smart Contract Security

Smart contracts are self-executing programs deployed on blockchain platforms like Ethereum. Although they offer transparency and automation, poorly written smart contracts are highly vulnerable to attacks. Past incidents, such as the DAO hack and Parity wallet vulnerabilities, highlight the consequences of insecure coding practices. To address this, several taxonomies of vulnerabilities have been proposed, such as the Smart Contract Weakness Classification (SWC) Registry.

2.2 Existing Detection Approaches

Traditional tools such as Mythril, Slither, and Oyente apply static or symbolic analysis to identify known vulnerabilities in Solidity contracts. While effective in some scenarios, these tools often rely on predefined rule sets and may not generalize well to complex or novel patterns. Moreover, they usually require deep domain knowledge, making them less accessible to general developers.

2.3 Use of Language Models in Smart Contract Analysis

Large Language Models (LLMs), like GPT-3 and CodeBERT, have shown strong capabilities in code understanding and generation. Recent work explores using LLMs to perform static analysis, code summarization, and vulnerability detection in source code. However, their application to smart contracts is still emerging. Fine-tuning such models on Solidity datasets has been shown to improve domain-specific performance (**?**).

2.4 Retrieval-Augmented Generation (RAG)

RAG is a hybrid technique that combines neural generation with external knowledge retrieval. It has been used to ground language models in domain-specific corpora, reducing hallucinations and improving factual consistency. In the context of smart contracts, RAG can be used to provide LLMs with access to vulnerability definitions, best practices, and real-world examples from sources like the SWC registry.

2.5 Agentic AI and Autonomous Knowledge Collection

Agentic AI refers to autonomous systems capable of initiating tasks and gathering resources with minimal human intervention. In this project, an agentic AI is used to crawl, extract, and store smart contracts in JSON format for further knowledge base construction.

2.6 Critique of the Literature

While existing tools and models address specific vulnerability types, they struggle with scalability, adaptability, and novel bad practices. Language models offer flexibility and contextual understanding but require task-specific training and domain grounding. RAG and agentic AI represent promising solutions to these limitations by bridging the gap between static knowledge and adaptive reasoning.

2.7 Summary

This chapter reviewed the current landscape of smart contract vulnerability detection, highlighting the limitations of traditional tools and the growing role of LLMs. It introduced the key technologies employed in SCALM, including fine-tuned LLMs, RAG, and agentic AI, positioning the proposed system as a novel and scalable solution for identifying bad practices in smart contracts.

Chapter 3

Methodology

Our proposed approach combines a RAG framework with fine-tuned open-source LLMs to detect vulnerabilities in smart contracts. As shown in Figure 1, the process includes three stages:

- **Stage 1:** SCALM first constructs a vulnerability knowledge graph based on the SWC Registry. It then leverages an agent to autonomously gather smart contracts and format them for further knowledge graph construction.
- **Stage 2:** For a given code snippet, we extract its functional sematics, then tracing its vulnerability information by knowledge graph, using it as an enriched context for LLM inference.
- **Stage 3:** Finally, our approach leverages fine-tune open-source LLMs and retrieves contextually relevant patterns from this knowledge graph to analyze the given smart contract. This, therefore, can lessen the possibility that LLMs may hallucinate.

3.1 Vulnerability Knowledge Graph Construction

As illustrated in Figure 3.1, the objective of phase 1 is to build a vulnerability knowledge graph. Therefore, SCALM leverages LLMs to extract the relevant vulnerability knowledge from existing Smart Contract Weakness Classification (SWC) instances, each of which includes at least a pair of vulnerable and non-vulnerable code snippets as samples. In cases where only the vulnerable version is available, and the patched version is absent, we use GPT-4 to generate its fixed version; after that, Mythril is used to verify that the patched code has no vulnerabilities. At the end of this stage, three-dimensional knowledge for each SWC instance will be constructed and stored in a vector database.

- **Functional Sematics:** This summarizes the key idea of the given smart contract in two levels: the first captures the general purpose of the code, whilst the second demonstrates detailed behaviors of it.
- **Vulnerability Causes:** This describes the reasons as to why vulnerabilities are present from three different perspectives:
  - Trigger action: The specific activity that activates the issue.
  - General vulnerability causes: An overview of the vulnerability causes.
  - Detailed vulnerability causes: An in-depth explanation of the vulnerability causes.

