TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Keisuke Konno 233917IVCM

Evaluating and Fine-tuning Large Language Model for Code Vulnerability Detection Across Programming Languages and Code Granularities

Master's Thesis

Supervisor: Hayretdin Bahsi
PhD

Co-supervisor: Danilo Gligoroski
PhD

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Keisuke Konno 233917IVCM

LLM-IDE HINDAMINE JA TÄIUSTAMINE KOODI HAAVATAVUSE TUVASTAMISEKS ERI KEELTES JA KOODIKOGUMITES

Magistritöö

Juhendaja: Hayretdin Bahsi
PhD

Kaasjuhendaja: Danilo Gligoroski
PhD

Tallinn 2025

Author’s Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Keisuke Konno

18.05.2025

Abstract

This thesis investigates the effectiveness of large language models (LLMs) in classifying code vulnerabilities in multiple programming languages and at various levels of code granularity, including file and function levels. The study addresses three key research questions: (1) How effectively can LLMs detect vulnerabilities across different programming languages? (2) How does detection performance vary between file-level and function-level analysis? (3) Does fine-tuning reduce false positive and false negative rates in multi-class vulnerability classification?

This study evaluates both unmodified and fine-tuned versions of the Microsoft Phi-4 model using a customized subset of the CVEfixes dataset. The selected samples include both vulnerable and patched (non-vulnerable) code from six programming languages: PHP, JavaScript, Java, TypeScript, Ruby, and Python and focus on four Common Weakness Enumeration (CWE) types: CWE-79, CWE-89, CWE-434, and CWE-352. To assess model performance, three prompting strategies are applied: zero-shot, few-shot, and chain-of-thought.

The results show that detection performance varies across programming languages, highlighting the influence of language-specific characteristics on model accuracy. Input granularity is important, as file-level analysis provides broader context that helps identify vulnerabilities spanning multiple functions and reduces false positives. Fine-tuning improves classification performance across different prompt strategies and input configurations, lowering both false positive and false negative rates.

These insights aim to guide and motivate further advancements in LLM-based vulnerability detection systems.

The thesis is written in English and is 95 pages long, including 7 chapters, 44 figures and 13 tables.

Annotatsioon

LLM-ide hindamine ja täiustamine koodi haavatavuse tuvastamiseks eri keeltes ja koodikogumites

Käesolevas töös uuritakse suurte keelemudelite (Large Language Models - LLM) efektiivsust turvaaukude klassifitseerimisel eri programmeerimiskeeltes ning erinevatel koodi detailsusastmetel, sealhulgas faili ja funktsiooni tasandil. Uuring keskendub kolmele põhilisele uurimisküsimusele: (1) Kuivõrd tõhusalt suudavad suured keelemudelid tuvastada turvaauke erinevates programmeerimiskeeltes? (2) Kuidas erineb tuvastamise täpsus faili- ja funktsioonitasandi analüüsi vahel? (3) Kas peenhäälestamine (fine-tuning) aitab vähendada valepositiivsete ja valenegatiivsete tulemuste osakaalu mitme klassiga turvaaukude klassifitseerimisel?

Uurimistöös hinnatakse nii originaalset kui ka peenhäälestatud Microsoft Phi-4 mudelit, kasutades kohandatud alamhulka CVEfixes'i andmekogust. Valitud proovid sisaldavad nii haavatavat kui ka parandatud (mittehaavatavat) koodi kuues programmeerimiskeeles: PHP, JavaScript, Java, TypeScript, Ruby ja Python ning keskenduvad neljale levinud turvanõrkuste klassifikatsioonile (Common Weakness Enumeration - CWE): CWE-79, CWE-89, CWE-434 ja CWE-352. Mudelite efektiivsuse hindamisel rakendatakse kolme päringustrateegiat (prompting strategies): näideteta (zero-shot), väheste näidetega (few-shot) ning mõtteahelaga (chain-of-thought).

