Transformer-Based Secure Code Generation and Vulnerability Repair Using Fine-Tuned CodeLlama and Retrieval-Augmented Generation

1st Younis Abdullah Al-Shibli
Department of Computing
Muscat College
Muscat, Oman
21140@email.muscatcollege.edu.om

2nd Ferddie Quiroz Canlas
Department of Computing
Muscat College
Muscat, Oman
ferddie@muscatcollege.edu.om

***Abstract*—Today's software systems are becoming more complex, and automatic detection and code remediation of software vulnerabilities becomes increasingly important for ensuring secure and functionally correct software code. Detecting security flaws is currently fairly effective, but creating secure alternative is not very effective. To fill this gap, here is an integrated framework that, on the test set of 262 examples, achieves a BLEU score of 30.65 and a CodeBERT embedding similarity score of 0.9643 that is impressively high in terms of semantic similarity to reference secure code. Qualitative results also illustrate the system's ability to successfully mitigate against various types of vulnerability within several programming languages such as SQL Injection, XSS, Path Traversal, Server-Side Request Forgery (SSRF) and Command Injection. We call our method a hybrid fine-tuning and RAG approach by adding a fine-tuning step to the pre-trained CodeLlama-13b model.To obtain these results, we propose a hybrid method consisting of fine-tuning a pre-trained CodeLlama-13b and a Retrieval-Augmented Generation (RAG) pipeline. Fine-tuning was performed through Quantized Low-Rank Adaptation (QLoRA) on a multilingual vulnerable and corrected code snippet dataset by quantizing to 4 bits.Fine-tuning is done using Quantized Low-Rank Adaptation (QLoRA) with a multilingual dataset of vulnerable code snippets and corrected code snippets in 4 bits quantization. The RAG pipeline uses cross-encoder reranking along with the BGE encoder, which is a feature of ChromaDB, to fetch relevant contextual knowledge from large CVE and CWE databases to prevent hallucinations and induce generation.**

Keywords—CodeLlama, QLoRA, Large Language Models, Secure Code Generation, Vulnerability Detection, Retrieval-Augmented Generation, Parameter-Efficient Fine-Tuning · · ·

I. Introduction

Nowadays, software security is crucial in several industries such as finance, healthcare, and education due to potential catastrophic impacts like data breach, financial loss, and reputational harm. With the increasing complexity of software systems, the problem of keeping the codebase secure has gained momentum. Vulnerability classes such as SQL injection, XSS, Path Traversal, SSRF, and Command Injection still represent issues for the security of applications in production environments. Current methods of software security appear ineffective in addressing this problem.

Conventional static and dynamic code analysis techniques are insufficiently equipped for solving such an issue. They do not possess enough generalizability, are mostly designed for detection rather than code repair, require manual effort from a developer and are mainly language-specific. Moreover, there is no automated tool capable of generating vulnerability-free code in real-time and in a multi-language environment.

With the emergence of LLMs, particularly transformer-based code models, there is a new way to address the problem above. Code models like CodeLlama have been demonstrating high performance in code completion tasks across several programming languages. However, general models such as CodeLlama are not necessarily optimized for code repair, and therefore they may generate code that reproduces patterns observed in their training corpora. Hence, domain-specific fine-tuning would be required.

PEFT techniques such as Quantized Low-Rank Adaptation (QLoRA) make it possible to efficiently fine-tune large language models on a domain-specific dataset without the need for hardware resources required by standard fine-tuning techniques. QLoRA applies low-rank adapter injection along with quantization at 4 bits in order to enable efficient model specialization. In addition to this technique, one could employ Retrieval-Augmented Generation (RAG) to further increase the effectiveness of code repair. Grounding generation in the context of a query using structured external knowledge from CVE and CWE databases would help reduce the hallucination rate.

In this paper, we demonstrate how to use a QLoRA fine-tuned transformer model together with RAG in order to generate secure code across programming languages. Specifically, our model uses a RAG pipeline that utilizes BGE embeddings for similarity search and chromadb as its vector storage with cross-encoder reranking. This combination allows the model to ground generation in CVE/CWE data and produce secure code with decreased likelihood of errors.

Gradio web application serves as the user interface that enables one to interact with the model in real time, submit their vulnerable code and receive back a secure code sample and explanation of vulnerabilities detected by the model.

