HOCHSCHULE
MITTWEIDA
University of Applied Sciences

Internship Report

Analyzing Smart Contract Exploits or Other Security Incidents with LLMs

by

Mohd Saiful Islam

Faculty:
Applied Computer Sciences and Biosciences

Study Programme:
Applied Mathematics in Networking and Data Science

University Supervisor:
Prof. Dr.-Ing. Alexander Lampe

BCCM Supervisor:
Dipl. Volkswirt, Dipl. Kfm. Mario Oettler

June 30, 2025

Acknowledgements

I would like to express my deepest gratitude to all those who contributed to the successful completion of this internship and research project on analyzing smart contract vulnerabilities using Large Language Models.

First and foremost, I am profoundly grateful to Prof. Dr.-Ing. Alexander Lampe, my supervisor, for his exceptional guidance and support throughout this internship. His expertise in blockchain technologies and machine learning was invaluable in shaping my research direction. I particularly appreciate his insightful feedback on my work with the LLaMA-3 model and Retrieval-Augmented Generation techniques, which significantly improved the quality of this investigation.

My sincere thanks go to the BCCM, Germany, and especially to Mario Oettler for providing an excellent research environment and for sharing his practical knowledge about smart contract security. Our discussions about real-world blockchain vulnerabilities greatly enhanced my understanding of the challenges in DeFi security.

I am deeply indebted to Hochschule Mittweida, particularly the Faculty of Applied Computer Sciences and Biosciences, for making this internship possible. The institution's emphasis on practical, research-oriented learning provided the perfect foundation for this work at the intersection of artificial intelligence and blockchain technology.

Abstract

This Research Application focuses on the application of Large Language Models (LLMs) to detect vulnerabilities in Ethereum smart contracts. The primary objective was to explore how models such as Llama-3-8b-Instruct can assist in identifying security risks, particularly reentrancy attacks, which are common in decentralized finance (DeFi) applications. The work began with a review of the literature to understand the current landscape of smart contract vulnerabilities and LLM applications in security analysis. Using Solidity contracts as input, a Retrieval-Augmented Generation (RAG) system was built around the Llama-3-8b-Instruct model to generate vulnerability assessments. The system was tested on a curated dataset of both vulnerable and secure smart contracts. Evaluation metrics such as accuracy, precision, recall, and the F1 score were used to assess performance. Based on the initial results, iterative improvements were made to enhance the model's effectiveness and reduce false positives. The project demonstrates that LLMs can serve as valuable tools in smart contract auditing, offering scalable and intelligent insights that can complement traditional security reviews.

Contents

Chapter 1

Introduction

1.1 Background on Ethereum and Vulnerabilities

Ethereum is a decentralized, open-source blockchain platform that facilitates the development and execution of smart contracts and decentralized applications (DApps). The rapid rise of blockchain technologies and decentralized finance (DeFi) has revolutionized traditional financial systems by enabling automated, peer-to-peer transactions. However, smart contracts are prone to a range of security vulnerabilities — including integer overflows, reentrancy attacks, and improper access control — which have led to major financial losses and diminished user trust in DeFi platforms [1].

As smart contracts become foundational components of DeFi ecosystems, the importance of implementing strong security measures and advanced auditing techniques grows increasingly critical. Many smart contract exploits arise from subtle coding flaws that compromise the integrity and functionality of decentralized applications. One notable example is the reentrancy vulnerability, where an attacker manipulates control flow by repeatedly calling an external function before the contract updates its internal state, allowing unauthorized, repeated withdrawals of funds.

1.2 Role of LLMs in Security Auditing

Recent advancements in research have investigated the application of Large Language Models (LLMs) in smart contract auditing, capitalizing on their capability to understand and generate code in a semantically meaningful way. Unlike traditional static analysis tools, LLMs can interpret code within its broader context, account for dynamic execution paths, and identify complex vulnerability patterns that might otherwise go undetected [1].

By incorporating techniques such as Retrieval-Augmented Generation (RAG) and model fine-tuning, LLMs can be tailored to adapt to the evolving threat landscape in smart contract security [1]. Their integration into the auditing workflow not only improves the precision of vulnerability detection but also minimizes the burden of manual code review by efficiently reducing false positives.

1.3 Research Objectives

This internship aims to evaluate the performance of a baseline Meta-LLaMA-3-8B-Instruct model in detecting vulnerabilities in Ethereum smart contracts. Building on this foundation, the study explores the enhancement of model performance through Retrieval-Augmented Generation (RAG), which provides additional contextual information during inference.

By comparing key evaluation metrics—such as accuracy, precision, recall, and F1-score—between the base and RAG-enhanced models, this work demonstrates the potential of LLMs as effective tools for automated smart contract auditing and security verification.

Chapter 2

Literature Review

2.1 Smart Contract Exploits

2.1.1 Smart Contract Vulnerabilities

Smart contract vulnerabilities pose significant threats to the security of blockchain systems. Exploiting these weaknesses can enable malicious actors to perform unauthorized transactions, drain funds, or trigger unintended behaviors in decentralized applications (dApps).

One of the most prominent and well-documented vulnerabilities is the *reentrancy attack*, in which an attacker repeatedly invokes a contract's function before the previous execution is completed. This recursive execution can manipulate the contract's internal state and enable the unauthorized withdrawal of assets.

Beyond reentrancy, other critical vulnerabilities include:

- **Integer overflow and underflow:** Arithmetic operations that exceed or fall below the limits of their data types, potentially resulting in incorrect logic or exploitable conditions.
- **Unchecked external calls:** Failing to verify the success or behavior of calls to external contracts, which can introduce unexpected state changes or denial-of-service risks.
- **Gas limit manipulation:** Exploiting transaction gas limits to disrupt contract execution or prevent fallback functions from operating correctly.

These vulnerabilities compromise not only individual smart contracts but also the reliability, integrity, and trustworthiness of the broader blockchain ecosystem.

2.1.2 Reentrancy Attacks in Solidity

Reentrancy attacks are a critical vulnerability in smart contracts, particularly in those developed using Solidity. In such an attack, a malicious contract repeatedly invokes a function in the target contract before the previous execution is completed, allowing the attacker to drain funds or manipulate the contract state.

This vulnerability occurs when a contract relinquishes control flow by making an external call, especially to an untrusted or malicious contract, before updating its state. This opens a window for the external contract to recursively call back into the vulnerable function.

Types of Reentrancy Attacks

1. Single-Function Reentrancy A single-function reentrancy attack occurs when the same function is recursively called before the contract's state is updated. This is typically easier to exploit and detect than cross-function reentrancy.