Tulemused näitavad, et turvaaukude tuvastamise täpsus sõltub programmeerimiskeelest, rõhutades keelespetsiifiliste omaduste mõjule mudeli täpsuses. Sisendi detailsusaste mõjutab tulemusi oluliselt: failitasemel analüüs pakub laiemat konteksti, mis võimaldab paremini tuvastada funktsioonideüleseid turvaauke ning vähendab valepositiivsete tulemuste hulka. Mudelite peenhäälestamine parandab klassifitseerimise täpsust kõikide päringustrateegiate ja sisendkonfiguratsioonide puhul, vähendades nii valepositiivsete kui ka valenegatiivsete tulemuste esinemissagedust.

Need tulemused pakuvad suuniseid ja motivatsiooni suurte keelemudelipõhiste turvaaukude tuvastamise süsteemide edasisteks arendusteks.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 95 leheküljel, 7 peatükki, 44 joonist, 13 tabelit.

List of Abbreviations and Terms

LLM	Large Language Model
CWE	Common Weakness Enumeration
CVE	Common Vulnerabilities and Exposures
CoT	Chain-of-Thought
FS	Few-Shot
ZS	Zero-Shot
PEFT	Parameter-efficient Fine-tuning
QLoRA	Quantized Low-Rank Adaptation
RoPE	Rotary Position Embedding
LoRA	Low-Rank Adaptation
FT	Fine-tuning
FT-explain	Fine-tuning with reasoning-augmented input
FNR	False Negative Rate
FPR	False Positive Rate
TP	True Positive
FP	False Positive
TN	True Negative
FN	False Negative
Ma	Macro-averaged

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

Identifying and preventing vulnerabilities is critical in software development. Neglecting vulnerability assessment can expose systems to potential attacks and compromise sensitive data. The growth of libraries, frameworks, and dynamic programming languages has led to an increase in reported Common Weakness Enumerations (CWEs), making vulnerability assessment more critical than ever [1].

Large language models (LLMs) have become powerful tools for vulnerability assessment and code analysis By leveraging their understanding of programming languages, coding patterns, and software engineering best practices, LLMs can effectively identify potential security flaws and enhance overall code quality [2]. Unlike traditional static analysis tools that often depend on complete program contexts or compilation databases, LLMs can generate comprehensive test cases and synthesize meaningful inputs even in the absence of these prerequisites [3]. Additionally, LLMs can scale to large codebases, detecting common programming errors, unsafe coding practices, and known vulnerabilities. Beyond detection, LLMs offer detailed explanations to help developers assess issue severity, prioritize remediation efforts, and implement effective fixes [4].

1.2 Problem Statement

Current research reveals important areas of gaps in the effective application of LLMs for code security analysis. One major limitation is the lack of clarity on whether LLMs can reliably detect vulnerabilities across different programming languages. Most existing studies [5, 6, 7] focus on a single language, such as C, C++, leaving their performance on widely used languages such as Python, Java, and JavaScript uncertain. Another challenge is that LLMs are typically evaluated at the function level and not at broader context. As the study [8] points out, function-level analysis alone can overlook vulnerabilities that arise from the interaction between components. A function may appear secure in isolation yet become vulnerable in specific contexts. File-level analysis preserves critical relationships between functions, variables, and dependencies. Therefore, incorporating both function and file-level classifications is essential for comprehensive vulnerability detection. In addition, LLMs often have difficulty distinguishing between vulnerable and non-vulnerable code and similar vulnerability classes, leading to false positives and false negatives [6, 9].

1.3 Research Objectives and Questions

This thesis investigates the ability of LLMs to address these challenges. First, we investigate whether LLMs can effectively analyze code samples from multiple languages by testing their adaptability across 6 different languages. Second, we compare detection performance at different granularities: file level and function level, to determine how context affects accuracy. Third, we evaluate how often LLMs misclassify vulnerabilities, either by labeling secure code as vulnerable (false positives) or by failing to detect actual vulnerabilities (false negatives).