The primary contributions of this paper are:

- Domain-specific QLoRA fine-tuning process of a CodeLlama code model using quantization at 4 bits. The data consists of vulnerable/secure code pairs collected from CVE and CWE repositories.
- Development of two-stage RAG pipeline based on BGE embeddings and chromadb vector database with cross-encoder reranking that retrieves precise vulnerability context during inference.
- Development of end-to-end secure code generation system capable of automatically generating secure code across several programming languages including Python, PHP, Javascript, and C. Generated code repairs vulnerabilities such as XSS, SQL Injection, Path Traversal, SSRF, and Command Injection.
- Gradio-based web app that allows one to interact in real time with the proposed system, submitting their code and receiving back code diagnostics and generation suggestions.

The rest of this paper is organized as follows. Comprehendible review of related studies and previous research has been carried out in Section II. The methodology and system architecture proposed is explained in Section III, with a dedicated subsection for the experimental setup. The obtained experimental results and detailed discussion are given in section IV. Last, Section V wraps up the paper and suggests some avenues for future work.

II. Review of Related Literature

A. Parameter-Efficient Fine-Tuning (PEFT) and Code LLMs

QLoRA is a 4-bit quantization parameter-efficient fine tuning method introduced by Dettmers et al. [1] that is able to achieve full fine tuning performance using significantly less GPU memory for fine tuning. Using this approach, you are able to specialise large models on resource-constrained hardware. The technique draws its origins from a study named LoRA by Hu et al. [2] which showed that updating low-rank matrices in the layers of a transformer could yield competitive results with respect to the full parameter updates. Rozière et al. [3] built upon this, creating a series of state-of-the-art code-specialized foundation models, Code Llama, as the backbone of many code pipelines for code generation and understanding. These works offer solid foundations in terms of the calculations and the architecture, but typically do not include more specialized security optimizations.

B. Retrieval-Augmented Generation and Code Repair

The Retrieval-Augmented Generation (RAG) framework was introduced by Lewis et al. [4] as a new approach that shows that incorporating external text databases improves factual accuracy and reduces hallucinations in knowledge-intensive applications. Afrin et al. [5] carried out a systematic literature review of 27 papers on PEFT techniques, and found a lack of security-annotated fine-tuning datasets and security-centric goals for fine-tuning. Moreover, Feng et al. [6] proposed a pre-trained model named CodeBERT that can learn the semantic relationship between the natural language and programming code. While the code models, such as CodeBERT, have been extensively used for code semantic similarity assessment, they have not been extensively researched for their use in integrated real-time patching pipelines.

C. Parameter-Efficient Fine-Tuning for Large Code Models: A Systematic Review

In their systematic literature review, Afrin et al. [5] discussed all PEFT methods, such as LoRA, QLoRA, and adapter-based methods, used for various code-related tasks on large code models. The analysis of those 27 studies (conducted from 2020 to 2025) showed that PEFT strategies are strongly effective in decreasing the computational expenses without compromising performance attributes compared with the full fine-tuning approach. There were, however, two important gaps in previous work that the survey found: First, little existing research considers software security applications, and second, there was a lack of good quality, security-annotated optimization test sets. The literature review highlighted several gaps, which the present work investigates, such as building a multilingual code library from vulnerable and fixed code pairs in CVEs and CWEs repositories, and fine-tuning using QLoRA in a customized training strategy specific to the security domain.

D. LLMs in Software Security and Vulnerability Detection

They surveyed the state of the art in software security methods based on language models by Sheng et al. [8] that focus on techniques for detecting vulnerabilities, fixing them and assisting developers to write secure code. In their survey, they identified key areas of weakness in current techniques, including data imbalance and lack of generalization. In addressing empirical threats, Pearce et al. [9] analyzed the security of LLM-generated code by examining the security issues in the code generated by GitHub Copilot, showing that about 40% of the generated code had security vulnerabilities. Likewise, Sandoval et al. [10] performed a formal user study, which studied security aspects of LLM-generated code for C, and found that a significant drawback of using LLMs is that they do not necessarily enhance its security. In an attempt to reconcile code quality and fine-tuning, Haider et al. [11] scored use of a fine-tuned Code Llama model with QLoRA for creating automatic code review comments. Their focus, however, was on code quality and style offers that didn't include automatic security vulnerability remediation.

E. Research Gap

From the literature analysis presented above, it is evident that there is a gap in security-aware fine-tuning with efficient specialization and external knowledge. Previous work either concentrates on using PEFT without looking into any security concerns [1], [2], [5], looks at the vulnerabilities or weaknesses in models but does not offer automatic correction [9], [10], or does not have context in place to prevent hallucination problems [11], [8]. The current work addresses the problem and uses a fine-tuned QLoRA model of Code Llama with a two-stage RAG engine from CVE and CWE databases.