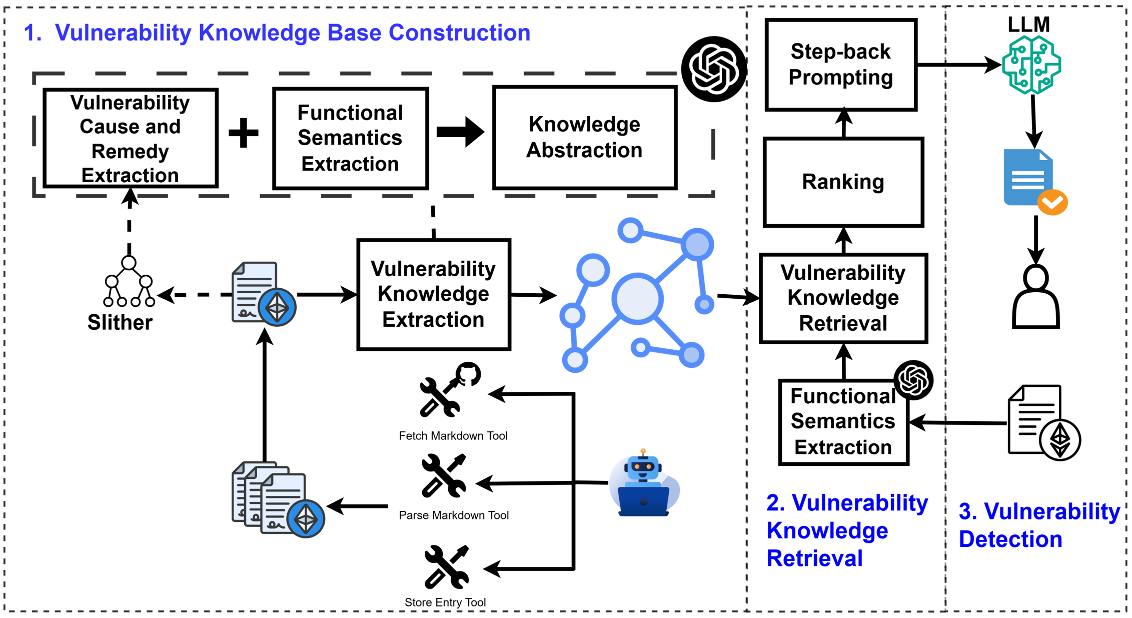

1. Vulnerability Knowledge Base Construction
Vulnerability Cause and Remedy Extraction
Functional Semantics Extraction
Knowledge Abstraction
Slither
Vulnerability Knowledge Extraction
Fetch Markdown Tool
Parse Markdown Tool
Store Entry Tool
Step-back Prompting
Ranking
Vulnerability Knowledge Retrieval
Functional Semantics Extraction
2. Vulnerability Knowledge Retrieval
LLM
3. Vulnerability Detection

Figure 3.1: Overall Achitecture of SCALM

- Remedies: It summarizes the solution strategy by comparing the vulnerable code with its corresponding patch, which would highlight the changes made to address the vulnerability.

3.1.1 Fetcher Agent

To support the SCALM system with up-to-date and structured knowledge, an agentic AI was implemented to autonomously collect and process data from the Smart Contract Weakness Classification (SWC) Registry.[1] This agent performs three key tasks: fetching markdown documentation, parsing relevant information, and storing structured entries into a knowledge base.

Agent Workflow

The agent was implemented using the `smolagents` framework and is powered by the `Qwen2.5-7B-Instruct` large language model. It leverages tool-based prompting to orchestrate the following components:

- **fetch_markdown:** Downloads raw markdown content from the SWC registry on GitHub using a specified SWC ID (e.g., SWC-100).
- **parse_markdown:** Extracts the vulnerability *description* and any Solidity `.sol` code examples directly from the markdown using regular expressions.
- **store_entry:** Appends the structured content as a JSON object into a persistent knowledge file named `swc_contracts.json`.