To improve detection reliability, we explore fine-tuning LLMs on multi-language datasets and evaluate whether this approach reduces errors. We also employ prompt engineering techniques, such as chain-of-thought reasoning and few-shot learning, to enhance the model's ability to reason through complex code structures and recognize subtle vulnerability patterns. By systematically evaluating these methods, this work aims to provide a clearer understanding of LLMs' strengths and limitations in vulnerability detection, offering practical insights for improving automated security analysis in real-world development environments.

1.3.1 Research Questions

This study aims to systematically evaluate the capabilities of LLMs in detecting code vulnerabilities across three critical dimensions: programming language adaptability, analysis granularity, and classification accuracy. We investigate how fine-tuning can enhance detection performance while reducing critical errors in real-world vulnerability assessment scenarios.

The research is guided by three focused questions:

- RQ1: How effectively can LLMs detect vulnerabilities across different programming languages?
- RQ2: How does detection performance vary between file-level and function-level analysis?
- RQ3: Does fine-tuning reduce false-positive and false-negative rates in multi-class vulnerability classification?

We begin by conducting a literature review to assess the current state of LLMs in code vulnerability detection, identifying which models, environments, and prompting techniques produce the best results. Based on the insights gained from this review, we design experiments aimed at improving LLM performance in this area.

Our experimental study uses the CVEFixes dataset [10], which contains code samples from multiple programming languages, annotated with various Common Weakness Enumeration (CWE) classes and detailed descriptions. We split the dataset into training and test sets to enable proper model evaluation.

First, we evaluate the performance of unmodified, pre-trained LLMs using three types of prompts: zero-shot, few-shot, and chain-of-thought. These prompts guide the model in analyzing code, detecting vulnerabilities, and classifying them into the appropriate CWE categories.

To improve performance, we fine-tuned the LLMs on the training portion of the dataset to adapt them specifically to the task of code vulnerability detection. Then, we re-evaluate the fine-tuned models on the test set using the same prompt styles.

Finally, we conduct a statistical analysis of the results and compare the performance of the original and fine-tuned models using various evaluation metrics.

1.3.2 Novelty & Contribution

Our master thesis focuses on the following points.

- **Impact of Code Context Granularity:** We compare LLM effectiveness at different granularities to determine whether broader code contexts improve vulnerability identification compared to function-level analysis.
- **Multilingual Dataset Expansion:** We expand the scope of language-specific vulnerability detection by evaluating LLMs across 6 programming languages. This multilingual assessment reveals how detection capabilities vary across language paradigms and identifies opportunities for improving cross-linguistic generalization.
- **Enhanced Distinction Between Similar Vulnerabilities:** We provide a systematic evaluation of how fine-tuning affects classification accuracy, with particular focus on reducing false positives and false negatives in multi-class vulnerability detection.

2. Literature Review

This literature review aims to explore existing studies, methodologies, and applications of LLMs in code vulnerability detection. By examining recent advances and identifying gaps in current research, we can gain valuable insights that can guide future developments in this critical area.

2.1 Literature Review Questions

To explore the current landscape of LLMs in code vulnerability detection, this literature review is structured with main 3 research questions:

RQ1: What is the capability of LLMs in code vulnerability detection?

This question examines how effectively LLMs can identify vulnerabilities in code, assessing their accuracy, reliability, and the types of vulnerabilities they can detect.

RQ2: What methods, techniques, and algorithms are used to optimize performance?

This question explores the various strategies employed to enhance the performance of LLMs, including specific algorithms, training techniques, and optimization methods used in the studies.

RQ3: What is lacking in the current state of LLMs in code vulnerability detection?

We identify the limitations and gaps in existing LLMs when applied to vulnerability detection, highlighting areas where they fall short or require improvement.

By addressing these questions, this review aims to provide a comprehensive understanding of how LLMs are currently utilized in code vulnerability detection, the challenges faced, and the advancements made to optimize their performance. This will not only highlight the state of the art but also identify opportunities for future research and development in this critical field.

2.2 Systematic Literature Review

This systematic literature review aims to identify existing academic literature related to the posed research questions on LLMs in code vulnerability detection. By examining the discovered papers, the review seeks to determine the current state of the art in this field and to identify existing gaps in knowledge.