III. METHODOLOGY

A. System Overview

The design of the suggested solution was done using CRISP-DM [12] (Cross-Industry Standard Process for Data Mining), an iterative approach that was chosen due to its effectiveness for implementing machine learning and AI-related projects. CRISP-DM includes a set of six clearly defined stages such as business understanding, data understanding, data preparation, modeling, evaluation, and deployment that help ensure that all parts of the system undergo validation and delivery process effectively. Overall, the architecture of the proposed system encompasses two main components: QLoRA fine-tuned CodeLlama model used for generating code in a safe way and RAG retrieval pipeline for getting structured knowledge on vulnerabilities, as shown in Fig. 1

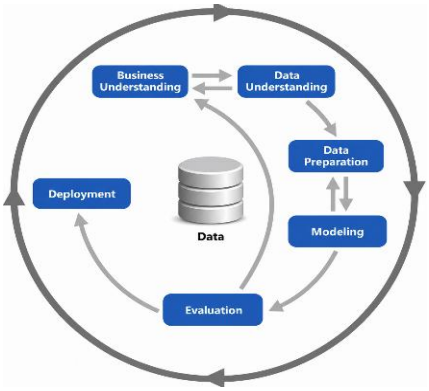

Figure 1: CRISP-DM Methodology

B. Data Understanding

Two datasets were used in the development of the hybrid approach. The first one is the CVEfixes, a large dataset of commits that fix vulnerabilities in open-source software. This dataset consists of more than 12,000 commits, around 51,000 files that have been modified, and more than 138,000 function-level pairs of vulnerable and fixed code with CVE identifiers, CWE identifiers, and 272 vulnerability categories. This particular dataset was used for fine-tuning because of its valuable content that allows the model to learn how to transform the insecure code into a semantically equivalent one. The second dataset used was called CVE and CWE Dataset 1999-2025 and included thousands of vulnerability entries with CVE identifiers, CVSS scores, CWE IDs, and descriptions. This dataset was not fine-tuned, but it was indexed into a vector database to become an external knowledge source for the Retrieval-Augmented Generation (RAG) pipeline when making predictions. Thus, the combination of these two datasets forms a complementary system where CVEfixes enables generating code, and CVE/CWE dataset improves factual information about vulnerabilities.

C. Data Preparation

Data preprocessing was done independently for both the fine-tuning pipeline and RAG pipeline. In particular, vulnerable and corresponding fixed code pairs were collected from the CVEfixes dataset and processed through an elaborate preprocessing pipeline including the removal of version diffs, whitespace cleaning, and comment stripping; formatting normalization; conversion of each sample to a structured text-to-text format where the inputs are vulnerable code accompanied by a structured instruction prompt and outputs are secure code with a vulnerability description. Length token filtering's were used to make sure all samples adhere to the maximum context window size allowed by the transformer architecture while meeting minimum requirements for the code length. To avoid data leakage between train/validation /test splits, CVE-based splitting was used to guarantee that vulnerabilities of a certain type do not appear in both train, validation, and test sets. Finally, the prepared dataset is hosted on the Hugging Face platform to facilitate the loading process during training.

On the other hand, preprocessing of the CVE/CWE dataset for semantic indexing purposes involved the cleaning of textual fields, standardization of CVE and CWE IDs, and removal of duplicated and missing records. Next, each record was converted to a structured document and split into chunks appropriate for semantic retrieval. A prefix "passage:" was added to each chunk to conform to input format requirements for the BGE model.

D. Model Architecture

In this paper, the base model that was used is CodeLlama-13B which is a transformer based large language model developed by Meta AI using LLaMA 2 architecture. The model has been pretrained on vast amounts of open source codes written in programming languages such as Python, C++, JavaScript, and PHP. CodeLlama-13B features an internal architecture comprised of 32 transformer layers with self-attention using Rotational Positional Encodings (RoPE) along with RMS Layer Norm and Residuals. The model uses SwiGLU Feed Forward networks with the token representation going through these layers sequentially before being passed to a linear output layer in order to generate tokens, the model architecture is shown in the Fig. 2.

QLoRA Fine tuning was done by adding low rank adapter matrices in all the layers of the frozen quantized 4-bit backbone. This means that only adapter matrices were learned for attention projections while the other parameters were left frozen. Parameters for adapter matrix rank and scaling factor were carefully selected so that the model could be specialized to learn code transformations geared towards enhancing cybersecurity measures without having to train any parameters.