How it works:

1. The target contract has a function that:
   - Verifies a condition (e.g., sufficient balance),
   - Makes an external call (e.g., transfers Ether),
   - Updates the contract's state (e.g., deducts the user's balance).
2. The attacker's contract uses a `fallback()` or `receive()` function to call the same function in the target contract recursively.
3. This allows the attacker to bypass the state update and repeatedly exploit the function.

2. Cross-Function Reentrancy In cross-function reentrancy, the attacker takes advantage of shared state across multiple functions. The vulnerable function might not perform the external call directly, but another function that shares its state does.

How it works:

1. The target contract includes two or more functions that share a common state (e.g., a `balances` mapping).
2. One function performs an external call without securing the shared state.
3. The attacker's contract reenters by calling a different function that also manipulates the shared state.
4. This results in inconsistent or exploitable changes in contract logic.

Preventing Reentrancy Attacks

Best practices to defend against reentrancy vulnerabilities include:

- **Checks-Effects-Interactions Pattern:** Always update the contract's internal state before making external calls.
- **Use Reentrancy Guards:** Utilize constructs such as OpenZeppelin's `ReentrancyGuard` modifier or mutex locks to block reentrant calls.

- **Avoid Low-Level External Calls:** Prefer using `transfer()` or `send()` over `call()` when transferring Ether, as they forward limited gas and are safer.
- **Static Analysis Tools:** Tools such as Slither, Mithril, Oyente, or LLM-based vulnerability scanners can be used to detect potential reentrancy issues during the development phase.

Illustration of Reentrancy Attack

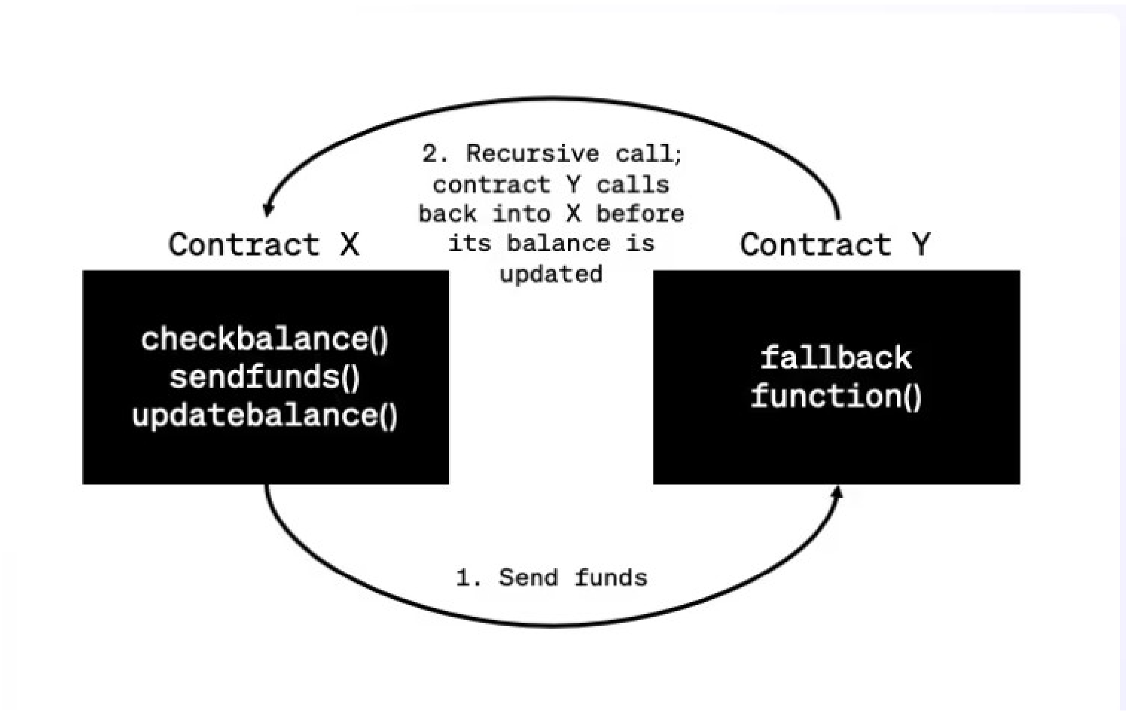

Figure 2.1: Illustration of a Reentrancy Attack Flow in Solidity [9]

2.2 Large Language Models (LLMs)

2.2.1 Definition

Large Language Models (LLMs) represent a significant advancement in artificial intelligence, particularly within the domain of natural language processing. These models are capable of generating human-like text with a high degree of fluency and coherence. Built upon transformer architecture, LLMs are trained on extensive corpora comprising diverse and large-scale datasets. This enables them to capture complex linguistic patterns involving syntax, semantics, and contextual understanding.

LLMs are highly adaptable through fine-tuning processes, allowing them to be customized for specific downstream tasks such as translation, summarization, and conversational agents. Core mechanisms such as self-attention and positional encoding enable these models to model long-range dependencies and generate contextually relevant responses. As a result, LLMs have transformed AI-driven applications across numerous domains, thereby enhancing human-computer interaction and automating content generation.

2.2.2 Key Applications of Large Language Models

This section outlines the primary domains in which Large Language Models (LLMs) are making substantial contributions, highlighting their significance in both research and industry.

Natural Language Understanding (NLU)

LLMs exhibit robust performance in natural language understanding tasks, including sentiment analysis, named entity recognition, and text classification. These capabilities allow models to interpret semantic meaning and contextual relationships effectively. Real-world applications include tracking public sentiment on social media, analyzing customer feedback, and automating content categorization.

Natural Language Generation (NLG)

One of the hallmark features of LLMs is their proficiency in generating coherent and contextually relevant text. Applications of NLG span content creation, dialogue systems, and virtual assistants. For instance, LLMs can automatically generate product descriptions, craft personalized email responses, and create dynamic scripts for interactive applications.

Information Retrieval and Summarization

LLMs are instrumental in extracting relevant information from large volumes of unstructured data and generating concise summaries. This functionality is widely employed in summarizing news articles, academic literature, and business reports. By distilling complex information into digestible formats, LLMs support quicker decision-making and improved comprehension.

Language Translation

Advancements in machine translation have been significantly influenced by LLMs, which provide improved accuracy and context-aware translations across multiple languages. This has enabled seamless cross-lingual communication in various sectors, including international commerce, diplomacy, and global customer support services.

Dialogue Systems and Conversational Agents

LLMs form the backbone of intelligent chatbots and conversational systems that provide personalized assistance, respond to user queries, and provide real-time support. Their capacity to handle multi-turn dialogue and maintain context over conversations enhances both user satisfaction and operational effectiveness.