2.2.1 Search Strategy

The search for relevant academic literature was conducted over multiple databases to ensure a wide coverage of available research. The databases consulted include the IEEE Digital Library, Scopus, and the ACM Digital Library. This approach was designed to compile a comprehensive collection of relevant literature.

2.2.2 Search Terms

To effectively locate studies pertinent to the research questions, specific search terms were employed. The search string used was:

> ("Large Language Models" OR "LLMs") AND ("Vulnerability Detection" OR "Code Security" OR "Security Code Review" OR "Secure Code Analysis" OR "Code Analysis" OR "Software Security" OR "Code Vulnerabilities")

This search string was applied to either the titles or abstracts of publications within the selected databases. The combination of keywords was carefully chosen to include various aspects of code vulnerability detection using LLMs.

2.2.3 Inclusion and Exclusion Criteria

The selection of literature was guided by specific inclusion and exclusion criteria to follow relevance and quality:

- **Relevance to Research Purpose:** Only studies directly related to LLMs in code vulnerability detection were included to maintain focus on the research objectives.

- **Publication Date:** Research published from 2019 onwards was considered. This time frame was selected because it marks the release of Google's BERT model, which is a major milestone that brought LLMs to the mainstream in the field.
- **Language:** Only literature written in English was included to ensure accurate comprehension and analysis.

2.2.4 Search Result

Source	Total Papers	After Filtering
IEEE	25	11
Scopus	69	9
ACM Digital Library	22	0

Table 1. Paper Counts Before and After Filtering

This table shows search results from various databases using specific search terms. The ACM Digital Library shows 0 results after filtering because the papers are either covered by Scopus or considered irrelevant.

2.3 Literature Review

I selected research papers that address the study's key questions, focusing on their contributions to methodology, algorithms, or findings.

2.3.1 Prompt Related Techniques

Zero-shot Approaches in LLM-based Vulnerability Detection

This paper, *"Software Vulnerability Detection using Large Language Models"*[5], explores the potential of LLMs for software vulnerability detection, with a particular focus on zero-shot (simple prompt-based) and supervised fine-tuned approaches. The study evaluates several LLMs, including GPT-3.5-Turbo and Davinci, for detecting two key types of vulnerabilities: SQL Injection and Buffer Overflow. Using two datasets, code gadgets and CVEfixes, the authors examine the performance of the models in real-world-like scenarios.

The zero-shot, prompt-based LLMs showed limited effectiveness with low recall and high false-positive rates, indicating that simple, non-fine-tuned use is not sufficient for reliable vulnerability detection. However, fine-tuning on labeled datasets significantly improved model performance, particularly for the Davinci model, which outperformed other models in detecting patterns associated with buffer overflow vulnerabilities.

Despite the promise of fine-tuned LLMs, the study highlights a major limitation: the models often misidentify non-vulnerable code as vulnerable due to high false positives. While fine-tuning improved recall rates, the inability of LLMs to fully understand data flow or pinpoint the exact location of vulnerabilities limits their accuracy, particularly in cases where code contains minor modifications or validation mechanisms.

Chain-of-Thought (CoT) and Tree-of-Thought (ToT) Reasoning in Vulnerability Detection

In recent work, researchers have studied the effectiveness of CoT and ToT reasoning strategies for improving vulnerability detection using LLMs. The study, *"Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study"*[11], benchmarks CoT and ToT techniques for detecting vulnerabilities in source code, comparing LLMs such as GPT-4 and Claude 3 Opus to traditional static analysis tools. Using CoT, models are guided to analyze code in sequential steps, simulating the structured approach a human reviewer might take. ToT reasoning builds on this by generating and evaluating multiple reasoning paths for each analysis step, allowing the model to select the most reliable answers. This combination resulted in high recall and F1 scores, often surpassing conventional tools such as CodeQL and SpotBugs. The authors concluded that while CoT and ToT prompts improved LLM performance for comprehensive security audits.