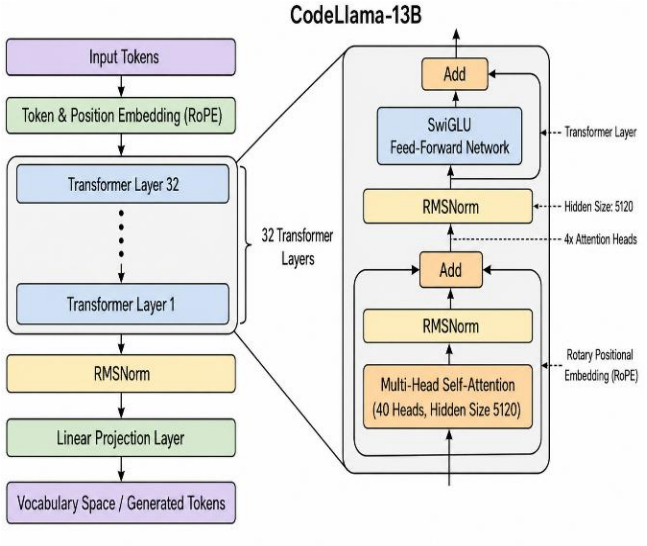

Figure 2: Model Architecture

E. RAG Pipeline Design

The RAG pipeline was created in a two-step fashion in order to maximize the precision of the vulnerability context provided to the model during inference. First, the programming language of the submitted code and high-level security context of this code are identified via the use of context keyword dictionaries and vulnerability normalization mapping, a process which normalizes vocabulary used in security context as well as links vulnerability names to corresponding CWE identifiers. Using the obtained information, the first step query is generated to retrieve initial candidates using ChromaDB and the BAAI/bge-large-en-v1.5 embedding model.

Next, using the extracted vulnerability type a more focused query is formulated which is used in a similar way in order to identify a smaller number of relevant candidate documents. A cross-encoder reranking model then evaluates the relevance of all candidates based on the provided query and selects those that are most relevant to the task. This information is then used to build a structure-based prompt for fine-tuning a language model, allowing for generating a secure code rewrite alongside vulnerability explanation.

In order to overcome the structural limitations of the context window of the pre-trained language model during its inference phase, an efficient text-splitting setting is designed within the vector database design. As a result of the native context length being 4096 tokens for the CodeLlama-13b model, recursive splitting of both CVE description data and CWE taxonomy entries is applied upon importing them to ChromaDB via Recursive Character Text Splitter. The document chunks are set with 512 tokens as the fixed chunk size and 50 tokens of chunk overlap. Thus, semantic information about the security flaws will be preserved, and at the same time, out-of-memory issues are avoided.

F. End-to-End Secure Code Generation Pipeline

The entire process works as follows. The user provides their source code that suffers from vulnerabilities via the website interface. The source code goes through preprocessing and tokenization using the CodeLlama tokenizer. At the same time, the RAG pipeline extracts security context information, creates the queries for two stages of processing, retrieves relevant documents related to CVE/CWE, reranks the documents, and prepares the augmented prompt for CodeLlama. Then, the fine-tuned CodeLlama model processes the prompt and generates the code that is free of vulnerabilities and provides an explanation of the vulnerabilities in the original code. Finally, the system uses rules for secure coding and validates whether the generated code avoids the original vulnerability class, the pipeline is shown in the Fig. 3.

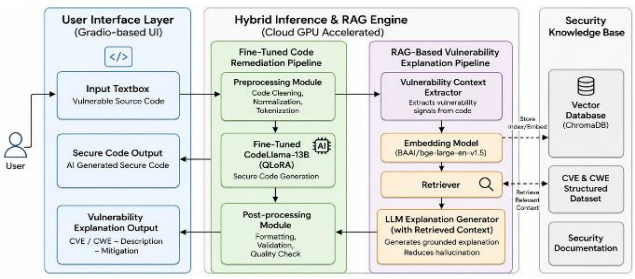

Figure 3: End-to-End Secure Code Generation Pipeline

G. Deployment Architecture

The architecture is organized based on three logical tiers. The User Interface tier is implemented using Gradio at version 5.49.1, enabling a web-based user interface where users can input their code in different programming languages and get the result back in real-time. The Hybrid Inference Tier uses the fine-tuned CodeLlama-13B model for securing the code alongside the RAG pipeline for vulnerability explanations. All these processes are facilitated by LangChain version 1.2.8 to orchestrate the pipeline while ChromaDB version 1.5.3 is used for storing and retrieving the vector representation of the models. Lastly, the Output Delivery Tier presents the model's response in an understandable format with both the secure code rewrite and vulnerability explanation presented together. This whole inference stack was run on an NVIDIA GPU on Google Colab, while the models were downloaded from the Hugging Face hub and stored on Google Drive.

