BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

SecRAG: A Retrieval-Augmented and LLM-Driven Framework for Function Level Code Vulnerability Detection and CWE Classification

A Z Hasnain Kabir
200042102
Mamunur Rahman
200042116
Mukit Mahdin
200042170

Department of Computer Science and Engineering
Islamic University of Technology
September, 2025

SecRAG: A Retrieval-Augmented and LLM-Driven Framework for Function Level Code Vulnerability Detection and CWE Classification

A Z Hasnain Kabir
200042102
Mamunur Rahman
200042116
Mukit Mahdin
200042170

Department of Computer Science and Engineering
Islamic University of Technology
September, 2025

Declaration of Candidate

This is to certify that the work presented in this thesis is the outcome of the analysis and experiments carried out by **A Z Hasnain Kabir**, **Mamunur Rahman**, and **Mukit Mahdin** under the supervision of **Dr. Md Moniruzzaman**, Assistant Professor, Department of Computer Science and Engineering, Islamic University of Technology, Dhaka, Bangladesh. It is also declared that neither this thesis nor any part of it has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others have been acknowledged in the text and a list of references is given.

Dr. Md Moniruzzaman
Assistant Professor
Department of Computer Science and Engineering
Islamic University of Technology (IUT)
Date: September 26, 2025

A Z Hasnain Kabir
Student ID: 200042102
Date: September 26, 2025

Mamunur Rahman
Student ID: 200042116
Date: September 26, 2025

Mukit Mahdin
Student ID: 200042170
Date: September 26, 2025

Contents

List of Figures

List of Tables

List of Abbreviations

RAG	Retrieval Augmented Generation
LLM	Large Language Model
PLM	Pre-trained Language Model
BERT	Bidirectional Encoder Representations from Transformers
RoBERTa	Robustly Optimized BERT Approach
CWE	Common Weakness Enumeration
CVE	Common Vulnerabilities and Exposures
LLaMA	Large Language Model Meta AI
UniXcoder	Unified Cross-Modal Code Representation Model
API	Application Programming Interface
GPU	Graphics Processing Unit
NLP	Natural Language Processing
AST	Abstract Syntax Tree
GNN	Graph Neural Network
NL–PL	Natural Language–Programming Language
CoT	Chain of Thought
BM25	Best Matching 25
RRF	Reciprocal Rank Fusion
AdamW	Adaptive Moment Estimation with Weight Decay
F1	Harmonic Mean of Precision and Recall
CI/CD	Continuous Integration / Continuous Deployment
MITRE	MITRE Corporation
T4	NVIDIA Tesla T4 GPU
RQ	Research Question

Acknowledgement

We are profoundly grateful and extend our heartfelt appreciation to our supervisor, **Dr. Md Moniruzzaman**, for his patience, invaluable support, guidance and his insightful critiques throughout the duration of our research.

Abstract

Static code vulnerability detection and CWE classification are fundamentally challenging due to noisy labels, skewed distributions, and a lack of grounded evidence in model predictions. Traditional fine-tuning of Pre-trained Language Models (PLMs) incurs significant time and resource costs and is not easily expandable to dynamic or new vulnerability categories. We introduce SecRAG, a two-stage, retrieval-augmented LLM powered framework that overcomes this limitation for interpretable, function-level vulnerability analysis. Stage I (Data Curation & Diagnostics) establishes a high-integrity, CWE-aware knowledge base by merging and repairing BigVul and PrimeVul data. This involved deduplication, rebalancing non-vulnerable samples, and selecting only well-supported categories, resulting in a 22k+-sample corpus spanning 12 CWE classes. Diagnostics selected UniXcoder as the optimal embedding model (Vulnerability F1 of 74.65%) over CodeBERT and GraphCodeBERT. Stage II (RAG Inference) uses UniXcoder embeddings to retrieve the top-3 nearest vulnerable neighbors, which are then used to constrain downstream LLMs (GPT-4.1, Llama-3.3-70B, etc.) with a parseable prompt schema for final binary and multi-class predictions. Empirically, SecRAG significantly improves CWE-level classification performance over fine-tuned PLM baselines. For instance, GPT-4.1 achieved a CWE Accuracy of 61.37%—a substantial improvement over the best-performing baseline (UniXcoder at 53.26%). The retrieval mechanism preserves competitive binary detection and yields complementary strengths: GPT-4.1 excels in CWE accuracy (61.37%) and Precision (59.69%), while GPT-4o mini offers stronger vulnerability Recall (79.95%). By leveraging clean, CWE-aware retrieval coupled with constrained LLM generation, SecRAG delivers a reproducible, interpretable, and dynamically updatable pipeline where predictions are directly grounded in retrieved evidence.

Chapter 1

Introduction

Software today powers nearly every aspect of our lives - from the apps on our phones to the systems controlling critical infrastructure. Yet as applications grow in size and complexity, hidden security flaws can creep in, waiting to be discovered by attackers. Vulnerable code detection aims to spot those weak spots - tiny mistakes or oversights that, if exploited, could undermine system integrity, leak sensitive data, or cause outages.

To further highlight the critical nature of code vulnerabilities, it must be noted that software vulnerabilities are one of the primary entry points for cyber attacks. moreover, as more and more critical systems rely on software, code security is more importantt now, than ever. the cost and risk of security breaches can be significatly reduced through early detection and remediation, further strenghtening the motivation for this work

In this work, we aim to solve three research questions namely:

- **RQ1:**How effectively can code-specific pre-trained language models (PLMs) detect code vulnerabilities when fine-tuned on a custom, enriched vulnerability dataset?
- **RQ2:**Can Retrieval Augmented Detection techniques improve the accuracy and robustness of code vulnerability detection compared to standard fine-tuned PLMs?
- **RQ3:**How does model's reasoning capacity influence the effectiveness of a RAG-powered vulnerability detection system?

To conduct our research, we introduce **SecRAG**, a multi-stage retrieval-augmented framework that brings together a carefully curated, CWE-enriched dataset, advanced

retrieval mechanisms, and a specialized large language model to deliver both accurate and interpretable vulnerability detection.

Initial attempts at dataset creation, even after merging and deduplication, revealed critical limitations. Specifically, while binary vulnerability detection metrics might appear high, early results demonstrated that these metrics were often inflated and misleading for real-world applications due to underlying data imbalances and a scarcity of sufficient samples for multi-class classification, especially across various Common Weakness Enumeration (CWE) types.

This necessitated a rigorous dataset refinement process to select and focus on a subset of well-represented CWE classes. The significant improvement in CWE classification accuracy achieved with this refined, CWE-enriched dataset compared to earlier merged versions underscores the importance of high-quality, balanced data for developing robust and interpretable vulnerability detection systems. This further validates the approach of building a unified, CWE-aware knowledge base and demonstrates how the proposed framework addresses existing gaps in dataset reliability.

To construct our knowledge base, we merged and refined entries from the PrimeVul [2] and BigVul [4] datasets, resulting in a balanced corpus annotated with twelve key CWE classes - ranging from buffer overflows and improper input validation to use-after-free and integer overflows - each accompanied by official CWE IDs, descriptions, and suggested mitigations. This enriched dataset ensures that our framework learns from high-quality examples and maintains traceability between detected issues and their standardized definitions.

SecRAG's retrieval pipeline begins by converting the input function into a dense vector using UnixCoder [6] embeddings and indexing it in Pinecone - a vector database for approximate nearest-neighbor search. It then retrieves the closest relevant examples through UnixCoder [6].By grounding the subsequent language model query in these retrieved examples, we dramatically reduce the risk of hallucinations and focus the model's reasoning on concrete, domain-relevant evidence. For the inference stage, we leverage both reasoning and non-reasoning LLMs to analyze the assembled context and generate a vulnerability verdict.

1.1 Code Vulnerability and Its Importance

Modern software underpins everything from personal communication and banking to national defense and healthcare. Yet, beneath this backbone of digital life lies a

persistent threat: **code vulnerabilities**. A single overlooked buffer overflow or misconfigured access control can be enough to enable adversaries to exfiltrate sensitive data, disrupt services, or compromise critical infrastructure. Vulnerabilities are not merely bugs - they are potential attack vectors, and their exploitation has repeatedly been linked to billion-dollar damages, loss of trust, and even threats to human safety.

As software systems grow in complexity, the number of potential weaknesses inevitably increases. Studies consistently show that the cost of fixing a vulnerability rises dramatically the later it is discovered in the development cycle, making early detection an imperative. Robust vulnerability detection and accurate CWE classification are therefore not academic luxuries, but practical necessities for ensuring the resilience, safety, and reliability of the digital systems that modern society depends on.

1.2 Motivation and Scope

For decades, developers have relied on static analysis tools and handcrafted rules to find vulnerabilities. These tools remain useful, but they are inherently limited: rules must be constantly updated, they cannot easily capture complex semantic interactions, and they often generate overwhelming false positives. In parallel, the rise of large language models (LLMs) has shown striking ability to understand and generate code. Yet, when applied directly to security tasks, they remain brittle - hallucinating outputs, mislabeling edge cases, and struggling to generalize across diverse vulnerability classes.

This tension between the promise of LLMs and the rigor demanded by security drives the scope of this work. Our approach, **SecRAG**, leverages retrieval-augmented generation (RAG) to ground LLM predictions in curated, labeled exemplars. By combining a CWE-aware knowledge base with code-specific embeddings and a strict output schema, SecRAG is designed to achieve both *accuracy* and *interpretability*. Specifically, this research focuses on:

- Constructing a unified, CWE-enriched knowledge base from BigVul and PrimeVul, addressing label noise, duplication, and class imbalance.
- Embedding functions with UniXcoder and indexing them in Pinecone to retrieve contextually relevant examples during inference.
- Conditioning reasoning and non-reasoning LLMs through a carefully engineered prompt schema to ensure deterministic and parseable predictions.

1.3 Problem Statement

Despite remarkable progress in code analysis and machine learning, existing methods for vulnerability detection suffer from persistent limitations:

- **Shallow contextual understanding:** Models often fail to capture the deeper semantic and structural cues of source code, leading to inaccurate or incomplete vulnerability detection.
- **Weak generalization:** Many approaches perform well on benchmark datasets but falter on real-world vulnerabilities or emerging CWE categories.
- **Absence of knowledge-augmented reasoning:** Most systems analyze code in isolation, without leveraging curated external knowledge to guide predictions.
- **Low-quality training data:** Public vulnerability datasets frequently contain duplication, label errors, and extreme imbalance, undermining the reliability of downstream models.