Code Generation and Software Development

In software engineering, LLMs are increasingly being used to support the analysis of codebases, automated code generation, bug detection, and adherence to coding standards. These tools assist in accelerating development cycles, ensuring code quality, and simplifying system maintenance.

Scientific Research and Knowledge Discovery

In academic and research contexts, LLMs facilitate efficient review of the literature, generate plausible hypotheses, and help interpret complex scientific data. Their ability to manage and process domain-specific language enhances research productivity and supports the discovery of emerging trends[4].

2.2.3 LLaMA (Large Language Model Meta AI)

LLaMA (Large Language Model Meta AI) is a series of large language models designed to achieve state-of-the-art performance with optimized computational efficiency [1]. The LLaMA 3 model introduces significant improvements in both contextual understanding and computational efficiency, making it particularly suitable for tasks that require deep code analysis and reasoning.

In the domain of smart contract auditing, the ability of LLaMA 3 to process long sequences and identify complex patterns in code establishes it as a powerful analytical tool. When integrated with fine-tuning techniques and domain-specific knowledge bases, the model can effectively detect vulnerabilities that are often missed by conventional static analysis tools.

2.2.4 Retrieval-Augmented Generation with LLMs

The emergence of Large Language Models (LLMs), such as Meta LLaMA, has introduced new possibilities for automating aspects of the smart contract auditing process. LLMs have demonstrated exceptional capabilities in understanding and generating human-like text, making them suitable for tasks involving code analysis and pattern recognition. However, despite their strengths, LLMs are inherently limited by the scope of their pretraining data and may lack the domain-specific precision required for comprehensive smart contract auditing.

To overcome these limitations, the concept of *Retrieval-Augmented Generation* (RAG) has been proposed. RAG enhances the capabilities of LLMs by integrating them with retrieval mechanisms that access a curated knowledge base or vector store. This hybrid approach enables the model to generate more accurate, fact-based, and contextually relevant outputs by referencing external sources of information during inference.

In the specific context of smart contract auditing, a RAG-LLM system can retrieve documented examples of known vulnerabilities from a pre-indexed vector store. This capability significantly improves the model's ability to detect similar security issues in newly written contracts. By incorporating this technology, we aim to develop a scalable and cost-effective solution that democratizes access to automated smart contract security auditing.

The integration of RAG with LLMs extends the generative capacity of the model by providing real-time access to external knowledge beyond its initial training corpus. RAG operates by enriching the prompt context with retrieved content, thereby increasing the model's effective knowledge base during inference. This dual-retrieval mechanism—leveraging both the model's internal parameters and external databases—allows RAG-enhanced LLMs to deliver more accurate and contextualized responses than traditional LLMs, particularly in specialized or evolving domains such as blockchain security[5].

Chapter 3

Research Methodology

3.1 Literature Review and Problem Definition

An initial review of existing tools, datasets, and documented smart contract vulnerabilities—such as reentrancy, integer overflow and underflow, and unchecked external calls—was conducted. This review provided insights into how vulnerabilities are typically identified using traditional static and dynamic analysis techniques. Furthermore, it served as a foundational step in exploring how Large Language Models (LLMs) could potentially assist in automating and enhancing the vulnerability detection process.

3.2 Dataset Collection and Structure

A dataset of smart contracts written in Solidity was compiled, comprising both vulnerable and non-vulnerable examples. These contracts were sourced from public vulnerability databases, academic literature, and curated code repositories. Each contract was manually labeled for evaluation purposes, enabling supervised learning and performance assessment.

The final dataset consists of 40 distinct smart contract samples, evenly divided into two categories: 20 vulnerable and 20 non-vulnerable contracts. To evaluate the detection capability of the model, each contract was tested using 5 distinct structured prompts. Furthermore, each prompt was executed 10 times to account for variability in the LLM's responses, resulting in 50 test instances per contract.

- Total vulnerable contract inputs: $20 \times 50 = 1000$
- Total non-vulnerable contract inputs: $20 \times 50 = 1000$

In total, the evaluation involved **2000 inputs**, which provided a balanced and comprehensive testbed for assessing the model's performance on vulnerability detection tasks.

Table 3.1: Summary of Smart Contract Dataset

Category	**Number of Contracts**	**Test Inputs (5 prompts $\times$ 10 runs)**
Vulnerable	20	1000
Non-Vulnerable	20	1000
Total	**40**	**2000**

3.3 Data Preprocessing

To enable effective input processing for both the base LLaMA model and the RAG-enhanced LLaMA system, two distinct data preprocessing pipelines were implemented. For the base LLaMA model, the objective was to standardize and tokenize smart contract data for direct prompt-based classification without external context. In contrast, the RAG-enhanced LLaMA pipeline aimed to augment this input by integrating domain-specific knowledge retrieved from a pre-indexed vector database. This allowed the model to reference similar examples during inference. While both pipelines shared foundational steps such as contract parsing and prompt formatting, the RAG-based approach incorporated additional retrieval and context construction components to support knowledge-aware reasoning.
This section outlines the specific steps implemented for each setup.

3.3.1 Prompt Design Rationale

To systematically evaluate the models' capability in detecting various forms of reentrancy vulnerabilities, we crafted five distinct prompts. Each prompt was inspired by a specific type of reentrancy attack or mitigation pattern, including:

- Single-function reentrancy
- Cross-function reentrancy
- Use of Checks-Effects-Interactions (CEI) pattern
- Reentrancy guards (e.g., mutex, `noReentrancy` modifier)
- Use of low-level calls such as `call()`, `send()`, and `transfer()`

Every smart contract was evaluated using all five prompts to test detection consistency under various contextual interpretations. This allowed for a more granular understanding of model performance across different attack vectors.

3.3.2 Base LLaMA Model

The base LLaMA model was provided only with smart contract code and structured prompts. No retrieval or external context was used. The inputs were manually curated and formatted into prompt-code pairs. The preprocessing included tokenization, truncation, and repeated inference to assess prediction consistency. The preprocessing steps for this setup were as follows:

- **Prompt Engineering and Formatting**: Each contract was paired with five structured prompts aimed at testing its susceptibility to reentrancy vulnerabilities.
  An example of the prompt used is as follows:

  ```
  [INST] As a smart contract security expert,
  Analyze the following Solidity smart contract and determine whether
  it follows the Checks-Effects-Interactions (CEI) pattern to prevent
  reentrancy? Then, provide a short reasoning explaining your decision.
  Format your answer exactly like this:
  Prediction: <vulnerable/not vulnerable>
  Reasoning: <short explanation>
  Smart contract: {contract_code}
  ```

  The model was expected to return a two-line structured response indicating whether the contract was vulnerable and a brief justification. This prompt format was applied uniformly across all contracts during the base LLaMA evaluation.