[1] https://github.com/SmartContractSecurity/SWC-registry

Automation Loop

To collect multiple entries, the agent iterates over a list of SWC IDs. For each ID, it constructs a high-level prompt (e.g., “Fetch, parse, and store SWC-128”), which the agent interprets and executes through the composition of the registered tools.

```python
for swc_num in range(128, 129):
    swc_id = f"SWC-{swc_num}"
    result = manager.run(f"Fetch, parse, and store {swc_id}")
```

Listing 3.1: Automation loop for collecting SWC entries

Example Output

After execution, the system stores entries in structured JSON format. A sample stored entry for `SWC-128` may look like:

```json
{
  "id": "SWC-128",
  "description": [
    "Incorrect inheritance order can lead to unexpected behavior..."
  ],
  "IncorrectInheritance.sol": "contract A { ... }"
}
```

Listing 3.2: Sample output in swc_contracts.json

Benefits

This autonomous agent ensures that the knowledge base remains current and machine-readable. It eliminates the need for manual data collection and forms a foundation for downstream tasks like Retrieval-Augmented Generation (RAG) and fine-tuning LLMs on domain-specific vulnerabilities.

3.1.2 Vulnerability Knowledge Extraction

For each SWC instance, SCALM leverages LLMs to extract a three-dimensional knowledge representation. Finally, the system utilizes LLMs to abstract this knowledge to a more generalized context. We will explain each step below:

Functional Sematics Extraction: We prompt LLMs to summarize both the general purpose and detailed behaviors of the buggy smart contract with the following instructions:

Prompt for General Purpose Extraction: [Vulnerable Code] What is the purpose of the above code snippet? Please summarize the answer in one sentence with the following format: “General purpose:”.
Prompt for Detailed Behaviors Extraction: [Vulnerable Code] Please summarize the functions of the above code snippet in the list format without any other explanation: “Detailed Behaviors: 1. 2. 3...”.

Vulnerability Cause and Remedy Extraction Given that the vulnerability causes and remedies are logically related, SCALM extracts them at the same time so that the interpretability of LLMs could be strengthened. In detail, SCALM is equipped with two rounds of extraction at this stage. The first one prompts LLMs to explain why the modifications of the fixed code are essential. Because of the unique characteristics of Solidity, particularly in the way contracts invoke one another, the CFG plays a pivotal role in demonstrating the sequence of function calls and the flow of data in a contract. This is undoubtedly important for identifying reentrancy vulnerability, where a function may be re-invoked before its initial execution finishes. Therefore, input at this round would be both the Solidity codes and the CFG representations. Second, based on that explanation, the LLM summarizes the root cause and solution. This follows a Chain-of-Thought (CoT) approach for step-by-step reasoning. Moreover, SCALM incorporates few-shot learning by providing a demonstration example of vulnerability causes and corresponding solutions. The detailed prompts are as follows:

Round 1: This is a code snippet with a vulnerability [SWC ID]: [Vulnerable Code] The vulnerability is described as follows: [Description], and this is how the control flow of the vulnerable code is described in Slither: [CFG representation of Vulnerable Code] The code after modification is as follows: [Patched Code], and this is how the control flow of the fixed code is described in Slither: [CFG representation of Fixed Code] Why is the above modification necessary?
Round 2: Then, I want you to act as a vulnerability expert. Based on the explanation above, summarize (1) the specific behavior causing the vulnerability and (2) the solution to fix it. Use the following examples as a guide: "Vulnerability Description example: ...", "Solution Description example: ...".