H. Experimental Setup

1) Hardware and Software Environment:

All the training and inference were performed on Cloud GPU system through Google Colab. A system with the Intel Core i5 CPU, 8 GB of RAM was used for data preprocessing and interface development, and all intensive calculations with the use of GPUs were carried out with the use of cloud infrastructure. The applications feature Python 3.12.12 as the base programming language, PyTorch 2.9.0 [16] for neural network work, Hugging Face Transformers 4.57.1 [15] for loading, preprocessing, and fine-tuning models, the PEFT library for the configuration of low-rank adapters [2] and BitsAndBytesConfig for 4-bit weight quantization [1]. Data manipulation and analysis was done using Pandas 3.0.1. The RAG system was implemented through a combination of LangChain 1.2.8, an open-source chain of programs for developing generative AI applications, and ChromaDB 1.5.3, an open-source vector database embedding store for storing vectors and embedding data, respectively. Gradio 5.49.1 was used to deploy the user-interface. For BLEU score calculation, Quantitative evaluation was conducted using SacreBLEU, and for semantic code similarity metrics Sentence Transformers [14] and CodeBERT [6] were used.

2) Dataset Configuration:

For domain-specific fine-tuning, the dataset was selected from the repository known as CVEfixes [7] which includes more than 12,000 commits that fix vulnerabilities and 51,000 changed files and more than 138,000 pairs of vulnerable/fixed function code for a total of 272 classes of vulnerabilities . A test set of 262 different examples was created in various target languages (Python, PHP, JavaScript, C) after comprehensive data cleaning, CVE respectively CWE based splitting and strict filtering of CVE length.The external knowledge base for the RAG pipeline was created separately based on the CVE and CWE Dataset (with the data set from 1999 to 2025). This is a dataset with thousands of records with the CVE Number, CWE classification, CVSS severity scores, and detailed text description. To support fact-grounded retrieval [13], [4], these're converted to documents and recursively split into

chunks of fixed-size 512 tokens, with an hop-over of 50 tokens per chunk, and prepended by the prefix "passage:".

3) Fine-Tuning Configuration:

The model used was the base CodeLlama-13B model (meta-llama/CodeLlama-13b-hf) loaded in 4-bit (NormalFloat4 - NF4) quantized format. Parameter-efficient specialization [2] was achieved by injecting trainable LoRA adapter matrices into all the attention projection layers through the PEFT package. The Training Arguments class was used to set key training arguments such as training epochs, per-device batch size, gradient accumulation steps and target learning rate. We used a token data collator to prepare batches when training. Care was taken to prevent possible overfitting and to optimise the convergence of the model by tracking the training/validation loss on an epoch-wise basis and implementing an early stopping callback to stop the training when the loss stops improving and before divergence. Finally, if the convergence is a success, the converged adapter parameters were saved and uploaded to the Hugging Face model repository.

4) Evaluation Metrics:

To validate the effectiveness of the suggested model quantitatively on the test dataset of 262 samples, two different metrics were used. First, we measured the BLEU score by the use of the SacreBLEU package to measure the exact n-gram overlap between the generated code and the ground truth. Second, we measured the CodeBERT Embedding Similarity that uses CodeBERT encoder [6] to map both generated and reference patches into the high-dimensional space and to compute cosine similarity. Unlike lexical measures, the latter approach can capture semantic equivalence and functional correctness regardless of the code patterns in the generated code. In our experiments, the maximum number of generation tokens for testing varied depending on the reference patch size, and the caching approach was used to execute multiple inference iterations.

5) Baseline Systems:

To study the isolated effect of each architectural element of the hybrid system, the proposed model was evaluated in comparison to several baseline variants. These baselines consist of the standard, non fine-tuned base CodeLlama model to serve as a baseline to evaluate the effect of domain specific training [3]; standard full fine-tuning and LoRA based fine-tuning methods to evaluate memory-performance trade-offs [2]; and variants of not utilizing the RAG pipeline to illustrate the importance of grounding in an external database to reduce hallucinations [4]. Moreover, the system was compared with several popular rule-based and commercial code assistants to confirm its ability to repair cross-lingual code and reproduce results [10], [9]. The above more detailed comparative information is embedded in Tables II and III IV and V of the results section.

IV. Results and Discussion

A. Fine-Tuning Performance

Monitoring of the tuning process occurred by tracking training and validation loss throughout training epochs. Both the training and validation loss curves continued to converge towards each other, while the validation loss continued to decrease, which means that the model was learning how to transform vulnerable code into secure code, and there was no overfitting in doing so. The use of the early stopping technique helped prevent overfitting by ending training before divergence of both curves could begin.

B. Quantitative Evaluation Metrics

The optimized system was then tested against the 262-example test data set through BLEU score and embedding similarity based on CodeBERT model. The testing results are shown in Table I.