- **Tokenization and Truncation**: The prompt and contract code were tokenized and truncated to fit within a 2048-token limit using the `transformers` tokenizer. Padding and attention masks were applied as needed.

- **Inference Reproducibility**: Each prompt-contract pair was tested ten times to observe consistency and variation in outputs.

3.3.3 RAG-Integrated LLaMA Model

The RAG-enhanced LLaMA system required additional preprocessing steps to support knowledge retrieval from an external database of annotated contracts. These steps are described in detail below.

PDF Text Extraction and Parsing

The dataset was initially stored in PDF format, containing annotated smart contracts. The `PyPDF2` library was used to extract raw text from the document. Contracts were embedded in a standardized markdown structure with metadata such as contract name, label (vulnerable or non-vulnerable), category, and explanation. A regular expression parser was implemented in Python to extract these structured fields:

- **Name**: Unique identifier of the contract.
- **Label**: Indicates whether the contract is vulnerable.
- **Category**: Classification of contract use-case (e.g., Escrow, Wallet).
- **Explanation**: Brief description of the contract's logic.
- **Code**: Solidity source code of the contract.

Embedding and Vector Indexing

To support Retrieval-Augmented Generation (RAG), each contract's metadata and code were concatenated into a single text block. These blocks were then embedded using the `all-MiniLM-L6-v2` model from the `sentence-transformers` library. The resulting vector representations were indexed using FAISS (Facebook AI Similarity Search), enabling efficient semantic retrieval during inference.

Prompt Engineering and Formatting

For the RAG-integrated LLaMA model, each prompt was dynamically generated by combining retrieved context passages with a new smart contract under analysis. The prompt followed a structured template that instructed the model to classify the contract's vulnerability status specifically concerning reentrancy.

The template emphasized a consistent response format and encouraged the use of context when available. An example prompt structure is shown below:

```
[INST]
As a smart contract security expert,
Classify the following smart contract as either "vulnerable" or "not vulnerable" to
Use your knowledge and refer to similar patterns from context if needed.

Format your answer exactly like this:

Prediction: <vulnerable/not vulnerable>
Reasoning: <short explanation>

Context for reference:
{context}

Smart contract to analyze:
{contract_code}
```

An example of a smart contract input embedded in the prompt is as follows:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract Rewards {
    mapping(address => uint) public ledger_480;
    bool private locked;

    function deposit() public payable {
        ledger_480[msg.sender] += msg.value;
    }

    function registerRewards() public {
        require(ledger_480[msg.sender] > 0, "No deposit");
        withdrawRewards();
    }