SecRAG is designed to overcome these limitations by providing a retrieval-augmented, LLM-driven framework that grounds predictions in high-quality exemplars, enforces a deterministic schema, and evaluates performance with fairness and reproducibility in mind.

1.4 Research Challenges

Building such a system introduces several research challenges:

- **Constructing a CWE-focused knowledge base:** Merging and refining datasets like BigVul and PrimeVul requires eliminating inconsistencies, rebalancing negatives, and filtering sparse classes without losing coverage.
- **Ensuring stability in multi-class classification:** Many CWE categories are underrepresented, making accurate classification difficult without enrichment strategies.
- **Controlling LLM variability:** General-purpose LLMs often generate non-deterministic outputs, requiring prompt engineering and parsing mechanisms that enforce strict compliance.
- **Balancing performance and efficiency:** While reasoning models such as GPT-4.1 provide strong performance, they are computationally demanding.

Lightweight models like GPT-4o mini are efficient but may sacrifice accuracy. A fair evaluation must account for both.

- **Cross-language applicability:** Although this work focuses on C/C++ functions, extending the methodology to other languages presents further challenges and opportunities.

1.5 Contributions

The primary contributions of this thesis are:

1. **SecRAG: A Retrieval-Augmented, LLM-Driven Framework**
   A two-stage methodology that integrates dataset refinement with a retrieval-augmented pipeline, embedding code with UniXcoder, indexing in Pinecone, and guiding LLMs through structured prompts for vulnerability detection and CWE classification.

2. **Dataset Refinement and CWE Enrichment**
   A unified corpus created by merging BigVul and PrimeVul, correcting noisy annotations, deduplicating overlapping functions, rebalancing negatives, and filtering by frequency to retain 12 well-supported CWE classes.

3. **Diagnostic Fine-Tuning of Code-Specific PLMs**
   Comparative fine-tuning of CodeBERT, GraphCodeBERT, and UniXcoder on binary vulnerability detection and multi-class CWE classification tasks, validating dataset quality and establishing UniXcoder as the most suitable embedding model.

4. **Prompt Design for Deterministic Outputs**
   A structured, role-specific prompt schema that constrains model responses to a strict three-line format, ensuring determinism and interpretability across both reasoning (GPT-4.1, LLaMA-3.3) and non-reasoning (GPT-4o mini, GPT-3.5-turbo) LLMs.

5. **Unified Evaluation Protocol**
   A rigorous evaluation strategy using a shared test set, single-pass inference, and task-appropriate metrics (accuracy, precision, recall, F1-score), ensuring fair, reproducible, and interpretable comparisons.

1.6 Organization

The thesis unfolds as follows:

- **Chapter 2: Related Works** - Surveys existing vulnerability detection research and LLM applications.
- **Chapter 3: Proposed Methodology** - Details the framework's architecture, including dataset curation, retrieval and LLM integration.
- **Chapter 4: Results and Discussions** - Presents evaluation results across refined datasets, different PLMs and LLMs.
- **Chapter 5: Conclusion** - Recaps findings, limitations, and future directions.

Chapter 2

Related Works

The advent of Large Language Models (LLMs) and Retrieval-Augmented Generation (RAG) has been transforming software vulnerability detection. Earlier detection techniques, being predominantly static and dynamic analysis-based, provided good groundwork but suffered from scalability, semantic, and high false-positive issues. As deep learning and then transformer-based language models came into prominence, new possibilities were opened up for examining code at scale and identifying vulnerabilities with enhanced semantic precision. LLMs and RAG-based methods more recently ushered in another paradigm shift and enabled context-aware reasoning, knowledge lookup, along with few-shot or zero-shot detection capability. This section reviews prior works across five domains: background on vulnerabilities and detection methods, dataset creation and enhancement, LLM-based detection, RAG-based detection, and comparative studies.

2.1 Background on Software Vulnerabilities and Detection

2.1.1 Vulnerabilities and Standards

A software vulnerability is an inconsistency, flaw, or defect within a software system which, if exploited, can breach the system's security features, like confidentiality, integrity, or availability. Examples of vulnerabilities are a buffer overflow, an injection flaw, a race condition, and even some use-after-free errors. In order to aid the research community, academia, and the industry in cross-discipline identification and classification of vulnerabilities, they use well established standards and metrics. The Com-

mon Vulnerabilities and Exposures (CVE) system provides a unique and global reference identifier for each reported vulnerability within a security domain approach and compensates CVE weaknesses. Each CVE identifier is mapped to the Common Weakness Enumeration (CWE) standard which provides a structured and tiered classification of weaknesses, for instance, structural class or block: CWE-120: Buffer Copy without Checking Input Size. CVE and CWE, in conjunction, construct a primary methodology for tracking weaknesses and creating the needed labelled datasets for machine learning and LLM based detection systems

2.1.2 Traditional Detection Approaches

Before the rise of machine learning, vulnerability detection was dominated by static and dynamic program analysis techniques. Arguably the more advanced methods of static analysis – namely, symbolic execution, taint analysis, and AST inspection – all sought to make determinations about program behavior without actually running the code. For its part, dynamic analysis sought to determine a program's behavior under test inputs with the use of fuzzing and program runtime monitoring. While each of these approaches could identify certain qualitative categories of a program's vulnerabilities, the approaches suffered greatly from high false positive rates, a lack of scalability to more complex, industrial grade projects, and an inability to capture nuanced semantic relationships hidden deep within the code. The difficulty of balancing precision and recall in these techniques left open a significant gap that later data-driven methods sought to fill.

2.1.3 Code Vulnerability Detection Datasets and Methods

The introduction of machine learning and deep learning into software vulnerability detection led to the creation of several benchmark datasets and models. Notable datasets include Devign[18], BigVul[4], PrimeVul[2], Reveal[16], and SVEN[9], each providing labeled vulnerable and non-vulnerable code snippets for training and evaluation. Early deep learning-based systems such as VulDeePecker[12] analyzed semantic slices of code, also called "code gadgets," to identify vulnerabilities. Reveal[16] made use of graph neural networks (GNNs) to represent control-flow and data-flow information. It performed better in detection than static and dynamic approaches but was marred by two pervasive issues: label noise (e.g., erroneous commits labeled as vulnerability fixes) and dataset imbalance (with far fewer vulnerable samples than non-vulnerable samples).

2.1.4 CWE Identification Efforts

Another critical research direction has been the identification of vulnerabilities at the CWE level, rather than simple binary classification. Models based on transformers, such as CodeBERT[5] and GraphCodeBERT[5], exhibited advancements regarding better understanding the semantics of code and better "vulnerabilities detection" tasks. They, however, still focused on the binary "vulnerable" / "non-vulnerable" tasks and even as such, were limited in generating actionable insights at the CWE refinement levelMislabeling issues in benchmark datasets such as PrimeVul[2] further constrained the effectiveness of CWE identification, highlighting the need for cleaner, better-annotated datasets and more robust detection frameworks.

2.2 Dataset Creation and Enhancement with LLMs

2.2.1 CleanVul: Refining Vulnerability Datasets

A major bottleneck in training effective vulnerability detection models lies in the scarcity and noisiness of labeled datasets. To address this, CleanVul[10] was proposed, which combined LLM-based heuristics with additional filtering to refine vulnerability datasets. The system, called VulSifter[10], analyzed function-level code changes and commit messages, assigning a confidence score to estimate whether a commit represented a genuine vulnerability fix. It further incorporated heuristics drawn from software testing frameworks to filter out irrelevant modifications such as test-related commits. This method significantly improved dataset quality, raising the percentage of correctly labeled vulnerable functions from 28.7% to 90.6%. A model trained on CleanVul[10] achieved 58.09% accuracy on PrimeVul[2], outperforming PrimeVul-trained models (56.61%), though still trailing SVEN[9]. CleanVul[10] demonstrated that dataset refinement, guided by LLMs and heuristics, directly improves downstream detection accuracy.

2.2.2 VulScribeR: LLM-driven Vulnerability Augmentation

In parallel, another line of work focused on augmenting vulnerability datasets to address their limited size. The VulScribeR[1] framework proposed an LLM-driven augmentation pipeline incorporating three strategies: mutation, injection, and extension. Mutation introduced semantics-preserving edits such as renaming variables or altering control structures. Injection involved embedding vulnerable segments into otherwise benign code, a process strengthened through RAG-based retrieval of similar

vulnerabilities for contextually accurate insertion. Extension expanded vulnerable samples by integrating elements of clean code to diversify contexts.

The framework demonstrated remarkable effectiveness when evaluated on datasets such as Devign[18], BigVul[4], and Reveal[16]. Notably, the injection strategy consistently outperformed baselines such as VulGen[14] and ROS[14], achieving 30.8% F1-score improvement over non-augmented baselines. However, the framework was not without limitations, as LLMs occasionally hallucinated unrealistic vulnerabilities, potentially introducing new noise. Moreover, dataset imbalance remained a persistent concern.

2.3 LLM-based Vulnerability Detection

2.3.1 LProtector: RAG and Chain-of-Thought Integration

The LProtector[15] system represented one of the most comprehensive integrations of LLMs with retrieval-based reasoning for vulnerability detection. LProtector used GPT-4o combined with RAG and Chain-of-Thought (CoT) prompting to analyze C/C++ codebases from the BigVul[4] dataset. The system maintained a knowledge base of 500 vulnerable samples stored in a vector database , enriched with metadata such as CWE IDs and vulnerability descriptions. Incoming code samples were embedded using OpenAI embeddings and matched against this database via cosine similarity. Retrieved entries were incorporated into prompts alongside CoT reasoning instructions, enabling GPT-4o to classify functions as vulnerable or non-vulnerable.

The results highlighted the importance of retrieval in this hybrid architecture. LProtector outperformed established baselines such as VulDeePecker[12] and Reveal[16], achieving higher F1-scores. Interestingly, removing the CoT component resulted in only a minor performance drop, while removing RAG caused accuracy to collapse. This finding emphasized that retrieval was the most critical factor in improving model robustness and precision.