TABLE I: EVALUATION METRICS RESULTS

Metric	Score
BLEU Score	30.65
Embedding Similarity	0.9643

The BLEU score of 30.65 reveals significant lexical similarities between generated secure code and reference code, revealing that the system is able to accurately reproduce secure pattern structures in most cases. The score of embedding similarity at 0.9643 shows that generated code is semantically aligned with reference code and is able to remove the target vulnerability without disrupting program semantics and functionality. High embedding similarity compared to BLEU score is anticipated since there can be multiple ways of implementing a piece of code that have different syntax but identical meaning in terms of program logic.

C. Comparison with Baseline Systems

To place the results of the system performance into context, an assessment was made based on comparisons with various baseline settings and presented in Table II.

TABLE II: BASELINES FOR COMPARING THE PROPOSED SYSTEM

System	Performance Level	CVE/CWE Accuracy	Code Fix Capability
Proposed System (QLoRA + RAG)	High	High (RAG-grounded)	Supported
Base CodeLlama (no fine-tuning)	Low	Low	Limited
General LLM (GPT-style)	Moderate	Moderate (hallucination-prone)	Partial
Static Analysis Tools	Moderate	Detection Only	Not Supported

CodeLlama without domain adaptation failed to provide good performance in terms of repairing vulnerabilities due to the absence of specialized training signals. In general, there was evidence of decent code generation performance; however, the presence of factual errors in the form of hallucinations with regard to CVE and CWE entries limited the usefulness of LLMs. On the other hand, static analysis provided reliable results in vulnerability detection but was not able to perform any repairs automatically.

D. Impact of RAG Integration

Without RAG in the loop, the hybrid solution used solely the inner parametric knowledge of the fine-tuned model, which often led to semantic drifts, generic explanations, or wrong CVE/CWE mappings. In fact, it is worth noting that a clear division of responsibilities between components in our approach is achieved: The RAG-based subsystem acts only as a knowledge base for the generation of comprehensive security-related explanations by pulling information from ChromaDB independently of the source code structure.

On the contrary, the QLoRA fine-tuned CodeLlama model takes care of code logic synthesis and repair. As can be seen from Table III, feeding only the isolated security explanation produced by RAG into the model reduces the amount of hallucinations per token significantly. The use of the cross-encoder reranking stage allows selecting the most accurate CVE data to feed into the fine-tuned model before generating the code itself, forcing it to follow safety guidelines.

TABLE III: EFFECTS OF RAG INCLUSION

Aspect	With RAG	Without RAG
CVE Accuracy	High	Low
CWE Classification	High	Moderate
Hallucination	Low	High
Explanation Quality	Detailed	Generic

When RAG was excluded from the experiment, the system made use only of the fine-tuning of its inner parameters, which generated explanations with irrelevant, inaccurate, or invented CVE and CWE. However, when RAG was included, the two-step retrieval pipeline provided the system with correct and accurate vulnerability metadata, generating a much better explanation as compared to one without the RAG pipeline. The inclusion of a cross-encoder reranking step helped select the most relevant documents from among the candidates. Thus, it can be concluded that RAG plays a crucial role in the system's functionality because it controls the facts stated in the explanation.

E. Comparison of Fine-Tuning Methods

Comparing the performance and efficiency of various fine-tuning techniques was conducted based on information from Table IV below.

TABLE IV: DIFFERENT FINE-TUNING TECHNIQUES COMPARED

Method	GPU Memory	Performance	Practicality
QLoRA (Proposed)	~14 GB	High	Excellent
LoRA	~24 GB	Comparable	Good
Full Fine-Tuning	>80 GB	Slightly Higher	Poor
Prompt Tuning	Low	Weak	Limited

Compared to other methods, full fine-tuning yielded slight improvements in terms of performance but had excessive memory usage, more than 80 GB. This method is therefore only applicable to researchers having access to advanced multi-GPU systems. Standard LoRA improved memory usage to about 24 GB without compromising performance. Despite this achievement, standard LoRA still requires large amounts of memory compared to what can be accessed using cloud computing environments. The use of QLoRA lowered memory usage further to around 14 GB while maintaining model performance; it is thus a suitable technique for constrained research setups. Prompt tuning could not deliver satisfactory performance levels since the model parameters cannot be updated effectively.

F. Comparison with Industry and Research System

The proposed approach has been further compared to commercial solutions and research-based systems as given in Table V below.