    function withdrawRewards() public {
        require(ledger_480[msg.sender] > 0, "No balance");
        (bool success, ) = msg.sender.call{value: ledger_480[msg.sender]}("");
        require(success);
        ledger_480[msg.sender] = 0;
    }
}
```

The dynamic portion of the prompt was generated by retrieving semantically similar contracts from the FAISS-based knowledge base and formatting them into the ‘context‘ block. This augmented context enabled the model to compare known patterns with the target contract and generate more informed predictions.

Tokenization and Truncation

To ensure compatibility with the token limits of Meta LLaMA-3-8B-Instruct, the input prompt (including context) was tokenized and truncated to a maximum input length of 2048 tokens. The `transformers` tokenizer was used for this purpose, with appropriate padding and attention masks applied during generation.

Inference Reproducibility

To account for generation variability in language models, each contract and prompt combination was tested ten times, resulting in 50 test instances per contract. This setup allowed us to evaluate not only the accuracy of the models but also their consistency across multiple runs.

Summary of Preprocessing Pipeline

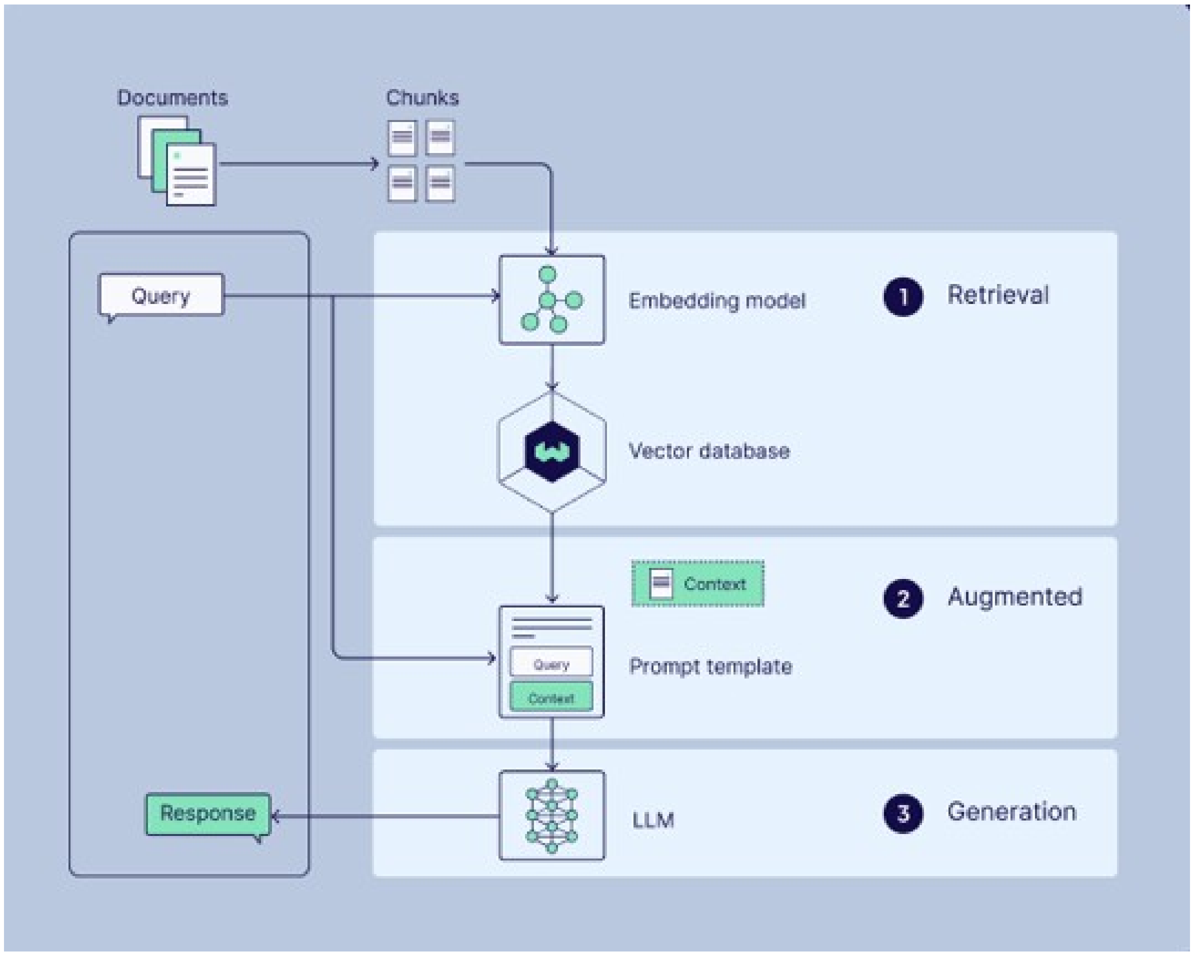

Figure 3.1: Preprocessing and Inference Pipeline for RAG-integrated LLaMA Model [10]

3.4 Model Loading

All experiments were conducted using **Google Colab Pro** as the development and execution environment, which provided access to high-RAM virtual machines with GPU acceleration. This setup allowed for efficient testing and deployment of large-scale language models.

The **Meta LLaMA 3 8B Instruct** model was loaded via the `transformers` library from **Hugging Face**, using a locally downloaded copy stored on Google Drive. Quantization settings such as 8-bit loading were applied using `BitsAndBytesConfig` to optimize memory consumption and enable faster inference.

Hugging Face's `AutoTokenizer` and `AutoModelForCausalLM` classes were used to initialize the LLaMA model, while the `SentenceTransformer` model `all-MiniLM-L6-v2` was employed for vector embedding during RAG integration.

All model inference, embedding, vector indexing (via FAISS), and testing procedures were implemented using Python 3.10 in Colab notebooks.

3.5 Evaluation Metrics

To assess the performance of both the base LLaMA model and the RAG-integrated LLaMA system in classifying smart contracts as vulnerable or not vulnerable to reentrancy attacks, we utilized standard classification evaluation metrics. These metrics provide a comprehensive view of model behavior across correct and incorrect predictions.

Let the following terms represent the outcomes of model predictions:

Confusion Matrix

To evaluate the performance of a binary classifier, such as in smart contract vulnerability detection (vulnerable vs. not vulnerable), a **confusion matrix** is used. It summarizes the outcomes of model predictions by comparing the predicted labels against the true (actual) labels. The confusion matrix is composed of four elements:

- **True Positive (TP)**: The contract is vulnerable and is correctly predicted as vulnerable.
  Example: $(1, 1)$ — the model correctly identifies a vulnerability.
- **False Negative (FN)**: The contract is actually vulnerable but is incorrectly predicted as not vulnerable.
  Example: $(1, 0)$ — the model misses the vulnerability.
- **False Positive (FP)**: The contract is actually not vulnerable but is incorrectly predicted as vulnerable.
  Example: $(0, 1)$ — the model raises a false alarm.
- **True Negative (TN)**: The contract is not vulnerable and is correctly predicted as not vulnerable.
  Example: $(0, 0)$ — the model correctly identifies no vulnerability.

Table 3.2: Confusion Matrix Structure

Actual / Predicted	**Vulnerable (1)**	**Not Vulnerable (0)**
Vulnerable (1)	True Positive (TP)	False Negative (FN)
Not Vulnerable (0)	False Positive (FP)	True Negative (TN)

This matrix provides the foundation for computing key performance metrics such as accuracy, precision, recall, and F1 score. It is particularly useful in understanding where the model tends to make mistakes and how well it distinguishes between vulnerable and non-vulnerable smart contracts.