2.3.2 Evaluations of GPT-3.5 and GPT-4

Other studies have systematically compared GPT-3.5 and GPT-4 for vulnerability detection tasks [17]. A variety of prompt-engineering strategies were employed, including role-based prompting, injection of CWE knowledge, and retrieval of semantically similar code samples. Results revealed that incorporating CWE knowledge im-

proved GPT-3.5 accuracy by 18.2%, while retrieval-based augmentation added a further 19.6%. GPT-4 significantly outperformed CodeBERT[5], achieving a 34.8% improvement in accuracy, though at considerably higher computational costs.

A notable trade-off was observed: GPT-3.5 demonstrated higher precision, reducing false positives, but suffered from lower recall, missing actual vulnerabilities. On the other hand, CodeBERT[5] was able to recall much more, but at the cost of precision. This suggests that recall and precision need to be managed, depending on whether the system is more sensitive to false positives or the risk of weak points escaping detection.

2.3.3 Limitations of LLM-based Detection

Despite promising results, LLM-based methods remain constrained by several limitations. The computational cost of large models like GPT-4 poses a practical barrier to large-scale deployment. Risks of data leakage are also significant, as some test datasets may overlap with the pretraining corpora of commercial LLMs, leading to inflated performance estimates. Furthermore, many evaluations rely on artificially balanced datasets, which do not reflect the imbalanced nature of real-world software projects, thereby skewing results. LLMs also remain sensitive to noise in CWE annotations, sometimes overpredicting vulnerabilities even when no security issues exist.

2.4 RAG-based Vulnerability Detection

2.4.1 Vul-RAG: Knowledge-Level Retrieval

A more advanced approach is presented by Vul-RAG [3], which extended RAG-based detection beyond simple retrieval to incorporate knowledge-level reasoning. Vul-RAG built a vulnerability knowledge base by extracting semantic and contextual information from CVE entries using LLMs such as GPT-3.5. This knowledge included functional semantics, root causes of vulnerabilities, and associated fixes. For new code snippets, Vul-RAG employed BM25 and Reciprocal Rank Fusion (RRF) retrieval methods to fetch the most relevant knowledge entries. GPT-4 was then used to iteratively reason over retrieved knowledge until a confident decision was reached.

When evaluated on the PairVul[3] benchmark and datasets such as [4], Devign[18], and Reveal[16], Vul-RAG achieved a 12.96% increase in accuracy over the best baseline. A user study confirmed its practical effectiveness, with human detection accuracy rising from 60% to 77% when supported by Vul-RAG explanations. Despite these

advances, Vul-RAG's reliance on the quality of its knowledge base remained a limitation, as irrelevant or poorly structured entries sometimes led to false positives or negatives.

2.4.2 Role of RAG in Other Works

The utility of RAG has also been observed in other studies. For example, in VulScribeR[1], RAG was used to retrieve vulnerable code segments that guided the injection and extension augmentation strategies, thereby enhancing realism and diversity. Similarly, in LProtector[15], RAG proved to be the most critical component of the detection pipeline, surpassing even Chain-of-Thought prompting in its contribution to accuracy. These results clearly show that the retrieval mechanisms integrated into LLMs are essential in enhancing context and reducing hallucination.

2.5 Comparative Studies

One study, "Outside the Comfort Zone" [8] investigated how specialized vulnerability-detection LLMs perform relative to general-purpose LLMs such as GPT-4. Six specialized models (e.g., VulBERTa, fine-tuned CodeBERT, and fine-tuned CodeLlama-7B) were compared with six general-purpose LLMs (e.g., GPT-4, Mixtral, and Mistral) across multiple datasets, including Devign, PrimeVul, and a newly curated benchmark.

The findings highlighted an important trade-off. Specialized fine-tuned models such as CodeLlama-7B achieved exceptional performance on curated datasets but performed poorly when generalized to diverse datasets. Conversely, general-purpose models like GPT-4 provided consistent performance across datasets, although they rarely matched the peak performance of specialized models. Furthermore, the study revealed significant mislabeling issues in benchmark datasets, confirmed through independent validation with Semgrep, which contributed to false positives and negatives. This underscores the fact that dataset quality continues to limit the reliability of vulnerability detection models, regardless of their architecture.

Chapter 3

Proposed Methodology

This chapter presents a comprehensive two-stage methodology for vulnerable code detection. The approach is designed to address limitations in existing datasets and detection models by :

1. Constructing a high-quality, CWE-focused knowledge base through dataset merging, curation, and diagnostic fine-tuning of code-specific pre-trained language models (PLMs)
2. Operationalizing a retrieval-augmented detection (RAG) pipeline that embeds code with UniXcoder, retrieves semantically similar neighbors from a vector database (Pinecone), and queries large language models (LLMs) using a carefully designed structured prompt.

The pipeline ultimately produces both a binary decision on whether a function is vulnerable and a corresponding CWE class if applicable. The methodology emphasizes clarity, reproducibility, and task-appropriate evaluation.

3.1 Overview of the Methodology

The proposed methodology is organized into two major stages, illustrated conceptually in Figure 3.1. Each stage plays a distinct but complementary role in achieving accurate and explainable vulnerability detection.

3.1.1 Stage A - Knowledge Base Construction and Diagnostic Fine-Tuning

The first stage focuses on building a reliable and representative knowledge base for downstream retrieval. To accomplish this, two large-scale public vulnerability corpora, **BigVul** and **PrimeVul**, are integrated. During integration, several corrective operations are applied:

i. Missing or inconsistent fields are repaired

ii. Redundant functions across datasets are deduplicated

iii. The imbalance between vulnerable and non-vulnerable samples is partially mitigated by restricting the number of negative examples per commit.

Following this cleaning process, the dataset is refined into a **CWE-enriched subset**. This refinement is motivated by the need for stable multi-class classification; thus, only CWE identifiers with a minimum of 300 instances are retained. The resulting dataset includes 12 CWE categories, of which 8 belong to the CWE Top 25 list of most dangerous software weaknesses.

To validate the quality of the enriched dataset and to identify the most effective embedding model, three code-specific PLMs - **CodeBERT**, **GraphCodeBERT**, and **UniXcoder** - are fine-tuned. Fine-tuning is performed on two tasks:

1. Binary vulnerability detection

2. Multi-class CWE classification

Performance comparisons across these three PLMs reveal that UniXcoder provides the most robust semantic representations for code, leading to its selection as the embedding model for the subsequent retrieval-augmented pipeline.

3.1.2 Stage B - Retrieval-Augmented Detection (RAG)

The second stage leverages the curated knowledge base to enhance detection performance through retrieval-augmented generation. Each function in the CWE-enriched dataset is transformed into a 768-dimensional vector representation using the **microsoft/unixcoder-base-nine** model. These embeddings, along with associated metadata (function code, vulnerability label, CWE ID, and CWE Name), are indexed in a **Pinecone Vector Database** configured to use cosine similarity as its distance metric.

During inference, an input function under analysis is first embedded and then matched

against the knowledge base. The **top-3 nearest neighbor functions** are retrieved based on cosine similarity. These retrieved examples serve as context for the classification step. A carefully designed **structured prompt** is then generated, combining the input function and the retrieved references. Each reference includes its code, vulnerability label, and CWE information, thereby grounding the analysis in concrete, labeled examples. Both **reasoning-oriented models** (GPT-4.1, LLaMA-3.3) and **non-reasoning models** (GPT-4o mini, GPT-3.5-turbo) are evaluated under this setup. All models are accessed via API, ensuring reproducibility and fairness across experiments.

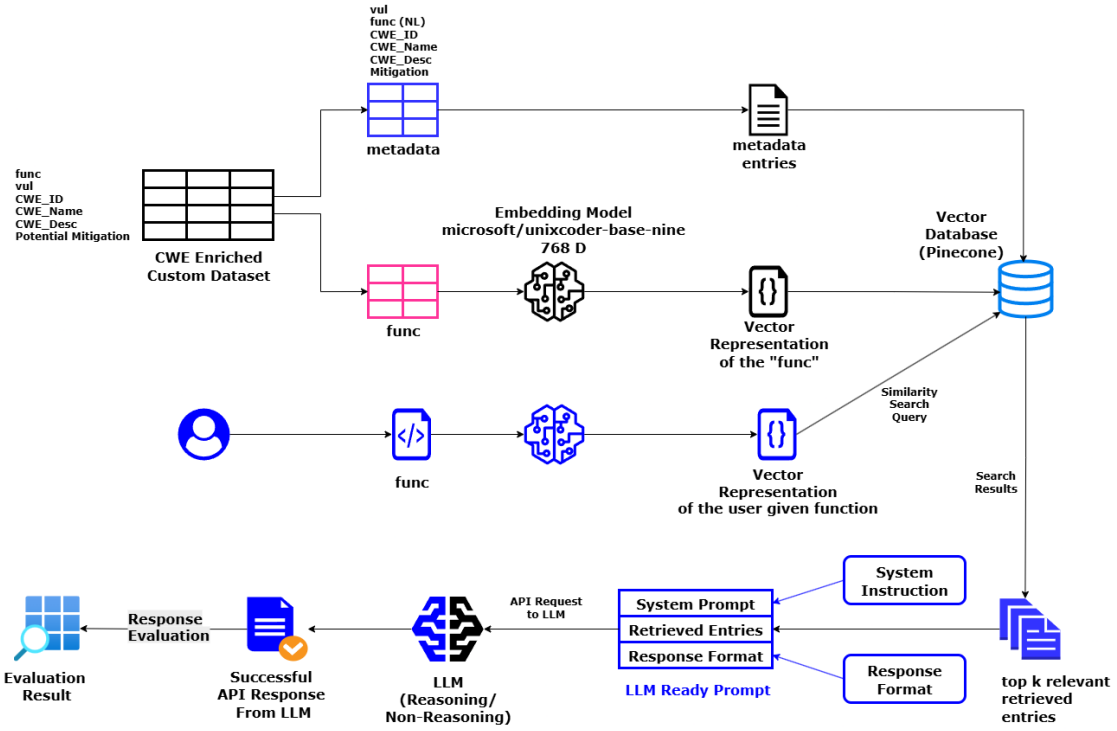

Figure 3.1: High-level architecture of the proposed two-stage methodology for vulnerable code detection. Stage A constructs a CWE-enriched knowledge base, while Stage B operationalizes the retrieval-augmented detection pipeline.