Similarly, the paper *"Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives"*[9] applies CoT and ToT within a framework called GPTLENS for detecting vulnerabilities in smart contracts. GPTLENS divides the analysis into two stages, with the LLM assuming the roles of Auditor and Critic to iteratively refine its findings. In the initial generation stage, the CoT and ToT allow the model to generate different vulnerability hypotheses, improving the depth of potential problem coverage, but with a slight increase in false positives. In the subsequent discrimination stage, the critic applies these reasoning steps to evaluate each hypothesis, reducing false positives without compromising detection accuracy. Their results show that GPTLENS significantly improves detection rates, identifying vulnerabilities 76.9% of the time compared to 38.5% for traditional methods.

Together, these studies demonstrate the value of prompting techniques in using LLMs for the security tasks, with CoT promoting structured analysis and ToT adding depth through iterative reasoning, marking an important evolution in LLMs oriented detection in both traditional software and smart contract environments.

Effectiveness of Prompt-Tuned LLMs in Vulnerability Detection

The paper *"PSCVFinder: A Prompt-Tuning Based Framework for Smart Contract Vulnerability Detection"*[12] presents PSCVFinder, a framework for detecting vulnerabilities in smart contracts that focuses on addressing the limitations of traditional methods and previous deep learning models. Traditional approaches, such as symbolic execution and static analysis, often struggle with adaptability and accuracy due to the rigid rules and irrelevant code content in smart contracts. PSCVFinder leverages prompt-tuning on the CodeT5 model to bridge the gap between pre-training and downstream tasks, specifically detecting reentrancy and timestamp dependency vulnerabilities in Ethereum contracts. By structuring inputs as prompts that align with the model's pre-training goal, prompt-tuning effectively improves PSCVFinder's precision and recall. In addition, a slicing technique called CSCV (Crucial Smart Contract for Vulnerabilities) is used to remove irrelevant code, allowing the model to focus on vulnerability-related segments.

The effectiveness of prompt-tuning in PSCVFinder is demonstrated through experiments on over 200,000 real-world smart contracts, where it achieved state-of-the-art results, outperforming deep learning approaches. The framework achieved F1 scores of 93.83% for reentrancy and 93.49% for timestamp dependency vulnerabilities, showing significant improvements in precision and recall compared to baseline methods. Prompt-tuning proves critical in leveraging the knowledge gained from pre-training, allowing the model to more accurately apply pre-trained language representations to vulnerability detection tasks.

2.3.2 Effectiveness of Supervised Fine-Tuned LLMs

Similar to the paper *"Software Vulnerability Detection using Large Language Models"*[5], which demonstrates that supervised fine-tuning improves the false positive rate, fine-tuning continues to prove essential for improving model accuracy and reducing false positives in vulnerability detection.

This paper *"Multitask-based Evaluation of Open-Source LLM on Software Vulnerability"*[6] investigates the effectiveness of fine-tuning in improving model accuracy and reducing false-positive rates for software vulnerability detection. It examines several open-source LLMs, such as CodeLlama and DeepSeek-Coder, and finds that while LLMs perform less effective in vulnerability assessment in binary (Non-vulnerable or vulnerable), fine-tuned on domain-specific datasets such as Big-Vul show improvements, particularly in false positive rates. These results confirm that fine-tuning to specific vulnerability data can be useful to refining model outputs for improved accuracy and reduced false aleart.

2.3.3 Artillery Information in Prompt

Static Analysis Augmentation for Enriching Prompts

In *"Software Vulnerability Detection with GPT and In-Context Learning"*[13], the authors present VUL-GPT, a vulnerability detection model that uses GPT-3.5 and static analysis information to enrich prompt information for identifying software vulnerabilities. Unlike models that rely solely on deep learning, VUL-GPT incorporates code retrieval methods such as TF-IDF and BM-25 to locate similar vulnerable code snippets in the database, which are used together as a static analysis result to generate result. This approach allows the model to focus on relevant vulnerability patterns, informed by structural and semantic details identified in the static analysis phase. In evaluations using the Devign dataset, VUL-GPT significantly outperformed other models such as VulDeePecker and SySeVR, achieving F1 score of 56.45% with TF-IDF, underscoring the effectiveness of static analysis in improving LLM-based vulnerability detection.