Accuracy

Accuracy measures the overall correctness of the model across all predictions.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1}$$

Precision

Precision measures the proportion of correctly predicted vulnerable contracts out of all contracts predicted as vulnerable.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.2}$$

Recall (True Positive Rate)

Recall, also known as Sensitivity or True Positive Rate, measures the proportion of actual vulnerable contracts correctly identified by the model.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.3}$$

F1 Score

The F1 Score is the harmonic mean of Precision and Recall. It provides a balanced metric when dealing with imbalanced datasets.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.4}$$

Specificity (True Negative Rate)

Specificity measures the proportion of actual non-vulnerable contracts correctly identified.

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{3.5}$$

False Positive Rate (FPR)

FPR measures the rate at which non-vulnerable contracts are incorrectly classified as vulnerable.

$$\text{FPR} = \frac{FP}{FP + TN} \tag{3.6}$$

False Negative Rate (FNR)

FNR measures the rate at which vulnerable contracts are incorrectly classified as not vulnerable.

$$\text{FNR} = \frac{FN}{FN + TP} \tag{3.7}$$

These evaluation metrics collectively help measure not just how often the model is right, but also how it handles errors, particularly important in security-sensitive applications like smart contract auditing [1].

Chapter 4

Experimental Results

4.1 Confusion Matrices

To evaluate the performance of both the baseline and RAG-enhanced LLaMA models, confusion matrices were generated. These matrices summarize the number of true positives (TP), false negatives (FN), true negatives (TN), and false positives (FP) in vulnerability classification.

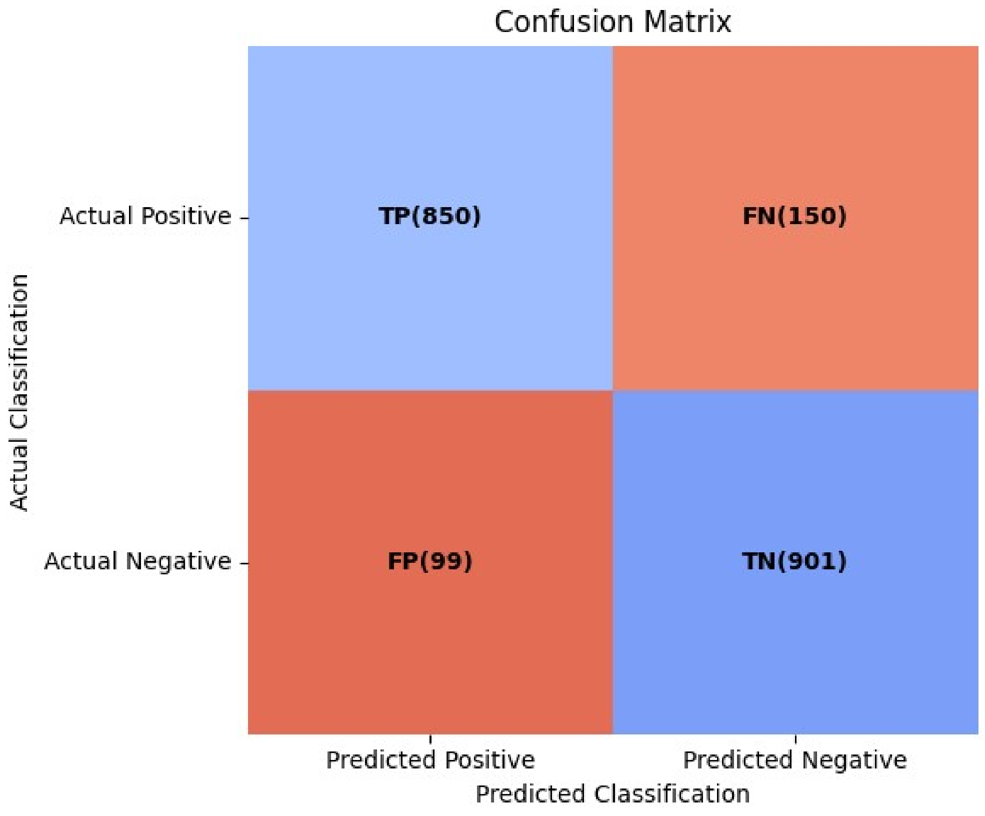

Figure 4.1: Confusion Matrix for LLaMA (Base) Model

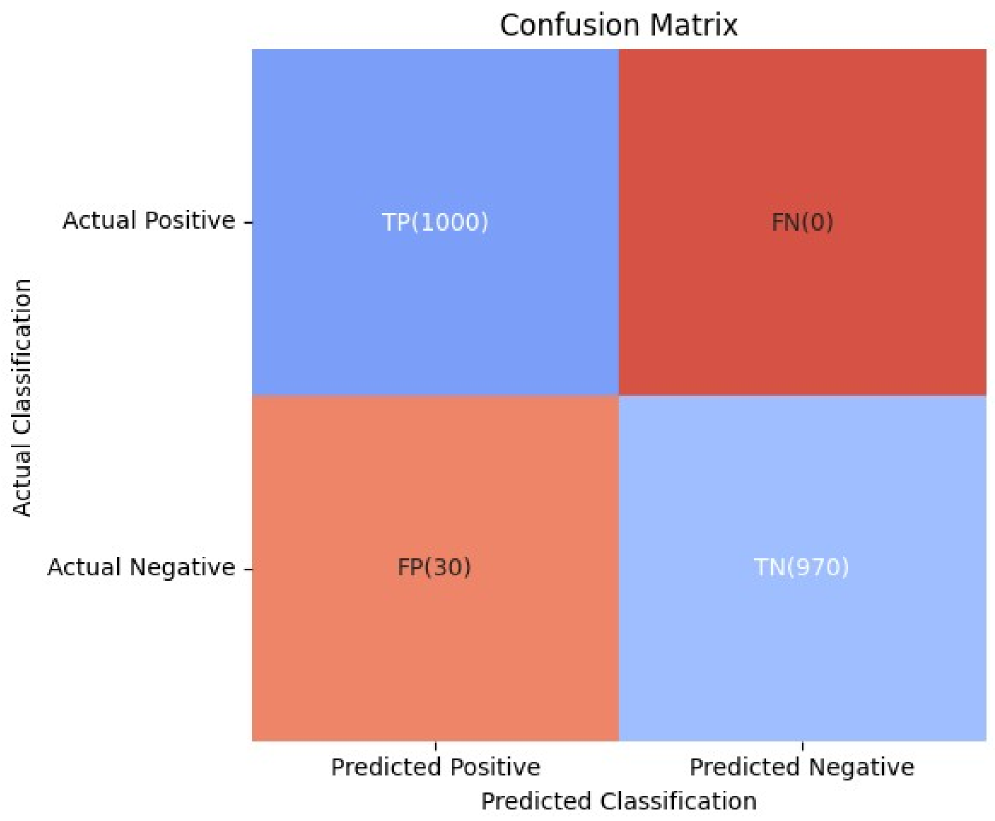

Figure 4.2: Confusion Matrix for RAG-Integrated LLaMA Model

The baseline LLaMA model achieved 850 true positives and 901 true negatives, while the RAG-enhanced model correctly identified all 1,000 vulnerable contracts with fewer false positives, demonstrating its superior detection capability.

Model	**TP**	**FN**	**TN**	**FP**
LLaMA (Base)	850	150	901	99
RAG-LLaMA (Enhanced)	1000	0	970	30

4.2 Performance Metrics Comparison

Table 4.1 presents a detailed comparison of key performance metrics. The RAG-integrated model shows substantial improvements in all major metrics, most notably achieving a 100% recall rate and significantly reducing both false positive and false negative rates.

4.3 Visual Representation

To illustrate the comparative results, two bar charts were created using Python and Matplotlib.

Table 4.1: Comparison of Performance Metrics for LLaMA and RAG-Integrated LLaMA

Metric	**LLaMA (Base)**	**RAG-Integrated LLaMA**	**Improvement**
Accuracy	87.55%	98.50%	+10.95%
Precision	89.57%	97.09%	+7.52%
Recall	85.00%	100.00%	+15.00%
F1 Score	87.22%	98.52%	+11.30%
Specificity	90.10%	97.00%	+6.90%
FPR (↓)	9.90%	3.00%	-6.90%
FNR (↓)	15.00%	0.00%	-15.00%

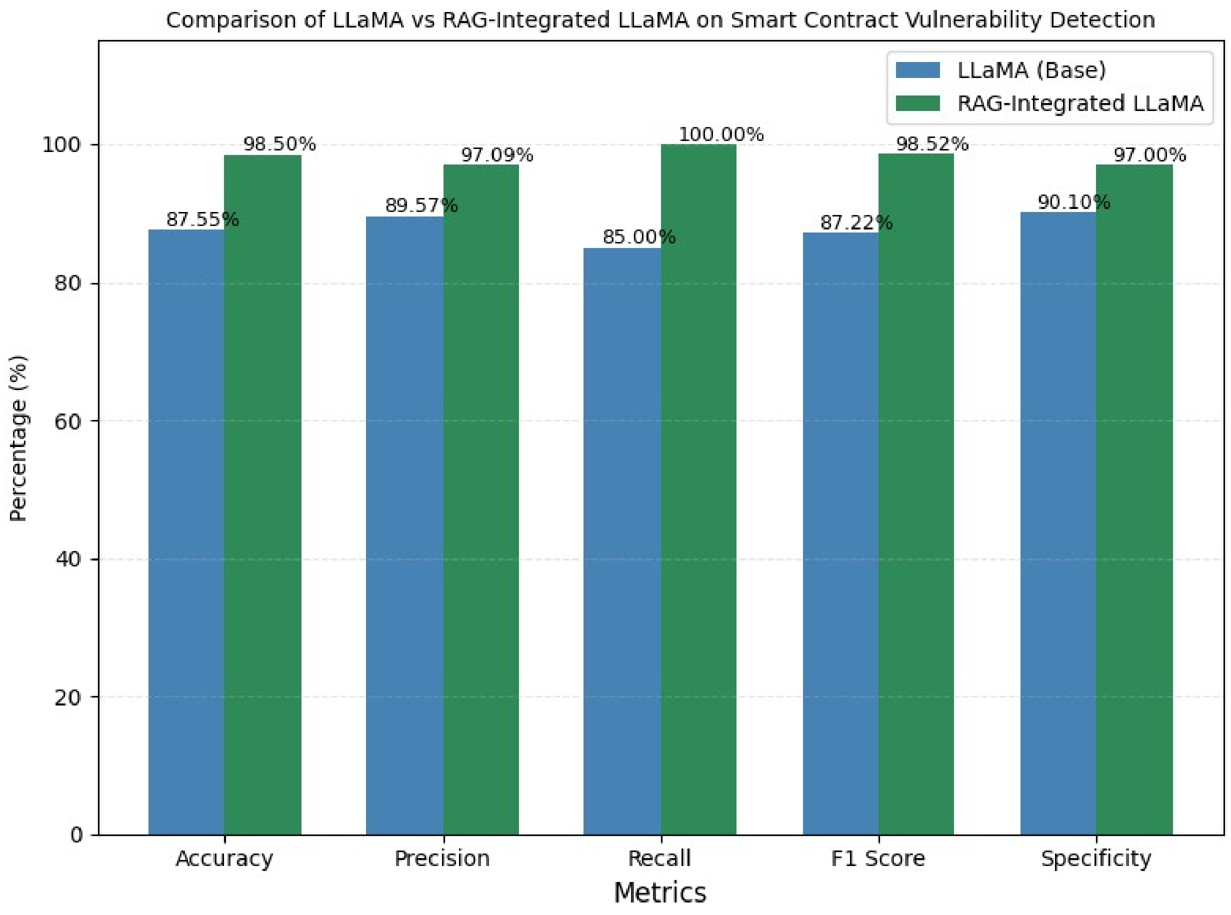

Figure 4.3: Comparison of Positive Performance Metrics (↑ is Better)

As seen in Figures 4.3 and 4.4, the RAG-Integrated LLaMA model consistently outperforms the baseline in accuracy, precision, recall, F1 score, and specificity. Furthermore, the reductions in FPR and FNR reinforce its reliability in both identifying true vulnerabilities and avoiding false alarms.

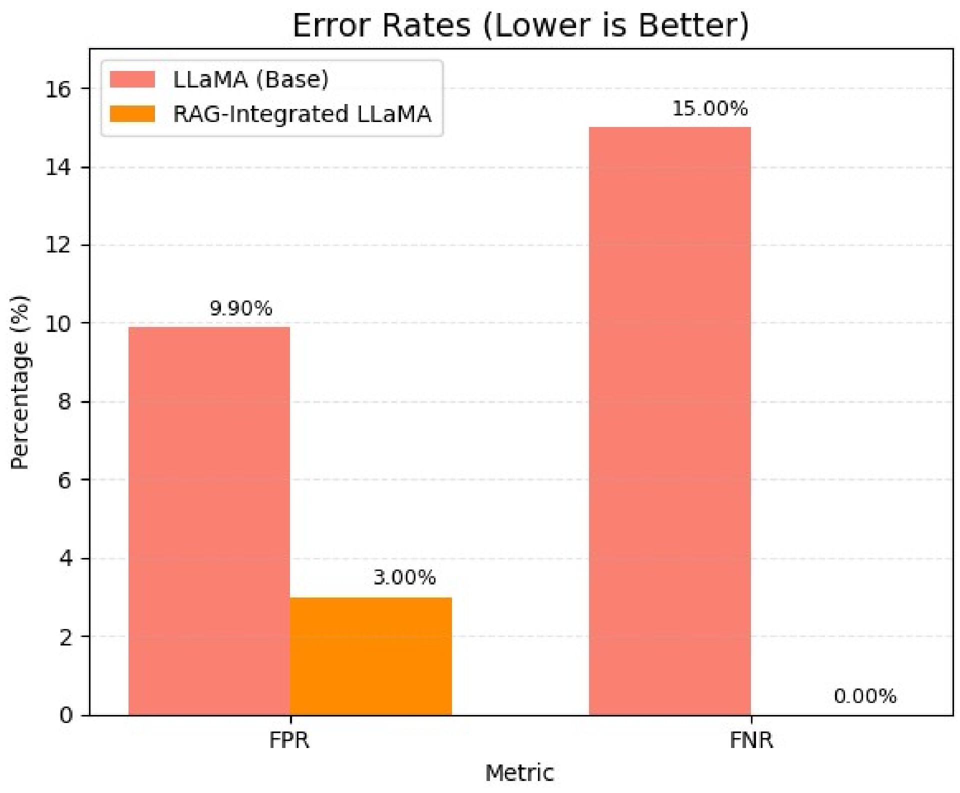

Figure 4.4: Comparison of Error Metrics (↓ is Better)

Chapter 5

Analysis and Discussion

5.1 Model Performance Overview

The baseline LLaMA model demonstrated strong overall performance, achieving an accuracy of 87.55% and a precision of 89.57%. These metrics indicate that the model is reasonably capable of identifying vulnerable smart contracts. However, one critical shortcoming lies in its **recall rate of 85.00%**, which means it failed to detect 15% of actual vulnerabilities (FNR = 15%). In the context of smart contract security, such omissions can have severe financial and operational consequences.