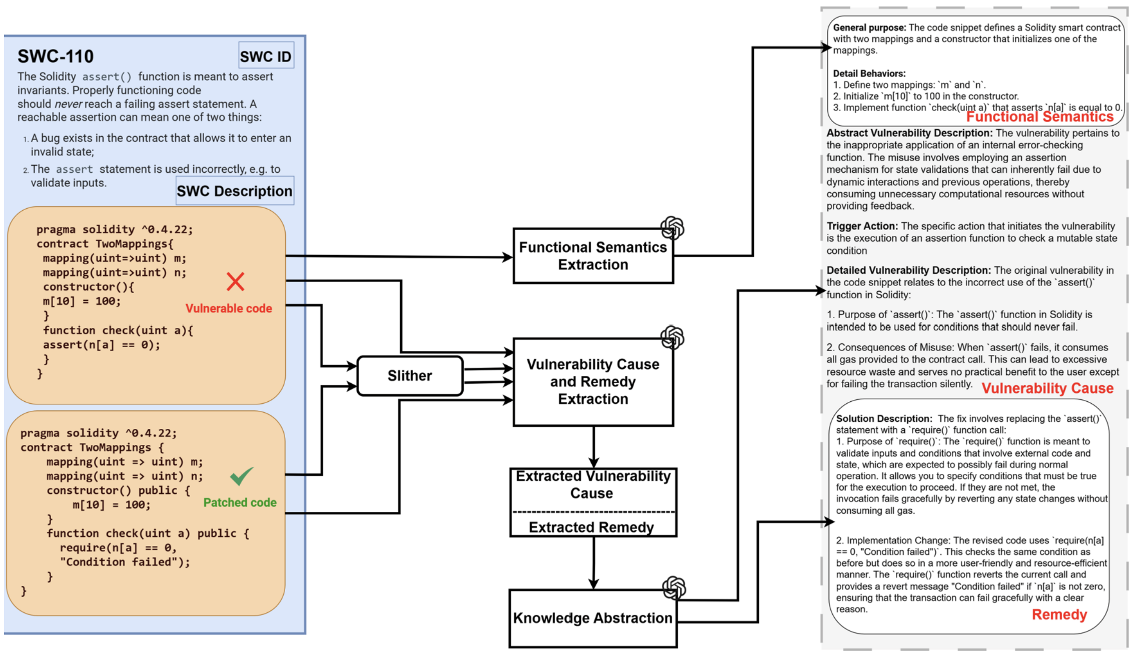
3.1.3 Knowledge Abstraction

In order to make the knowledge more general and reusable, we utilize LLMs to abstract retrieved vulnerability descriptions, eliminating specific code specifics because many vulnerabilities have similar patterns. Since functional semantics are used only for retrieval and not LLM input, they are not abstracted. The abstraction prompt is as follows:

Given the extracted vulnerability knowledge from Solidity smart contracts, your task is to abstract and generalize this knowledge to enhance its applicability across different smart contract implementations. Please follow the guidelines and examples below:

Guidelines for Abstraction:

- Abstracting Method Invocations: The extracted knowledge might contain concrete method invocations with detailed function calls (e.g., `token.transfer(msg.sender, amount)`) which can be abstracted into a generalized description (e.g., *"Performing an external token transfer to a user-controlled address."*).
- Abstracting Variable Names and Types: The extracted knowledge might contain concrete variable names or types (e.g., `owner = msg.sender`), which can be abstracted into a more general description (e.g., *"Assigning contract ownership to a deployer address."*).
- Abstracting Solidity-Specific Constructs: The extracted knowledge might con-



SWC-110

The Solidity `assert()` function is meant to assert invariants. Properly functioning code should *never* reach a failing assert statement. A reachable assertion can mean one of two things:

1. A bug exists in the contract that allows it to enter an invalid state;
2. The `assert` statement is used incorrectly, e.g. to validate inputs.

```
pragma solidity ^0.4.22;
contract TwoMappings{
 mapping(uint=>uint) m;
 mapping(uint=>uint) n;
 constructor(){
 m[10] = 100;
 }
 function check(uint a){
 assert(n[a] == 0);
 }
}
```

```
pragma solidity ^0.4.22;
contract TwoMappings {
    mapping(uint => uint) m;
    mapping(uint => uint) n;
    constructor() public {
        m[10] = 100;
    }
    function check(uint a) public {
      require(n[a] == 0,
      "Condition failed");
    }
}
```

General purpose: The code snippet defines a Solidity smart contract with two mappings and a constructor that initializes one of the mappings.

Detail Behaviors:
1. Define two mappings: `m` and `n`.
2. Initialize `m[10]` to 100 in the constructor.
3. Implement function `check(uint a)` that asserts `n[a]` is equal to 0.

Abstract Vulnerability Description: The vulnerability pertains to the inappropriate application of an internal error-checking function. The misuse involves employing an assertion mechanism for state validations that can inherently fail due to dynamic interactions and previous operations, thereby consuming unnecessary computational resources without providing feedback.

Trigger Action: The specific action that initiates the vulnerability is the execution of an assertion function to check a mutable state condition

Detailed Vulnerability Description: The original vulnerability in the code snippet relates to the incorrect use of the `assert()` function in Solidity:

1. Purpose of `assert()`: The `assert()` function in Solidity is intended to be used for conditions that should never fail.

2. Consequences of Misuse: When `assert()` fails, it consumes all gas provided to the contract call. This can lead to excessive resource waste and serves no practical benefit to the user except for failing the transaction silently.