TABLE V: COMPARISON WITH INDUSTRY AND RESEARCH SYSTEMS

System	Code Fix	CVE/CWE Reference	Hallucination Control	Availability
Proposed System	✓	✓	✓	Open-Source
GitHub Copilot	Partial	✗	Limited	Commercial
SonarQube	✗	Rule-Based	N/A	Freemium
Snyk Code	Partial	Partial	N/A	Freemium

The proposed approach is the unique approach that provides automation of vulnerability correction while providing factual reference for CVE and CWE along with management of hallucinations using retrieval augmentation generation (RAG) and cross-encoder reranking. The commercial solutions like GitHub Copilot and rule-based solutions like SonarQube and Snyk Code provide limited capability in vulnerability detection and suggestion but not in automated vulnerability correction and explanations.

G. Case study

In order to assess the real-life efficiency of our suggested solution, we have performed a qualitative case study of the following two most common types of software security bugs: XSS and SQLi.

As seen in Table VI below, our system is able to bridge the gap between vulnerability detection and automatically fixing the bug in the source code.

When it comes to the vulnerability CWE-79, i.e., XSS, at the beginning, the base model did not consider output sanitization of the variable and, therefore, the injected user input was vulnerable to script execution. Thanks to adding to our system the RAG pipeline responsible for retrieving context information from related CVEs, our codeLlama model detected the sink in the source code and embedded the htmlspecialchars() function preventing potential payload execution.

When it comes to SQL Injection (CWE-89), a vulnerable input was concatenating raw HTTP GET parameters into the SQL statement. By analyzing this bug from the data-plane perspective, the system was able to refactor the process of accessing the database, employing prepared statements in the process.

TABLE VI: QUALITATIVE CASE STUDY

Code Snippet / Metadata Details	Vulnerability Evaluation Aspect

`<?php`
`$user_name = $_GET['name'];`
`echo "<h1>Welcome back, " .`
`$user_name . "!</h1>";`
`?>`
	Vulnerable code

`<?php`
`$user_name =`
`htmlspecialchars($_GET['name']);`
`echo "<h1>Welcome back, " .`
`$user_name . "!</h1>";`
`?>`
	Secure Code

CVE/CWE: CVE-2001-1524 | CWE-79 Type: Cross-Site Scripting (XSS) Severity: MEDIUM (CVSS: 4.3) Fix Applied: Enforced htmlspecialchars() to sanitize user input before rendering, neutralizing arbitrary HTML/script injection.	RAG Explanation

`<?php`
`function build_query($x) {`
`  return "SELECT * FROM users`
`WHERE id = '" . $x . "'";`
`}`
`$user_id = $_REQUEST['user_id'];`
`$sql = build_query($user_id);`
`$result = $conn->query($sql);`
`?>`
	Vulnerable code

`<?php`
`function build_query($x) {`
`  return "SELECT * FROM users`
`WHERE id = ?";`
`}`
`$user_id = $_REQUEST['user_id'];`
`$stmt = $conn-`
`>prepare(build_query());`
`$stmt->bind_param('i', $user_id);`
`$stmt->execute();`
`$result = $stmt->get_result();`
`?>`
	Secure Code

CVE/CWE: CVE-2022-21449 | CWE-89 Type: SQL Injection (SQLi) Severity: HIGH (CVSS: 7.5) Fix Applied: Enforced Prepared Statements to treat input as data rather than executable SQL commands.	RAG Explanation

H. Discussion

Based on the above quantitative and qualitative findings, one can state that the hybrid solution developed performs well along both axes of the problem of secure code generation. Indeed, high embedding similarity values prove that the combination of domain-specific fine-tuning via QLoRA allows for effective pattern encoding necessary for producing semantically accurate patches even in cases when the degree of surface lexical similarity does not reach a high value. RAG is an indispensable part of the pipeline in terms of explanation quality since it translates general information regarding vulnerabilities into CVE/CWE-based solutions applicable for developers. In contrast to fine-tuning models, which require more resources, the usage of QLoRA makes this system reproducible via GPUs available in the cloud. However, a limitation of the research design used here is that BLEU and embedding similarity values show the distance between generated fixes and only one reference fix, while several solutions are possible for one vulnerability.

V. Conclusion and Future work

In this paper, an automated secure code generation and vulnerability repair framework based on fine-tuning a 13B version of the CodeLlama model, followed by a two-stage Retrieval-Augmented Generation (RAG) pipeline to leverage CVE and CWE databases, was presented. The system was created to fill the critical need for tools that can detect, but not seamlessly and accurately remediate and explain vulnerabilities in a factually informed manner.