3.2 Stage A - Knowledge Base Construction

The first stage of the methodology focuses on constructing a high-quality, CWE-focused knowledge base that can serve as a reliable foundation for the retrieval-augmented pipeline. This stage consists of four main components:

i. Analyzing the source datasets and their limitations.

ii. Merging and repairing inconsistencies across datasets.

iii. Refining the dataset into a CWE-enriched subset.

iv. Conducting diagnostic fine-tuning of code-specific PLMs to select the most suitable embedding model

3.2.1 Source Datasets and Observed Limitations

Two large-scale, publicly available datasets were used as the foundation: **BigVul** [4] and **PrimeVul** [2]. Both contain function-level source code, commit metadata, and vulnerability annotations (including CWE and CVE identifiers where available). These datasets have been widely used in prior vulnerability detection research, but they exhibit several limitations that hinder direct application:

1. **Missing or incomplete fields:** A significant number of entries lack project identifiers, commit hashes, CWE IDs, or CVE IDs.
2. **Commit-level labeling noise:** CWE and CVE labels are assigned at the commit level, which propagates identifiers even to non-vulnerable functions, introducing spurious associations.
3. **Class imbalance:** Many commits contain disproportionately more non-vulnerable functions than vulnerable ones, creating severe skew in the label distribution.
4. **Sparse CWE coverage:** While both datasets reference many CWE categories, only a small subset has sufficient representation to support robust classification.

These issues necessitate a principled merge-and-repair pipeline before the datasets can be used to construct a reliable knowledge base.

Table 3.1: BigVul Split-Level Commit–Function Statistics

Split	Commits	Funcs	Avg func/Commit	Max func/Commit	Min func/Commit
Train	4,005	150,908	37.68	1,221	1
Test	3,215	33,050	10.28	272	1
Validation	3,256	33,049	10.15	265	1

Note: “Commits” here refer to unique commits per split. The global total (4,019) in Table 3.2 is not equal to the sum of split-level commits because of dataset partitioning.

Table 3.2: BigVul Dataset Statistics (Corpus-Level Overview)

Metric	Value
Train functions	150,908
Test functions	33,049
Validation functions	33,050
Total functions	**217,707**
Unique projects	309
Unique commits (overall)	4,019
Vulnerable functions	10,895
Non-vulnerable functions	206,112
Unique CWE IDs	90
Unique CVE IDs	3,512

Features available: `commit_id`, `commit_msg`, `func`, `project`, `vul`, `CWE ID`, `CVE ID`.

Table 3.3: PrimeVul Split-Level Commit–Function Statistics

Split	Commits	Funcs	Avg/Commit	Max/Commit	Min/Commit
Train	5,363	175,797	32.78	2,227	1
Test	686	24,788	36.13	869	1
Validation	694	23,948	34.51	851	1

Split-level figures reflect the provided statistics for each partition. Totals in Table 3.4 use the dataset's summary counts.

Table 3.4: PrimeVul Dataset Statistics (Corpus-Level Overview)

Metric	Value
Train functions	175,797
Test functions	24,788
Validation functions	23,948
Total functions	**224,533**
Unique projects	754
Unique commits (overall)	6,716
Vulnerable functions	6,004
Non-vulnerable functions	218,529
Unique CWE IDs	130
Unique CVE IDs	5,370

Features available: `commit_id`, `commit_msg`, `func`, `project`, `vul`, `CWE`, `CVE`, `CVE_Desc`.

3.2.2 Merge, Repair, and Deduplication

The two datasets are merged using shared fields, including `CWE ID`, `CVE ID`, `project`, `commit ID`, `function code`, and `vulnerability label`. Several corrective operations are then applied:

- **Removal of incomplete entries:** Functions missing any critical metadata fields are excluded to maintain dataset integrity.
- **Deduplication:** Identical function bodies across the two datasets (arising from overlapping scraping methods) are removed by comparing the `func` field.
- **Correction of CWE/CVE assignments:** For functions labeled as non-vulnerable ($vul = 0$), any associated CWE or CVE identifiers are replaced with a placeholder value `NOT_APPLICABLE`.
- **Rebalancing negatives:** To reduce class imbalance, the number of non vulnerable functions per commit is capped at 2. This conservative strategy avoids the semantic distortions that can arise with oversampling or synthetic example generation, which may alter code semantics or the presence of vulnerabilities. Techniques such as SMOTE or code mutation are particularly problematic in this context due to the structural and semantic sensitivity of source code - small changes can unintentionally introduce or remove vulnerabilities. In domains like vulnerability detection, data augmentation can introduce bias or degrade

model generalization, a challenge similarly observed in related software engineering tasks [13]

These operations collectively reduce noise, eliminate redundancies, and ensure that metadata more accurately reflects function-level vulnerability status.

3.2.3 CWE-Enrichment by Frequency Thresholding

Although the merged dataset spans a diverse range of CWE categories, the data distribution is highly imbalanced. Certain CWEs - such as CWE-20 - are heavily represented, while others are scarcely present, resulting in a long-tailed class distribution. This skew presents serious challenges for supervised learning: infrequent classes offer insufficient statistical support for stable representation learning, often leading to overfitting, poor generalization, and biased evaluation metrics. Recent work by Li et al. (2025) highlights this challenge, showing that models trained on vulnerability datasets struggle to generalize to the CWE Top 25 weaknesses due to the underrepresentation of many classes in training corpora [11].

To mitigate these challenges, we apply a frequency threshold: we keep only those CWE classes that have at least 300 samples in the merged corpus, creating the **CWE-enriched dataset**. This ensures that every class retained has sufficient representation to support stable training. While some rare classes are excluded, the refinement yields 12 CWE categories, 8 of which are included in the MITRE CWE Top 25 list. This trade-off improves statistical reliability without discarding the most critical real-world weaknesses, thereby supporting robust and fair evaluation.

Table 3.5: CWE classes retained in the CWE-enriched dataset.

CWE ID	Description
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer
CWE-20	Improper Input Validation
CWE-125	Out-of-bounds Read
CWE-399	Resource Management Errors
CWE-200	Information Exposure
CWE-787	Out-of-bounds Write
CWE-264	Permissions, Privileges, and Access Controls
CWE-416	Use After Free
CWE-476	NULL Pointer Dereference
CWE-190	Integer Overflow or Wraparound
CWE-189	Numeric Errors
CWE-362	Concurrent Execution with Improper Synchronization

3.2.4 Diagnostic Fine-Tuning of Code-Specific PLMs

To validate the quality of the CWE-enriched dataset and to determine the most suitable embedding model for the retrieval stage, we conduct diagnostic fine-tuning experiments on three representative pre-trained language models (PLMs) specialized for source code:

- **CodeBERT** [5]: A bimodal model based on the RoBERTa transformer, pre-trained on natural language–programming language (NL–PL) pairs. It captures semantic correspondences between code and text through masked language modeling and replaced token detection.
- **GraphCodeBERT** [7]: An extension of CodeBERT that incorporates data-flow graphs during pre-training, enabling improved modeling of program structure and variable dependencies.
- **UniXcoder** [6]: A unified encoder–decoder transformer pre-trained on large-scale programming corpora with multi-task objectives, supporting code completion, search, and summarization. Its design allows more comprehensive semantic understanding of source code.

These models are fine-tuned on two complementary tasks:

- **Binary vulnerability detection:** $y \in \{0, 1\}$, where $y = 1$ denotes a vulnerable

function and $y = 0$ a non-vulnerable one.

- **Multi-class CWE classification:** $c \in \{1, \dots, C\}$, where $C = 12$ corresponds to the CWE-enriched label space.

Preprocessing and modeling. Each function is tokenized using the model-specific tokenizer with a maximum sequence length of 512 tokens. Functions exceeding this length are truncated, while shorter ones are padded as required. The pooled representation of each function (the [CLS] token) is passed through task-specific classification heads:

$$\text{Vulnerability head: } \mathbb{R}^{768} \to \mathbb{R}^2, \qquad \text{CWE head: } \mathbb{R}^{768} \to \mathbb{R}^C.$$

Training objective. The fine-tuning objective combines binary classification and multi-class classification losses. Let $z^{bin} \in \mathbb{R}^2$ be the logits for binary vulnerability detection and $z^{cwe} \in \mathbb{R}^C$ the logits for CWE classification. Then:

$$\mathcal{L}_{bin} = -\Big[\, y \log \sigma(z_1) + (1 - y) \log(1 - \sigma(z_1)) \,\Big],$$

$$\mathcal{L}_{cwe} = -\sum_{c=1}^{C} \mathbf{1}[c = y] \log p_c, \quad p_c = \frac{e^{z_c}}{\sum_j e^{z_j}},$$

$$\mathcal{L} = \mathcal{L}_{bin} + \mathcal{L}_{cwe}.$$

This joint optimization encourages the model to simultaneously learn binary vulnerability discrimination and multi-class CWE representations. By combining the two objectives, the model is encouraged to capture both general vulnerability signals and fine-grained CWE-specific semantics.

Hyperparameters and environment. Fine-tuning is carried out under the following configuration:

- **Epochs:** 3, which balances convergence and overfitting risk.
- **Batch size:** 8, constrained by GPU memory.
- **Learning rate:** 2×10^{-5}, a standard setting for transformer fine-tuning that provides stable convergence.
- **Optimizer:** AdamW with weight decay 0.01, which helps regularize the model and mitigate overfitting.

- **Hardware:** Training is conducted on two NVIDIA T4 GPUs to accelerate experiments and enable parallelization.

The fine-tuning experiments revealed that **UniXcoder** achieved superior performance compared to CodeBERT and GraphCodeBERT, particularly in CWE classification accuracy and vulnerability detection F1-score. Its unified encoder–decoder architecture demonstrated a stronger capacity to capture program semantics and vulnerability patterns. Based on these results, UniXcoder was selected as the **embedding model** for Stage B of the retrieval-augmented pipeline.