In contrast, the RAG-integrated LLaMA model achieved significant improvements across all key metrics. Most notably:

- It reached **100% recall**, ensuring that no vulnerable contracts were missed.
- The **false positive rate (FPR)** dropped from 9.90% to 3.00%, a 6.9% reduction.
- The **false negative rate (FNR)** decreased from 15.00% to 0.00%, a complete elimination.
- It achieved an **F1 Score of 98.52%**, indicating a highly balanced performance in terms of both precision and recall.

These gains suggest that integrating RAG into the LLaMA architecture can enhance not only the model's sensitivity to vulnerabilities but also its precision in avoiding false alarms.

5.2 Why RAG Enhances Detection

The substantial improvement in performance with RAG integration can be attributed to its ability to provide contextual grounding during inference. RAG augments the input by retrieving semantically relevant passages or contract fragments before generating predictions.

In the domain of smart contracts, vulnerabilities such as reentrancy or improper access control often depend on **inter-function dependencies**, **state variable manipulations**, and **storage access patterns**. These are difficult to detect in isolation without understanding the broader execution context.

- **Function dependencies:** RAG helps in identifying logical links between deposit, withdraw, and fallback functions that are commonly exploited in reentrancy attacks.
- **Storage variables:** By retrieving related code blocks, the model is more aware of how balances, mappings, or mutexes are manipulated across different parts of the contract.
- **Execution sequence:** Contextual snippets help the model simulate the correct order of operations, enhancing its ability to spot misplaced effects (e.g., modifying state after external calls).

Thus, RAG provides a richer semantic understanding, allowing the model to reason more effectively about code behavior beyond individual functions.

5.3 Opportunities

The successful integration of LLMs with RAG for smart contract analysis opens several promising avenues:

- **Automated Auditing:** LLMs can significantly reduce the time and cost involved in manual smart contract audits.
- **Explainability:** Models can be prompted to explain their reasoning, which helps developers understand potential issues in their code.
- **Fine-tuning on Domain-Specific Data:** Further improvements can be achieved by fine-tuning on curated vulnerability datasets, including real-world exploits and test suites.
- **Integration into IDEs and CI/CD Pipelines:** LLM-powered security checks can be embedded directly into developer environments, providing real-time feedback [1].

5.4 Challenges and Limitations

Despite these advancements, several challenges remain:

- **Hallucination:** LLMs can sometimes generate incorrect or unsupported claims, especially in ambiguous or poorly documented code.
- **Dependence on Quality Retrieval:** RAG's performance depends heavily on the quality of the retrieved context. Irrelevant or redundant context may mislead the model.
- **Scalability:** As smart contracts grow in complexity, the cost of embedding and retrieving large numbers of contract chunks increases.
- **Lack of Formal Guarantees:** Unlike formal verification tools, LLM-based systems do not provide mathematical proof of safety and correctness [1].

5.5 Summary

The RAG-integrated LLaMA model outperforms the baseline significantly across all evaluation metrics, particularly in recall and error reduction. Its success in identifying complex vulnerabilities highlights the value of combining language models with contextual retrieval for code analysis. However, careful design, evaluation, and further enhancements are needed before such systems can be relied upon in high-stakes production environments.

Chapter 6

Conclusion

This internship project explored the application of Large Language Models (LLMs), specifically the Meta LLaMA-3-8B-Instruct model, for detecting vulnerabilities in Ethereum smart contracts, with a focus on reentrancy attacks. The study demonstrated that LLMs can serve as powerful tools for automated smart contract auditing when enhanced with Retrieval-Augmented Generation (RAG) techniques.