The proposed model showed a promising performance when evaluated quantitatively as well as qualitatively. It was trained on a multi-lingual data set of 262 samples, extracted from the CVEfixes data set, and, on this test set, its BLEU score has reached 30.65 and its code embedding similarity score 0.9643 based on the CodeBERT embedding model, indicating that the fine-tuned model produces secure code that is lexically reasonable and semantically faithful to the reference implementations. BGE with retrieval augmented by ChromaDB and cross-encoder reranking significantly reduced hallucinations in vulnerability explanations and resulted in accurate CVE and CWE attribution without retrieval augmentation. Testing showed a cross section of programming languages and vulnerabilities, like PHP, Python, C, cross section of SQL Vulnerability, XSS Vulnerability, Command Injection Vulnerability, Path Traversal Vulnerability and SSRF Vulnerability across five classes of vulnerability, that proved to show a generalization of the system across the vulnerabilities as well as across the programming languages. One of the most prominent practical benefits over full fine-tuning approaches that require over 80 GB of GPU memory, was the reduced memory demand of QLoRA which enabled

the complete training and deployment procedure to be reproduced on standard cloud-based GPU infrastructure with around 14 GB of memory.

There are some drawbacks to the existing system and these need to be recognised. The amount of repair is limited by the data coverage, and/or quality of training data set to the model, which might restrict generalization for rare and novel vulnerability classes that are not sufficiently present in CVEfixes. However, inferences on larger code inputs are non-trivial and will have computational cost which will affect real-time usability in production. The existing web interface is not intended to be used for production scale deployments, but was created for research and demo. Further, the results of the evaluation used single-reference comparisons,leading to a possibly underestimation of performance if there are more than one valid safe rewrite.

Future research will investigate integrating more types of vulnerabilities into the training set, developing more effective methods for fine-tuning to achieve even higher generation output scores, and rolling out a pipeline-based architecture to a multi-stage retrieval and reasoning with agents framework. The most effective approach for system integration would be from an IDE like Visual Studio Code so that vulnerabilities could be identified during active program development and the systems repaired in real-time.

REFERENCES

[1] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized LLMs," arXiv preprint arXiv:2305.14314, 2023.

[2] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," arXiv preprint arXiv:2106.09685, 2022.

[3] B. Rozière et al., "Code Llama: Open foundation models for code," arXiv preprint arXiv:2308.12950, 2023.

[4] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in Advances in Neural Information Processing Systems (NeurIPS), 2020.

[5] S. Afrin, M. Rahman, and R. Islam, "Parameter-efficient fine-tuning for large code models: A systematic review," ACM Computing Surveys, 2025.

[6] Z. Feng, H. Wang, et al., "CodeBERT: A pre-trained model for programming and natural languages," arXiv preprint arXiv:2002.08155, 2020.

[7] Zenodo, "CVEfixes: Automated vulnerability-fixing dataset (Version 3.1)," 2024. [Online]. Available: https://doi.org/10.5281/zenodo.13118970

[8] Y. Sheng, Z. Li, S. Kumar, M. Patel, and Y. Zhao, "LLMs in software security: A survey of vulnerability detection techniques and insights," IEEE Transactions on Software Engineering, 2025.

[9] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in IEEE Symposium on Security and Privacy, 2022.

[10] G. Sandoval, H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Lost at C: A user study on the security implications of large language model generated code," in USENIX Security Symposium, 2023.

[11] M. S. Haider, M. Mostofa, S. Bin Mosaddek, S. Iqbal, and F. Ahmed, "Prompting and fine-tuning LLMs for automated code review comment generation," Journal of Software Engineering Research, vol. 12, no. 3, pp. 45–61, 2024.

[12] R. Wirth and J. Hipp, "CRISP-DM: Towards a standard process model for data mining," in Proceedings of the 4th International Conference on the Practical Applications of Knowledge Discovery and Data Mining, 2000.

[13] L. Gao, X. Ma, J. Lin, and J. Callan, "Retrieval-augmented generation for large language models: A survey," arXiv preprint arXiv:2312.10997, 2023.

[14] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2019.

[15] T. Wolf, L. Debut, V. Sanh, et al., "Transformers: State-of-the-art natural language processing," in Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, 2020.

[16] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in Advances in Neural Information Processing Systems, vol. 32, 2019.

[17] V. Karpukhin et al., "Dense passage retrieval for open-domain question answering," in Proceedings of EMNLP, 2020.

[18] A. Vaswani et al., "Attention is all you need," in Advances in Neural Information Processing Systems (NeurIPS), 2017.

[19] ChromaDB, "Chroma: The open-source embedding database," 2023. [Online]. Available: https://www.trychroma.com

[20] LangChain, "LangChain: Building applications with LLMs through composability," 2023. [Online]. Available: https://www.langchain.com