3.3 Stage B - Retrieval-Augmented Vulnerability Detection and CWE Classification

3.3.1 Vectorization, Indexing (Pinecone) and Retrieval of Relevant Entries

Once the CWE-enriched dataset was finalized, each function f was transformed into a dense semantic vector representation using **unixcoder-base-nine**. Formally:

$$\phi(f) \in \mathbb{R}^{768}.$$

These embeddings, along with their associated metadata, were persisted in a **Pinecone** vector database. Pinecone provides a managed, high-performance similarity search service optimized for large-scale machine learning workloads. The index was configured with dimension $d = 768$ and cosine similarity as the distance metric. Each entry stored both the vector representation and the following metadata fields:

$$\{\texttt{func}, \texttt{vul}, \texttt{CWE ID}, \texttt{CWE Name}, \texttt{commit\_id}, \texttt{project}\}.$$

Cosine similarity between two vectors $u, v \in \mathbb{R}^{768}$ is defined as:

$$s(u, v) = \frac{u^\top v}{\|u\|_2 \, \|v\|_2}.$$

At inference time, an input function f^* is embedded as $\phi(f^*)$ and queried against the

index. The system retrieves the **top-3 nearest neighbors**:

$$\mathcal{N}_3(f^*) = \underset{g \in \mathcal{D}}{\arg\,\text{top3}}\, s\big(\phi(f^*), \phi(g)\big).$$

The choice of $k = 3$ was empirical: it balances *context sufficiency* with *prompt length constraints*. A single neighbor often provides too little evidence, while larger k values risk exceeding the context window and diluting relevance. Storing metadata alongside embeddings ensures interpretability and traceability, enabling retrieved examples to be directly incorporated into LLM prompts and allowing evaluation against ground truth.

3.3.2 LLM-Ready Prompt Construction

The second step involves formulating a structured prompt that conditions the LLM on both the input function and retrieved examples. The prompt was carefully designed with the following components:

- **Role instruction:** The model is explicitly instructed to act as a professional cybersecurity analyst with expertise in static code analysis and CWE classification. This anchors the model's reasoning within a domain-specific context.
- **Reference examples:** For each of the $k = 3$ retrieved neighbors, the function code and its known vulnerability label, CWE ID, and CWE Name are provided. These serve as evidence for analogical reasoning.
- **Input function:** The candidate function under analysis is appended, clearly separated from the references.
- **Strict output schema:** The model is required to respond in exactly three lines:

```
Vulnerability: <1|0>
CWE ID: <CWE-ID|NOT_APPLICABLE>
CWE Name: <CWE Name|NOT_APPLICABLE>
```

The instructions emphasize that no explanations or commentary are allowed, and the output must be wrapped in triple backticks. This strict schema ensures reliable structured parsing and prevents variability in model responses.

Sample Prompt Illustration

To provide clarity, Following shows a representative example of the complete prompt used in our experiments. The prompt begins with the system role and task instructions, includes three retrieved reference examples, and concludes with the input function and the required response schema.

```
You are a professional cybersecurity analyst with expertise in static code
    analysis and Common Weakness Enumeration (CWE) classification.

You will be provided with:
- One input function, whose vulnerability status you must assess.
- Several reference examples, each containing:
  - A code function
  - A known vulnerability label (1 for vulnerable, 0 for not)
  - CWE ID (if vulnerable)
  - CWE Name (if vulnerable)
---
### TASK
- Analyze the structure, logic, and behavior of the input function.
- Use deep comparison and reasoning based on the structure, intent, and
    usage patterns in the reference examples.
- Focus on the logical flow, operations performed, and overall behavior of
    the code to assess similarity and risk.

You must:
- Output whether the input function is vulnerable (1) or not (0)
- If vulnerable, identify the most appropriate CWE ID and CWE Name
- If not vulnerable, set CWE ID and CWE Name to "NOT_APPLICABLE"
---
### INSTRUCTIONS
- Carefully compare the input function with the reference examples
- Only use the structure and patterns from the examples for decision-making
- Strictly output your result in the exact 3-line format shown below
- No explanation or additional commentary
- Your answer must be wrapped in triple backticks for structured parsing
---
### Reference Example  Example - 1
Function:
<function code for Example - 1>
Vulnerability: <0|1>
CWE ID: <CWE-ID|NOT_APPLICABLE>
CWE Name: <CWE Name|NOT_APPLICABLE>
```

```
---
### Reference Example  Example - 2
Function:
<function code for Example - 2>
Vulnerability: <0|1>
CWE ID: <CWE-ID|NOT_APPLICABLE>
CWE Name: <CWE Name|NOT_APPLICABLE>
---
### Reference Example  Example - 3
Function:
<function code for Example - 3>
Vulnerability: <0|1>
CWE ID: <CWE-ID|NOT_APPLICABLE>
CWE Name: <CWE Name|NOT_APPLICABLE>
---
### Input Function
<function under analysis>
---
### Your Response
Vulnerability:
CWE ID:
CWE Name:
```

Used Large Language Models

To assess the generality of the proposed retrieval-augmented framework, we evaluate both reasoning-oriented and non-reasoning large language models. This design choice allows us to compare models that have advanced chain-of-thought and reasoning capabilities with lighter-weight models optimized for efficiency.

Reasoning models.

- **GPT-4.1** (OpenAI API): A state-of-the-art reasoning model known for its ability to perform multi-step inference, handle complex classification instructions, and maintain consistency across long prompts. GPT-4.1 is used as a benchmark for high-capacity reasoning under the structured prompt design.
- **LLaMA-3.3** (Cerebras API): A reasoning-augmented open-weight model family, accessed through Cerebras Inference API. This model provides complementary insights into reasoning performance beyond proprietary APIs and allows testing on diverse architectures.

Non-reasoning models.

- **GPT-4o mini** (OpenAI API): A lightweight, optimized model intended for fast, cost-efficient inference. It is less capable of step-by-step reasoning compared to GPT-4.1 but serves as a representative of smaller-scale LLMs that prioritize efficiency.
- **GPT-3.5-turbo** (OpenAI API): A widely deployed model that balances scalability and performance. It lacks explicit reasoning modules but has been extensively validated across industrial applications, making it a strong baseline for vulnerability classification tasks.

Inference protocol:

All models are accessed via their respective APIs. Decoding parameters such as temperature, top-p, and maximum tokens are kept at provider defaults to avoid introducing artificial bias. Determinism is enforced at the task level through the rigid three-line output schema described earlier, which ensures that responses remain structured and parseable regardless of model stochasticity. Every model is run once per input without retries, sampling, or ensembling, maintaining fairness and reproducibility across experiments.

3.3.3 LLM Interaction, Response Parsing, and Evaluation

Once the prompt is constructed, it is submitted to the chosen LLM via the corresponding provider API. The same held-out test set is used across all models to ensure fairness in comparison. Each input function is evaluated exactly once per model, with no retries or ensemble averaging, thereby ensuring consistent and reproducible results.

Response parsing

All model outputs are required to be enclosed in triple backticks and to follow the strict three-line schema introduced earlier. This design enables reliable automatic extraction of predictions. The first line indicates whether the function is vulnerable or not. The second and third lines provide the CWE ID and CWE Name if the function is predicted to be vulnerable; otherwise, both are set to `NOT_APPLICABLE`. This uniform format prevents ambiguity and ensures that results from different LLMs can be directly compared.

Evaluation protocol

Predictions generated by the models are compared against ground-truth labels in the test dataset. For binary vulnerability detection, we evaluate performance using standard classification metrics such as accuracy, precision, recall, and F1-score. These metrics together provide a balanced view of model effectiveness, capturing not only overall correctness but also the trade-off between detecting true vulnerabilities and avoiding false alarms. For CWE classification, which is a multi-class prediction task, accuracy is used as the primary measure. Since the CWE-enrichment step ensured that each retained class had adequate support, accuracy provides a fair and interpretable assessment of model performance.

3.4 Summary

This chapter presented a two-stage methodology for retrieval-augmented vulnerability detection and CWE classification. In **Stage A**, we constructed a high-integrity, CWE-focused knowledge base by merging the BigVul and PrimeVul datasets, repairing inconsistencies, removing duplicates, and applying a principled rebalancing strategy. Sparse CWE categories were filtered through frequency thresholding, and diagnostic fine-tuning of CodeBERT, GraphCodeBERT, and UniXcoder validated dataset quality while identifying UniXcoder as the most effective embedding model.

In **Stage B**, we operationalized a retrieval-augmented generation (RAG) pipeline. Functions were vectorized using UniXcoder and indexed in Pinecone, with the top-3 nearest neighbors retrieved to provide contextual exemplars. These neighbors were incorporated into a structured, role-specific prompt that guided both reasoning and non-reasoning LLMs in producing deterministic predictions of vulnerability status and CWE category. A strict output schema ensured parseability and reproducibility, while a unified evaluation protocol (shared test set, single-pass inference, and task-appropriate metrics) enabled fair comparison across models.

Together, these stages demonstrate how coupling semantic retrieval with constrained LLM generation can yield a robust, interpretable, and reproducible framework for automated vulnerability detection.

Chapter 4

Results and Discussion

This chapter presents and discusses the results of our experiments to evaluate the effectiveness of SecRAG, a framework for vulnerability detection in code. To assess the effectiveness of our models, we adopt four widely used metrics in classification tasks: *Precision*, *Recall*, *F1-score*, and *Accuracy*. Let TP, FP, TN, and FN denote the number of true positives, false positives, true negatives, and false negatives, respectively.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.1}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4.2}$$

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.3}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{4.4}$$

These metrics capture complementary aspects of model behavior. Accuracy provides an accurate measure of correctness while Precision emphasizes the ability to avoid false alarms, which is crucial in practical vulnerability detection where excessive false positives reduce developer trust. Recall reflects the capacity to capture actual vulnerabilities, a critical factor for security-sensitive applications. The F1-score balances these two concerns and is particularly valuable when the dataset is imbalanced across vulnerable and non-vulnerable samples.

With these evaluation criteria in place, we now turn to the central focus of this chap-

ter: answering our research questions. These questions guide the experimental analysis and help structure the discussion of how SecRAG and the evaluated models perform across different dimensions of the vulnerability detection task.

- **RQ1**: How effectively can code-specific pre-trained language models (PLMs) detect code vulnerabilities when fine-tuned on a custom, enriched vulnerability dataset?
- **RQ2**: Can Retrieval Augmented Detectio0n techniques improve the accuracy and robustness of code vulnerability detection compared to standard fine-tuned PLMs?
- **RQ3**: How does model's reasoning capacity influence the effectieveness of a RAG-powered vulnerability detection system?