Key Findings

The experimental results revealed several important insights:

- The baseline LLaMA model achieved respectable performance with 87.55% accuracy and 89.57% precision, but showed limitations with 85% recall, missing 15% of actual vulnerabilities.
- The RAG-enhanced model showed significant improvements across all metrics:
  - 100% recall (eliminating false negatives)
  - 98.50% accuracy
  - 97.09% precision
  - 98.52% F1 score
- The integration of RAG reduced the false positive rate from 9.90% to 3.00% and eliminated false negatives.
- Contextual retrieval proved particularly valuable for understanding inter-function dependencies and state variable manipulations that characterize reentrancy vulnerabilities.

Contributions

This work makes several contributions to the field of smart contract security:

- Demonstrated the effectiveness of LLMs for vulnerability detection in smart contracts

- Showed that RAG can significantly enhance model performance by providing relevant contextual information
- Developed a reproducible methodology for evaluating LLM-based security analysis tools
- Created a curated dataset of vulnerable and non-vulnerable contracts for benchmarking

Future Work

Several promising directions for future research emerged from this project:

- Extending the approach to detect other types of vulnerabilities beyond reentrancy
- Investigating fine-tuning strategies to further improve model performance
- Developing techniques to reduce model hallucination in security contexts
- Exploring integration with development environments and CI/CD pipelines
- Combining LLM-based analysis with formal verification methods for stronger guarantees

Final Remarks

The success of the RAG-enhanced LLaMA model in this project suggests that LLMs have significant potential to transform smart contract auditing practices. While challenges remain, particularly around hallucination and formal guarantees, these models can already serve as valuable assistants to human auditors, helping to identify potential vulnerabilities more efficiently. As the technology continues to mature, we anticipate LLM-based tools will play an increasingly important role in securing the decentralized finance ecosystem.

Bibliography

[1] Jun Kevin and Pujianto Yugopuspito. *SmartLLM: Smart Contract Auditing using Custom Generative AI*. Universitas Pelita Harapan, Jakarta, Indonesia. arXiv:2502.13167v1 [cs.CR], 17 Feb 2025.

[2] Zongwei Li, Xiaoqi Li, Wenkai Li, and Xin Wang. *SCALM: Detecting Bad Practices in Smart Contracts Through LLMs*. The Thirty-Ninth AAAI Conference on Artificial Intelligence (AAAI-25), 2025.

[3] Binbin Zhao, Saman Zonouz, Xingshuang Lin, Na Ruan, Raheem Beyah, Yuan Tian, Jiliang Li, and Shouling Ji. *Detecting Functional Bugs in Smart Contracts through LLM-Powered and Bug-Oriented Composite Analysis*. arXiv:2503.23718v1 [cs.SE], 31 Mar 2025.

[4] Zheyuan He, Zihao Li, Sen Yang, He Ye, Ao Qiao, Xiaosong Zhang, Ting Chen, and Xiapu Luo. *Large Language Models for Blockchain Security: A Systematic Literature Review*. arXiv:2403.14280v5 [cs.CR], 24 Mar 2025.

[5] Jeffy Yu. *Retrieval Augmented Generation Integrated Large Language Models in Smart Contract Vulnerability Detection*. Parallel Polis, San Francisco State University. arXiv:2407.14838v1 [cs.CR], 20 Jul 2024.

[6] Salam Al-E'mari and Yousef Sanjalawe. *A Review of Reentrancy Attack in Ethereum Smart Contracts*. License: CC BY 4.0.

[7] Mykola Striletskyy. *How To Build an AI Knowledge Base With RAG*. DZone – Data Engineering AI/ML, Jul. 9, 2024.

[8] Ayman Alkhalifah, Alex Ng, Paul A. Watters, and A. S. M. Kayes. *A Mechanism to Detect and Prevent Ethereum Blockchain Smart Contract Reentrancy Attacks*. Frontiers in Computer Science, Volume 3, 17 February 2021. https://doi.org/10.3389/fcomp.2021.598780.

[9] Shray Jain. *What is a Reentrancy Attack in Solidity?* Reviewed by Brady Werkheiser. Published on Alchemy, October 4, 2022.

[10] Mary Newhauser. *Introduction to Retrieval Augmented Generation (RAG)*. Weaviate, October 15, 2024.