Solution Description: The fix involves replacing the `assert()` statement with a `require()` function call:
1. Purpose of `require()`: The `require()` function is meant to validate inputs and conditions that involve external code and state, which are expected to possibly fail during normal operation. It allows you to specify conditions that must be true for the execution to proceed. If they are not met, the invocation fails gracefully by reverting any state changes without consuming all gas.

2. Implementation Change: The revised code uses `require(n[a] == 0, "Condition failed")`. This checks the same condition as before but does so in a more user-friendly and resource-efficient manner. The `require()` function reverts the current call and provides a revert message "Condition failed" if `n[a]` is not zero, ensuring that the transaction can fail gracefully with a clear reason.

Figure 3.2: Three-dimensional knowledge

tain Solidity-specific construct (e.g., `msg.sender == owner`), which can be abstracted into a more general description (e.g., *"Enforcing access control by validating caller's address."*).

Task: Based on the Extracted Vulnerability Knowledge below, provide:

- **Abstract Vulnerability Description:** A generalized version of the vulnerability.
- **Trigger Action:** The specific action that initiates the vulnerability.

Extracted Vulnerability Knowledge:

<Your Extracted Vulnerability Knowledge in previous stage>

Format Output as Plain Text:

```
Abstract Vulnerability Description:  <>
Trigger Action:  <>
```

3.2 Vulnerability Knowledge Retrieval

For each smart contract submitted for analysis, RAG-SmartVuln utilizes a large language model (LLM) to extract its functional semantics, as illustrated in Figure 3.1. These semantics are then used to retrieve relevant knowledge items from a vulnerability knowledge graph. The retrieval process is composed of two main steps:

- Semantic Similarity Search: The extracted functional semantics are compared against a large set of existing semantic representations in the knowledge graph using similarity search techniques.

- Knowledge Graph Traversal: For each semantically similar function retrieved, the system traces associated nodes in the knowledge graph—such as linked vulnerabilities and corresponding mitigation strategies—since the graph stores interrelated chains of knowledge (e.g., a function semantic linked to a vulnerability, which is in turn linked to a solution).

Through this process, RAG-SmartVuln effectively retrieves vulnerability knowledge that is highly relevant to the functional semantics of the input smart contract.

Finally, the top-k highest-ranked knowledge items are passed to the fine-tuned LLMs for vulnerability detection.

3.3 Vulnerability Detection

3.3.1 Fine-tuning open-source LLMs

We use 4-bit quantization (via bitsandbytes) and LoRA for efficient fine-tuning on three advanced open-source models: **Qwen-2.5-Coder-14B** and **DeepSeek-R1-Distill-Llama-8B**.

Qwen-2.5-Coder-14B: A 14B parameter model that has been improved through domain-specific pretraining and is optimized for code tasks such as generation and understanding.

Qwen-3-14B: a general-purpose 14B model that performs better on a variety of tasks and has enhanced reasoning and linguistic skills.

DeepSeek-R1-Distill-Llama-8B: A distilled, efficient version of DeepSeek-R1 with strong reasoning, built on reinforcement learning techniques, and improved data handling.

3.3.2 Vulnerability Detection

RAG-SmartVuln uses retrieved knowledge items and refined LLMs to assess whether the input code is vulnerable. Due to prompt length limits, it sends k separate prompts—each with the code and one knowledge item. If the output demonstrates the root cause of a vulnerability described in an item but lacks the corresponding fix, it's marked as vulnerable. Otherwise, the process continues with the next item. This repeats until a vulnerability is found or all items are checked. If none match, the code is considered non-vulnerable. The vulnerability detection prompt is structured as follows:

Instruction:
You are an Ethereum, Solidity, and DeFi security expert. Analyze the code for vulnerabilities, detailing each issue, its fix, and type.
Question:
List all vulnerabilities in this smart contract. Review the context below before analysis.
Context:
[Insert each knowledge item here.]
Code:
[Insert the code snippet here.]

Chapter 4

Results

4.1 Performance Comparison of Models on SolidiFI and SmartBugs Datasets

Fig 4.1 presents a comparative analysis of four models: **GPT-3.5**, **GPT-3.5-RAG**, **Finetune Qwen-2.5-14B with RAG (FTQR)**, and **Fine-tuned Deepseek with RAG (FTDR)**, evaluated on two benchmark datasets—**SolidiFI** and **SmartBugs**. The models are assessed using four metrics: Accuracy (Acc), Precision (Pre), Recall (Rec), and F1-score (F1).