Our findings demonstrate that while fine-tuned PLMs like UniXcoder achieve strong performance, particularly with high-quality, CWE-enriched data, the RAG-based approach can provide superior results in certain metrics, especially in CWE classification. Notably, the quality of the underlying dataset is a critical factor, significantly impacting model performance. Our curated dataset, which addresses the common issues of noise, duplication, and imbalance found in benchmarks like BigVul[4], enabled our models to achieve higher and more reliable F1 scores, validating our hypothesis. The results suggest that for complex, multi-class vulnerability classification, the combination of a robust LLM (e.g., GPT-4.1) and a structured, high-quality knowledge base is more effective than either fine-tuning or retrieval alone. This work highlights the potential of RAG for enhancing code vulnerability detection by providing models with accurate, context-rich information, thus moving beyond simple binary classification to more nuanced and interpretable vulnerability identification.

4.1 Dataset

For evaluating SecRAG, we relied on the validation split of our curated dataset as test set. The dataset consists of C and C++ functions annotated with CWE identifiers, descriptions, and binary vulnerability labels. Each entry contains a function body ("func"), its vulnerability status and a CWE class with mitigation where applicable.

The test dataset used in our experiments comprises of 2253 functions where function snippet lengths vary, ranging from very short (19 characters) to quite extensive (482,000 characters), drawn from a large corpus of 22,000+ samples and balanced across vulnerable and non-vulnerable samples to minimize bias.

The dataset spans twelve critical CWE categories, ensuring that the evaluation captures a diverse spectrum of software vulnerabilities. These include memory safety issues such as CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), CWE-125 (Out-of-Bounds Read), CWE-787 (Out-of-Bounds Write) and CWE-416 (Use-After-Free). Input validation and numeric errors are represented through CWE-20 (Improper Input Validation), CWE-190 (Integer Overflow), and CWE-189 (Numeric Errors). Broader security categories such as CWE-200 (Information Exposure), CWE-264 (Permissions, Privileges, and Access Controls), CWE-362 (Race Condition), CWE-399 (Resource Management Errors), and CWE-476 (NULL Pointer Dereference) further enrich the dataset. Collectively, these twelve classes provide balanced coverage of both low-level memory corruption vulnerabilities and higher-level design and logic flaws.

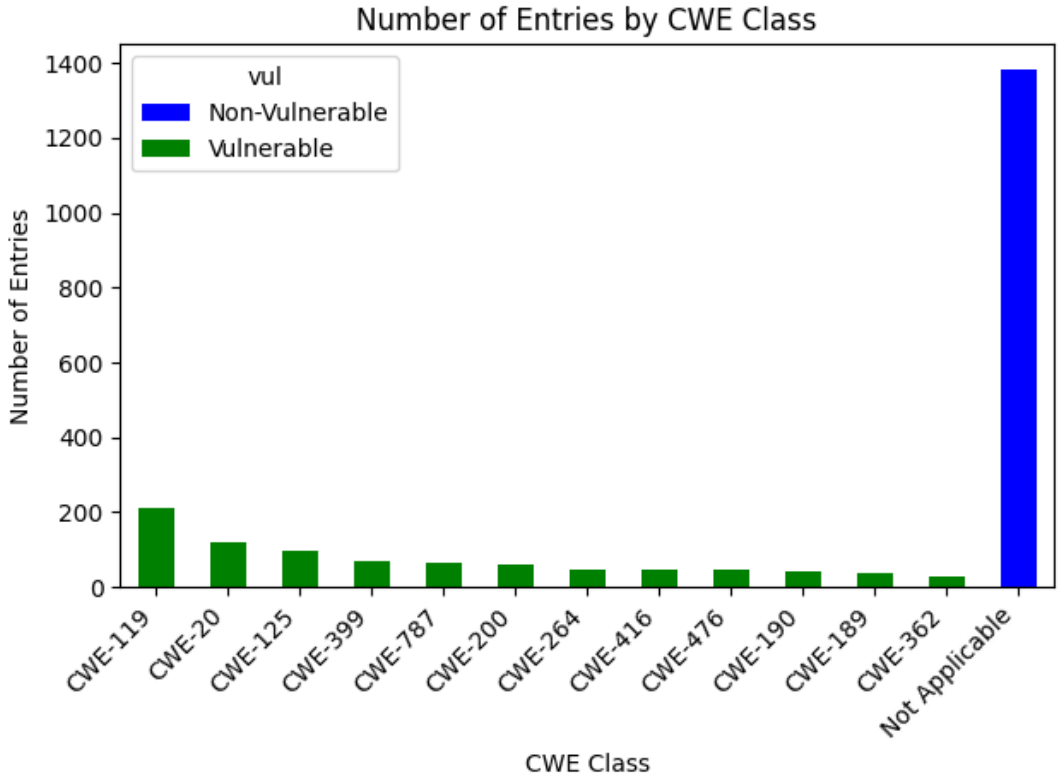

Figure 4.1: Distribution of functions in test set by CWE class and vulnerability status (vulnerable vs non-vulnerable).

As shown in Figure 4.1, the dataset exhibits a relatively balanced distribution across CWE classes, with certain classes such as CWE-119 and CWE-200 showing higher frequencies. The stacked bar chart highlights both vulnerable and non-vulnerable function counts, providing a clear visual understanding of dataset composition.

4.2 Experimental Setup

This section outlines the experimental configurations used to evaluate our approach across the three research questions. For RQ1, we fine-tuned code-specific pre-trained language models (CodeBERT, GraphCodeBERT, and UniXcoder) on a curated, CWE-enriched dataset, detailing preprocessing, tokenization, model architecture modifications, training hyperparameters and hardware. For RQ2 and RQ3, we describe the retrieval-augmented framework with large language models (GPT-4o mini, GPT-4.1, GPT-3.5 turbo, and Llama 3.3 70B), including embedding, retrieval, reranking, and prompt construction. The experimental setup ensures reproducibility, fair model comparison, and evaluation across both binary vulnerability detection and multi-class CWE classification.

4.2.1 Experimental Setup for RQ1: Fine-tuning of Code-Specific PLMs

This section describes the experimental configuration used to answer **RQ1**, which evaluates the ability of code-specific pre-trained language models (PLMs) to detect vulnerabilities when fine-tuned on a curated, CWE-enriched dataset. The description covers dataset preprocessing, model configurations, training hyperparameters, hardware, and loss formulation. Wherever appropriate, brief justifications for key choices are provided to aid reproducibility and to explain trade-offs.

Dataset preprocessing and tokenization

Each dataset entry was treated as an independent function (the *func* field). Prior to training, the following preprocessing steps were applied:

- **Tokenization:** Tokenization was performed with the model-specific auto tokenizers provided by the Hugging Face Transformers library. Each code snippet was truncated or padded to a fixed maximum length of **512 tokens**. The 512-token limit balances coverage of typical function lengths against memory constraints during fine-tuning; it is a common choice in transformer-based code modeling.
- **Label encoding:** Two label sets were prepared:
  1. **Binary vulnerability label** (0 = non-vulnerable, 1 = vulnerable).
  2. **Multi-class CWE label** (one discrete class per CWE; `num_cwe_classes` = 12 for our curated corpus).

Model loading and architectural modifications

We fine-tuned three code-focused PLMs as the backbone models for this experiment:

- `codebert-base`
- `graphcodebert-base`
- `unixcoder-base`

Each model was loaded from Hugging Face checkpoints and extended with lightweight task-specific heads:

- **Vulnerability head:** a linear classification layer mapping the model hidden dimension (768) to 2 logits ($768 \rightarrow 2$).
- **CWE classification head:** a linear classification layer mapping 768 to `num_cwe_classes` logits ($768 \rightarrow 12$).

Training Parameters and Hardware

The following training hyperparameters were used for all model fine-tuning runs reported in response to RQ1:

- **Epochs:** 3
- **Batch size:** 8
- **Learning rate:** 2×10^{-5}
- **Optimizer:** AdamW with weight decay $= 0.01$
- **Hardware:** 2x NVIDIA T4 GPUs

Hyperparameter rationale: The learning rate of $2{\times}10^{-5}$ and small batch size of 8 are commonly adopted defaults for fine-tuning transformer models on domain-specific tasks; they provide a stable optimization trajectory without requiring large updates that could destabilize pretrained weights. The small number of epochs (3) was chosen to reduce the risk of overfitting on the medium-sized curated dataset (22,529 examples), while still allowing the model sufficient passes to adapt to domain-specific patterns. AdamW with modest weight decay (0.01) was used to aid generalization by penalizing large weights.

The configuration described above provided a reproducible, resource-aware fine-tuning pipeline that facilitated a fair comparison among CodeBERT, GraphCodeBERT, and

UniXcoder for both binary vulnerability detection and CWE classification in answer to RQ1.

4.2.2 Experimental Setup for RQ2 and RQ3

All experiments to answer RQ2 and RQ3 were conducted using the RAG pipeline described in Chapter 3.The framework was evaluated with four large language models GPT-4o mini, GPT-4.1, GPT-3.5 turbo, and Llama 3.3 70B which are chosen to represent a spectrum of model sizes and capabilities. Each model was integrated into the same multi-stage retrieval-augmented workflow, which included UnixCoder[6] embeddings for code representation and Pinecone as the vector database (knowledge base), with cosine similarity used for approximate nearest-neighbor retrieval The outputs of these stages were then used to construct structured prompts, which the models analyzed to predict vulnerability status and CWE class. All models were tested in the same test set and no fine-tuning was performed directly on the evaluation data.

Performance was measured using standard classification metrics, including accuracy, precision, recall, and F1 score for binary vulnerability detection. In addition, we evaluated CWE classification accuracy to capture how well models distinguish among the twelve vulnerability categories present in the dataset. This dual focus on both binary and multi-class evaluation provides a holistic view of the framework's strengths and limitations.

4.3 Results

The results section evaluates SecRAG's effectiveness across three research questions. First, it examines how well code-specific pre-trained language models (PLMs) detect vulnerabilities when fine-tuned on a CWE-enriched dataset. Second, it assesses the impact of retrieval-augmented detection using large language models, comparing standard PLMs to models enhanced with context from a structured knowledge base. Finally, it explores how reasoning capacity of LLMs influence RAG-powered vulnerability detection. Across all experiments, both binary vulnerability detection and multi-class CWE classification are considered to provide a comprehensive view of model capabilities.