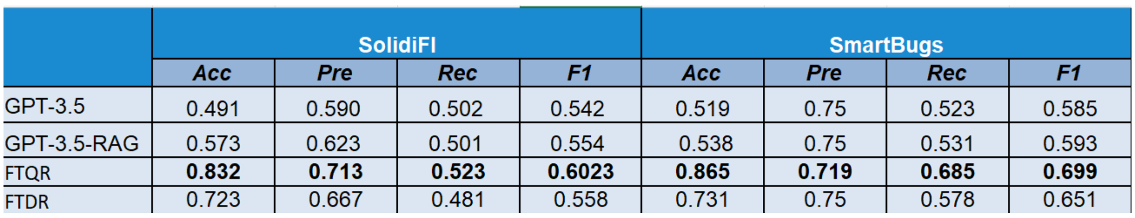
4.1.1 GPT-3.5 vs. GPT-3.5-RAG

- **On SolidiFI:** GPT-3.5-RAG demonstrates better *accuracy* (increasing from 0.491 to 0.573) and higher *precision* (0.590 to 0.623) compared to GPT-3.5. However, its *recall* slightly decreases (from 0.502 to 0.501), leading to only a modest gain in *F1-score* (from 0.542 to 0.554).
- **On SmartBugs:** GPT-3.5-RAG consistently improves upon GPT-3.5 across all metrics, with increases in *accuracy* (0.519 to 0.538), *precision* (0.750 to 0.750, unchanged), *recall* (0.523 to 0.531), and *F1-score* (0.585 to 0.593).

Conclusion: GPT-3.5-RAG provides slightly better performance than GPT-3.5, suggesting that retrieval-augmented generation (RAG) has a modest positive effect.

4.1.2 Finetune Qwen-2.5-14B with RAG (FTQR)

- **On SolidiFI:** FTQR achieves the highest *accuracy* (0.832) and a competitive *F1-score* (0.6023).
- **On SmartBugs:** FTQR further demonstrates its effectiveness with an *accuracy* of 0.865 and an *F1-score* of 0.699.



	SolidiFI				SmartBugs			
	Acc	***Pre***	***Rec***	***F1***	***Acc***	***Pre***	***Rec***	***F1***
GPT-3.5	0.491	0.590	0.502	0.542	0.519	0.75	0.523	0.585
GPT-3.5-RAG	0.573	0.623	0.501	0.554	0.538	0.75	0.531	0.593
FTQR	**0.832**	**0.713**	**0.523**	**0.6023**	**0.865**	**0.719**	**0.685**	**0.699**
FTDR	0.723	0.667	0.481	0.558	0.731	0.75	0.578	0.651

Figure 4.1: Comparision multiple models on two Benchmarks

Conclusion: FTQR is the best-performing model, significantly outperforming GPT-based models in both datasets.

4.1.3 Fine-tuned Deepseek with RAG (FTDR)

- **On SolidiFI:** FTDR achieves an *accuracy* of 0.723 and an *F1-score* of 0.558.
- **On SmartBugs:** FTDR shows solid performance with an *accuracy* of 0.731 and an *F1-score* of 0.651.

Conclusion: FTDR is the second-best model, offering a good balance between precision and recall, particularly on the SmartBugs dataset.

4.2 Summary

- **FTQR** $>$ **FTDR** $>$ **GPT-3.5-RAG** $>$ **GPT-3.5** in terms of overall performance.
- RAG improves GPT-3.5 marginally.
- Fine-tuned models (FTQR and FTDR) clearly outperform GPT-based models.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this work, we describe a sophisticated framework for smart contract vulnerability detection that makes use of graph-based knowledge representation, multi-agent collaboration, and fine-tuned open-source big language models. Our method builds on earlier work by breaking down the reasoning process across specialized agents and boosting contextual comprehension through structured knowledge graphs, both of which are inspired by SCALM's step-back prompting strategy. Using models like Qwen-2.5-14B-Coder, this multi-layered architecture maintains cost-efficiency while enabling deeper semantic analysis and more explicable detection results.