4.3.1 RQ1: How effectively can code-specific pre-trained language models (PLMs) detect code vulnerabilities when fine-tuned on a custom, enriched vulnerability dataset?

To address RQ1, we evaluated three representative code-specific pre-trained language models namely, CodeBERT, GraphCodeBERT, and UniXcoder on the test set of our curated dataset. Their performance was measured on two tasks: binary vulnerability detection (vulnerable vs. non-vulnerable) and multi-class CWE classification.

Table 4.1 summarizes the results across five key metrics: vulnerability accuracy, precision, recall, F1 score and CWE accuracy.

Table 4.1: Performance of code-specific PLMs on vulnerability detection and CWE classification

Metric	CodeBERT	GraphCodeBERT	UniXcoder
Vulnerability Accuracy	80.34%	79.45%	81.58%
Vulnerability Precision	80.20%	76.69%	79.97%
Vulnerability Recall	65.41%	67.47%	69.99%
Vulnerability F1	72.05%	71.79%	74.65%
CWE Accuracy	39.63%	46.05%	53.26%

The results for RQ1 highlight the effectiveness of code-specific pre-trained language models (PLMs) on our enriched dataset. Among the three models, UniXcoder consistently outperformed CodeBERT and GraphCodeBERT, particularly in recall, F1 score, and CWE classification accuracy, indicating its stronger ability to capture both general vulnerability patterns and fine-grained CWE semantics.

UniXcoder achieved a recall of 69.99%, higher than CodeBERT's 65.41%, showing it was better at identifying true vulnerable cases. Although CodeBERT had a slight edge in precision (80.20% vs. 79.97%), this came at the cost of lower recall, resulting in a weaker F1 score (72.05% vs. UniXcoder's 74.65%). GraphCodeBERT's performance fell between the two, suggesting limited gains from graph-aware representations.

The most notable gap appeared in CWE classification: UniXcoder achieved 53.26%, substantially higher than CodeBERT (39.63%) and GraphCodeBERT (46.05%). This underscores the advantage of UniXcoder's unified pretraining and the value of evaluating on a CWE-enriched dataset, which promotes more accurate vulnerability categorization.

When contrasted with findings reported by Ding et al.[2], the differences are instructive. That study showed significantly lower F1 scores for CodeBERT (62.88%) and UniXcoder (65.46%) when evaluated on BigVul[4], and even more drastic drops to

around 20% when tested on their PrimeVul dataset. By contrast, our results report substantially higher F1 scores 72.05% for CodeBERT and 74.65% for UniXcoder suggesting that the CWE-enriched dataset used in SecRAG supports more meaningful model learning.

One explanation for this discrepancy is the quality of the dataset. Ding et al.[2] argue that BigVul[4] suffers from poor label accuracy, duplication, and noise, all of which inflate reported accuracies while masking real-world difficulty. Our dataset explicitly addresses these issues by deduplication, removal of invalid CWE labels from non-vulnerable functions, and balanced sampling across CWE categories.

Interestingly, Ding et al.[2] reported much higher overall accuracy values (above 95%) compared to our results (around 80%). However, they caution that accuracy is a misleading metric in vulnerability detection due to class imbalance: a model predicting "not vulnerable" for most functions can still achieve high accuracy but fail in practice. Our dataset design intentionally balances vulnerable and non-vulnerable samples, which prevents such inflation. This explains why our accuracy figures are lower, but our F1 scores are higher and more reliable indicators of performance.

Taken together, these findings validate our hypothesis that high-quality, CWE-aware datasets can significantly enhance the performance of vulnerability detection models.

4.3.2 RQ2: Can Retrieval Augmented Detection techniques improve the accuracy and robustness of code vulnerability detection compared to standard fine-tuned PLMs?

To evaluate RQ2, we applied the SecRAG pipeline with four large language models: GPT-4o mini, GPT-4.1, GPT-3.5 turbo and Llama 3.3 70B using the test set of our dataset. Performance was measured on both binary vulnerability detection and multi-class CWE classification.

Table 4.2 reports the results across precision, recall, F1 score, overall accuracy, and CWE classification metrics.

Table 4.2: Performance of retrieval-augmented models on vulnerability detection and CWE classification

Model	Precision	Recall	F1	Accuracy	CWE Accuracy
GPT-4o mini	45.41%	79.95%	57.93%	54.99%	40.70%
GPT-3.5 turbo	48.30%	40.80%	44.24%	60.10%	52.56%
GPT-4.1	59.69%	57.77%	58.71%	68.51%	61.37%
Llama 3.3 70B	52.55%	56.70%	54.55%	63.38%	53.66%

The results indicate a clear difference across models. GPT-4.1 achieved the strongest overall performance, with an F1 score of 58.71% and a CWE accuracy of 61.37%, outperforming the other models on nearly all metrics. GPT-4o mini, while attaining the highest recall (79.%), suffered from low precision (45.41%), which reduced its overall balance and CWE classification ability. Llama 3.3 70B produced competitive results (F1 = 54.55%, CWE accuracy = 53.66%), while GPT-3.5 turbo lagged behind with the lowest F1 score (44.24%)

We contrast our results with findings reported by Sheng et al.[15]. They combines GPT-4o with Retrieval-Augmented Generation (RAG) and Chain of Thought (CoT) prompting, reporting an accuracy of 89.68%, precision of 30.52%, recall of 38.07%, and an F1 score of 33.49% on the BigVul[4] dataset. This outperformed earlier baselines like VulDeePecker[12] (19.15% F1) and Reveal[16] (22.87% F1).

In comparison, our experiments achieved substantially higher F1 scores: 58.71% for GPT-4.1, 57.93% for GPT-4o mini, 54.55% for Llama 3.3, and 44.24% for GPT-3.5 turbo. While Sheng et al.[15] reported higher raw accuracy, this figure is less reliable due to dataset imbalance, whereas our curated dataset was explicitly balanced across vulnerable and non-vulnerable samples. Precision and recall trade-offs were also more favorable in SecRAG, with GPT-4.1 balancing both (Precision 59.69%, Recall 57.77%) and GPT-4o mini achieving very high recall (79.95%).

The main difference lies in methodology and dataset. Experiments of Sheng et al.[15] relies on BigVul[4] and gains from CoT prompting, while SecRAG leverages a CWE-enriched, deduplicated dataset merged from BigVul[4] and PrimeVul[2], along with multi-stage retrieval. This cleaner and more representative benchmark enabled SecRAG to achieve significantly higher F1 scores and CWE classification accuracy, underscoring the role of knowledge base quality in retrieval-augmented vulnerability detection.

4.3.3 RQ3: How does model reasoning capacity influence the effectiveness of a RAG-powered vulnerability detection system?

To address RQ3, we investigated whether reasoning-oriented models outperform lighter, non-reasoning models when equipped with the same CWE-enriched knowledge base in SecRAG. Our evaluation compared GPT-4.1 and Llama 3.3 (reasoning-capable LLMs) against GPT-4o mini and GPT-3.5 turbo (non-reasoning LLMs) on the test set.

The results reveal a consistent advantage for reasoning models. GPT-4.1 achieved

the highest CWE accuracy at 61.37%, and Llama 3.3 followed at 53.66%. By contrast, GPT-3.5 turbo and GPT-4o mini attained 52.56% and 40.70%, respectively. While GPT-4o mini achieved the highest recall (79.95%), its limited reasoning capacity made it less effective at converting retrieved CWE-aware context into accurate predictions, resulting in lower CWE accuracy.

Figure 4.2 compares binary accuracy and CWE classification accuracy across models. The results show that reasoning-oriented models not only achieve higher CWE accuracy but also balance precision and recall more effectively. GPT-4.1 in particular demonstrates robustness across both binary and multi-class tasks, highlighting the synergistic effect of reasoning ability and structured retrieval.

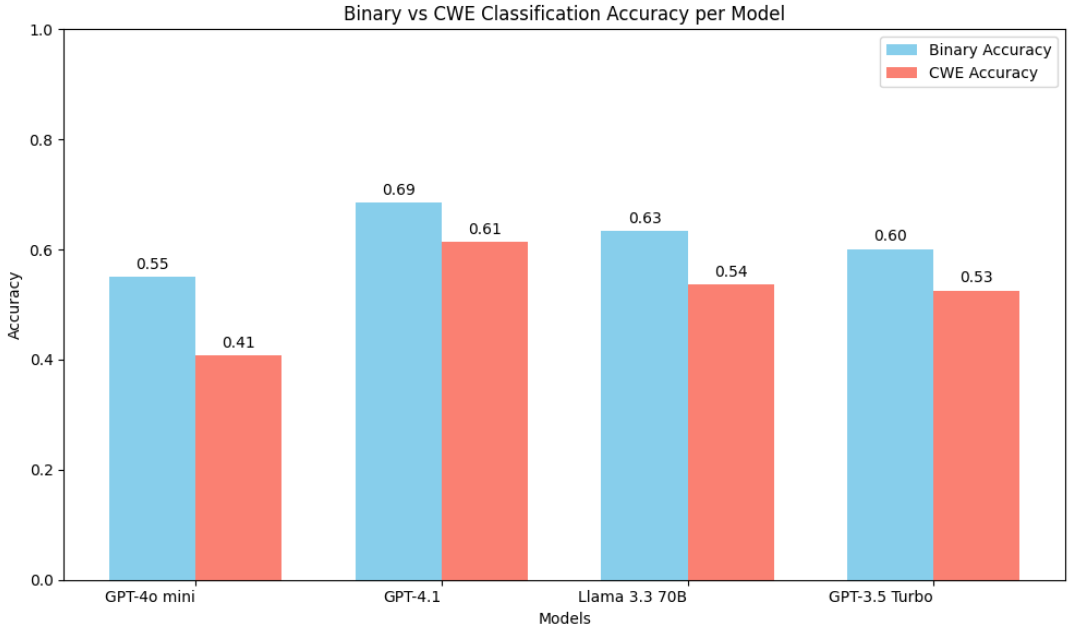

Figure 4.2: Comparison of binary vulnerability detection accuracy and CWE classification accuracy across different Large Language Models. Each model has two bars: the left bar represents binary classification accuracy, while the right bar shows multi-class CWE classification accuracy.

These findings confirm that while retrieval-augmented detection improves vulnerability detection broadly, the benefits are maximized in reasoning-capable models. Structured knowledge alone is not enough, models require the reasoning capacity to interpret retrieved CWE context and apply it effectively to diverse vulnerability classes. This reinforces the notion that high-quality datasets and advanced LLM reasoning work hand in hand in achieving robust vulnerability detection.

4.4 Challenges and Limitations

Despite the promising results achieved by SecRAG, several challenges and limitations were encountered throughout this research. First, creating a high-quality, balanced dataset with accurate CWE and CVE annotations for a wide range of vulnerabilities remains challenging. While our curated dataset addresses many issues in existing corpora, some vulnerability types are underrepresented, which may limit the generalization of the models.

Computational constraints posed additional limitations. Fine-tuning and inference with large models such as UniXcoder and Llama 3.3 require substantial GPU resources, which restricts experimentation with larger batch sizes, longer sequences, or extensive hyperparameter sweeps. Thirdly, Large language models can produce variable predictions and explanations across runs. Ensuring consistent results throughout model responses required careful prompt design and retrieval strategies, yet some output variability could remain unavoidable.

Finally, while the study focused on C and C++ functions, generalizing vulnerability detection across multiple programming languages with diverse syntax and semantics remains an open challenge. Models trained on one language may underperform on others without further fine-tuning or language-agnostic embeddings.

Taken together, these challenges highlight that while SecRAG demonstrates significant improvements over existing approaches, further work is needed to scale datasets, improve knowledge base coverage, handle long and complex code snippets, and enhance computational efficiency. Addressing these limitations offers a clear path for future research in automated vulnerability detection.

4.5 Implications and Significance of Findings

This work demonstrates that combining a high-quality, CWE-enriched knowledge base with a RAG pipeline yields measurable gains over vanilla fine-tuning of code specific pre trained models. Methodologically, it shows that retrieval-augmented architectures can improve both binary vulnerability detection and fine-grained CWE classification when the external knowledge is accurate and well-structured.

Empirically, the results highlight two priorities: (1) dataset quality (deduplication, correct labels, balanced classes) affects learning, and (2) larger LLMs better leverage retrieved context to reason about vulnerability types. Practically, SecRAG's outputs

(detection + CWE-aware explanations) can improve triage, prioritize fixes, and inform remediation guidance in development pipelines.

Finally, the findings point to clear next steps: expand and diversify the knowledge base, improve scaling/efficiency of the RAG stack, and validate explanation reliability with human experts to enable safe, production deployment.

4.6 Summary

This chapter evaluated the effectiveness of SecRAG across three research questions. First, fine-tuned code-specific pre-trained language models (PLMs) such as UniXcoder[6], CodeBERT[5], and GraphCodeBERT[7] were assessed on a CWE-enriched dataset. UniXcoder[6] achieved the strongest performance, particularly in recall, F1 score, and CWE classification, highlighting the value of high-quality, curated data for learning both binary and fine-grained vulnerability patterns.

Second, Retrieval-Augmented Generation (RAG) approaches using large language models (GPT-4.1, GPT-4o mini, GPT-3.5 turbo, and Llama 3.3) were compared to standard fine-tuned PLMs. RAG-based models demonstrated improved CWE classification and more balanced precision-recall trade-offs, with GPT-4.1 performing best, underscoring the effectiveness of combining retrieval with powerful LLM reasoning.

Third, the experiments emphasized the critical role of model reasoning capacity in retrieval-augmented detection. While all models benefited from the CWE-enriched knowledge base, reasoning-oriented LLMs such as GPT-4.1 and Llama 3.3 achieved substantially higher CWE classification accuracy, illustrating that retrieval alone is insufficient without the ability to effectively interpret and apply the retrieved context.

Overall, the results show that SecRAG's combination of high-quality datasets, retrieval-augmented reasoning, and capable LLMs enhances both vulnerability detection and interpretability, providing a strong foundation for future research and practical application in secure software development.

Chapter 5

Conclusion

This thesis set out to address the challenge of detecting vulnerabilities in source code and classifying them according to the Common Weakness Enumeration (CWE) taxonomy. The central objective was to develop a robust and interpretable framework - **SecRAG** - that combines dataset refinement with retrieval augmented, large language model (LLM)-driven inference. The guiding questions asked whether high-quality, CWE-enriched datasets could be built from existing corpora, whether retrieval augmented generation could reduce hallucinations and improve detection accuracy, and how reasoning and non-reasoning models would perform under structured prompts.

The research yielded several important findings. By merging BigVul and PrimeVul, repairing inconsistencies, and rebalancing negatives, a CWE-enriched dataset was created with 12 stable categories, eight of which belong to the CWE Top 25. Diagnostic fine-tuning validated dataset quality and identified UniXcoder as the most suitable embedding model. Building on this, the SecRAG pipeline embedded functions, retrieved top-3 nearest neighbors from Pinecone, and constructed deterministic prompts that conditioned both reasoning (GPT-4.1, LLaMA-3.3) and non-reasoning (GPT-4o mini, GPT-3.5-turbo) models. The strict schema enforced reproducibility, and results showed that reasoning models achieved stronger generalization and accuracy.

The contributions of this work include the development of SecRAG, a two-stage retrieval augmented and LLM-driven framework for vulnerability detection and CWE classification; the construction of a refined CWE-enriched dataset; empirical validation of UniXcoder as the optimal embedding model; a prompt schema that guarantees deterministic outputs; and a unified evaluation protocol ensuring fairness and interpretability. Together, these contributions advance automated vulnerability detection

by demonstrating how curated data and retrieval-augmented inference can be combined effectively.

Several limitations should be acknowledged. The experiments focused primarily on C/C++ functions, and further work is needed to assess cross-language generalization. The study was restricted to four LLMs due to resource constraints, and the retrieval step was limited to three neighbors. While the strict schema improved determinism, it may also limit opportunities for explainability.

Future research can extend SecRAG to multi-language corpora, explore hybrid retrieval and reranking strategies, and develop interpretability modules that provide actionable insights alongside predictions. Expanding the evaluation to include a broader range of LLMs and integrating the framework into real-world development pipelines would further validate its utility.

In conclusion, this thesis shows that vulnerability detection can be strengthened by combining dataset refinement, embedding-based retrieval, and structured LLM inference. SecRAG offers a reproducible and interpretable methodology for CWE classification and vulnerability detection, highlighting that progress in secure software engineering requires not only advanced models but also carefully designed pipelines that ground machine intelligence in curated knowledge.

Bibliography

[1] S. S. Daneshvar, Y. Nong, X. Yang, S. Wang, and H. Cai, *Vulscriber: Exploring rag-based vulnerability augmentation with llms*, ACM Transactions on Software Engineering and Methodology, 2025. [Online]. Available: https://doi.org/10.1145/3760775

[2] Y. Ding et al., *Vulnerability detection with code language models: How far are we?* 2024. arXiv: 2403.18624 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2403.18624

[3] X. Du et al., *Vul-rag: Enhancing llm-based vulnerability detection via knowledge-level rag*, 2024. arXiv: 2406.11147 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2406.11147

[4] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and cve summaries," in *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*, 2020, pp. 508–512. DOI: 10.1145/3379597.3387501

[5] Z. Feng et al., *Codebert: A pre-trained model for programming and natural languages*, 2020. arXiv: 2002.08155 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2002.08155

[6] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, *Unixcoder: Unified cross-modal pre-training for code representation*, 2022. arXiv: 2203.03850 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2203.03850

[7] D. Guo et al., "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[8] Y. Guo, C. Patsakis, Q. Hu, Q. Tang, and F. Casino, "Outside thenbsp;comfort zone: Analysing llm capabilities innbsp;software vulnerability detection," in *Computer Security – ESORICS 2024: 29th European Symposium on Research in Computer Security, Bydgoszcz, Poland, September 16–20, 2024, Proceedings, Part I*, Bydgoszcz, Poland: Springer-Verlag, 2024, pp. 271–289, ISBN: 978-3-031-70878-7. DOI: 10.1007/978-3-031-70879-4_14 [Online]. Available: https://doi.org/10.1007/978-3-031-70879-4_14

[9] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23, ACM, Nov. 2023, pp. 1865–1879. DOI: 10.1145/3576915.3623175 [Online]. Available: http://dx.doi.org/10.1145/3576915.3623175

[10] Y. Li et al., *Cleanvul: Automatic function-level vulnerability detection in code commits using llm heuristics*, 2025. arXiv: 2411.17274 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2411.17274

[11] Y. Li et al., "Out of distribution, out of luck: How well can llms trained on vulnerability datasets detect top 25 cwe weaknesses?" *arXiv preprint arXiv:2507.21817*, 2025.

[12] Z. Li et al., "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proceedings 2018 Network and Distributed System Security Symposium*, ser. NDSS 2018, Internet Society, 2018. DOI: 10.14722/ndss.2018.23158 [Online]. Available: http://dx.doi.org/10.14722/ndss.2018.23158

[13] R. More and J. S. Bradbury, "Assessing data augmentation-induced bias in training and testing of machine learning models," *arXiv preprint arXiv:2502.01825*, 2025.

[14] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai, "Vulgen: Realistic vulnerability generation via pattern mining and deep learning," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2527–2539. DOI: 10.1109/ICSE48619.2023.00211

[15] Z. Sheng, F. Wu, X. Zuo, C. Li, Y. Qiao, and L. Hang, *Lprotector: An llm-driven vulnerability detection system*, 2024. arXiv: 2411.06493 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2411.06493

[16] I. Vakilinia, D. K. Tosh, and S. Sengupta, "Privacy-preserving cybersecurity information exchange mechanism," in *2017 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, 2017, pp. 1–7. DOI: 10.23919/SPECTS.2017.8046783

[17] X. Zhou, T. Zhang, and D. Lo, *Large language model for vulnerability detection: Emerging results and future directions*, 2024. arXiv: 2401.15468 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2401.15468

[18] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, *Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks*, 2019. arXiv: 1909.03496 [cs.SE]. [Online]. Available: https://arxiv.org/abs/1909.03496