Based on benchmark smart contract vulnerability datasets, empirical results show that our system performs better in accuracy and F1-score than baselines based on GPT-3.5, both with and without Retrieval-Augmented Generation. A possible path for scalable, transparent, and efficient smart contract auditing is the combination of lightweight LLM inference, collaborative agent behavior, and external knowledge.

5.2 Future work

While the current system exhibits strong performance, several avenues remain for future improvement:

- **Attribution of Fine-grained Vulnerability:** By associating each identified vulnerability with particular code lines, contract elements, or graph paths for more understandable developer input, future iterations could improve explainability.
- **Contextual Integration with On-chain:** A more comprehensive foundation for behavioral vulnerability research might be offered by integrating runtime or on-chain data (such as gas consumption, execution traces, and past deployment patterns).
- **Input from Humans in the Loop:** By using rule-guided finetuning or reinforcement learning, expert-in-the-loop workflows could enable ongoing improvement of the multi-agent decision-making process.
- **Use of the Platform for Auditing:** Integration with CI/CD pipelines for real-time security auditing would be made possible by packaging the framework into a modular, containerized service (for example, using REST API or CLI tools).

References

Atzei, N., Bartoletti, M. and Cimoli, T. (2017), 'A survey of attacks on ethereum smart contracts (sok)', *Principles of Security and Trust* .

Boi, B., Esposito, C. and Lee, S. (2024), Vulnhunt-gpt: a smart contract vulnerabilities detector based on openai chatgpt, *in* 'Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing', pp. 1517–1524.

Chen, M. et al. (2021), 'Evaluating large language models trained on code', *arXiv preprint arXiv:2107.03374* .

Devlin, J. et al. (2018), 'Bert: Pre-training of deep bidirectional transformers for language understanding', *arXiv preprint arXiv:1810.04805* .

Du, X., Zheng, G., Wang, K., Feng, J., Deng, W., Liu, M., Chen, B., Peng, X., Ma, T. and Lou, Y. (2024), 'Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag'.
URL: *https://arxiv.org/abs/2406.11147*

Durieux, T. et al. (2020), 'Empirical review of automated analysis tools for smart contracts', *Proceedings of the ACM on Programming Languages* .

Feist, J. et al. (2019), 'Slither: A static analysis framework for smart contracts', *NDSS* .

Hu, S., Huang, T., İlhan, F., Tekin, S. F. and Liu, L. (2023), 'Large language model-powered smart contract vulnerability detection: New perspectives'.
URL: *https://arxiv.org/abs/2310.01152*

Izacard, G. and Grave, E. (2021), 'Leveraging passage retrieval with generative models for open domain question answering', *arXiv preprint arXiv:2007.01282* .

Jiang, B., Liu, Y. and Chan, W. K. (2018), Contractfuzzer: fuzzing smart contracts for vulnerability detection, Association for Computing Machinery.
URL: *https://doi.org/10.1145/3238147.3238177*

Johnson, J. et al. (2019), 'Billion-scale similarity search with gpus', *IEEE Transactions on Big Data* .

Lewis, P. et al. (2020), Retrieval-augmented generation for knowledge-intensive nlp tasks, *in* 'NeurIPS'.

OpenAI (2023), 'Gpt-4 technical report'. https://openai.com/research/gpt-4.

Raffel, C. et al. (2020), 'Exploring the limits of transfer learning with a unified text-to-text transformer', *JMLR* .

Tann, W. et al. (2023), 'Smartu: Learning-based smart contract weakness classification', *IEEE Transactions on Software Engineering* .

Touvron, H. et al. (2023), 'Llama: Open and efficient foundation language models', *arXiv preprint arXiv:2302.13971* .

Tsankov, P. et al. (2018), 'Securify: Practical security analysis of smart contracts', *CCS* .

Vaswani, A. et al. (2017), 'Attention is all you need', *NeurIPS* .

Wu, G., Wang, H., Lai, X., Wang, M., He, D. and Chan, S. (2024), 'A comprehensive survey of smart contract security: State of the art and research directions', *Journal of Network and Computer Applications* .
URL: *https://www.sciencedirect.com/science/article/pii/S1084804524000596*

Zhou, J. et al. (2023), 'Sola: A self-supervised framework for smart contract vulnerability detection', *arXiv preprint arXiv:2303.04542* .

Zhu, H., Yang, L., Wang, L. and Sheng, V. S. (2024), 'A survey on security analysis methods of smart contracts', *IEEE Transactions on Services Computing* .