Similarly, *"GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis"*[14], another hybrid tool, combines GPT with static analysis to detect logical vulnerabilities in smart contracts, especially those often missed by pattern-based methods. GPTScan uses static analysis tools such as ANTLR and crytic-compiler to parse and analyze the code's control flow and data dependencies, refining the model's focus to accurately match relevant scenarios and properties. This combination allows GPTScan to maintain high accuracy in simpler token contracts and acceptable accuracy in more complex contexts. In testing with over 400 smart contract projects, GPTScan achieved low false positive rates, showing the utility of static analysis in augmenting LLM-based vulnerability assessment.

The paper *"A New Approach to Web Application Security: Utilizing GPT Language Models for Source Code Inspection"*[7] explores how LLMs such as GPT-3.5 and GPT-4 can detect web application vulnerabilities, specifically the CWE-653 sensitive data isolation vulnerability in Angular-based projects. This approach uses prompt engineering techniques, including few-shot examples and chain-of-thought prompting, to improve the interpretation capabilities of LLMs. To further improve the detection, the authors introduce a JSON mapping technique that structures essential components, such as services and AuthGuards, for the LLM. This structured JSON input provides targeted context for data flows and protection mechanisms, improving the model's ability to detect security risks across application components. Using this method, GPT-4 achieved an 88.76% success rate in detecting vulnerabilities, although model limitations such as prompt sensitivity and coding practice dependencies were noted, especially with GPT-3.5.

Overall, these studies show that integrating artillery information into prompt can improve the effectiveness of LLMs in identifying software vulnerabilities.

Graph Data as Artillery Information

Recent studies show that integrating graph-based structures into prompts improves LLM's vulnerability detection performance. *"GRACE: Empowering LLM-based Software Vulnerability Detection with Graph Structure and In-Context Learning"*[15] uses abstract syntax trees (ASTs), control flow graphs (CFGs), and program dependency graphs (PDGs) to augment prompts, improving LLM's analysis of code relationships and data flows. This approach improves detection accuracy, especially for vulnerabilities with complex dependencies.

Similarly, *"SCALE: Constructing Structured Natural Language Comment Trees for Software Vulnerability Detection"*[16] augments ASTs with structured natural language comments, creating a hybrid representation that combines execution logic with semantic context. This method yields a 13.47% improvement in F1 score by helping LLMs identify vulnerabilities missed by token-based approaches.

Other advances combine graph data with CoT reasoning. *"Exploration on Prompting LLM with Code-Specific Information for Vulnerability Detection"*[17] integrates data flow graphs (DFGs) and CoT to break down vulnerability analysis into structured reasoning steps, while *"Prompt-Enhanced Software Vulnerability Detection Using ChatGPT"*[18] pairs DFGs with API call sequences. *"Generalization-Enhanced Code Vulnerability Detection via Multi-Task Instruction Fine-Tuning"*[19] introduces VulLLM, which uses PDGs with CoT and Self Verification (CoT-SV) to improve both detection and interpretability, achieving an 8% F1 score gain.

2.4 Analysis

2.4.1 RQ1: What is the capability of LLMs in code vulnerability detection?

LLMs have proven to be powerful tools for code vulnerability detection, offering precise detection and detailed explanation in security code analysis. These advanced LLMs can classify code in binary (vulnerable/non-vulnerable) and multi-class, identifying specific vulnerability types. In addition to simple classification, LLMs provide descriptive explanations of why a particular code segment is vulnerable, offering insight into potential security risks by pinpointing the exact lines or blocks of code that introduce vulnerabilities.

2.4.2 RQ2: What methods, techniques, and algorithms are used to optimize performance?

LLMs have demonstrated significant potential for accurate detection of code vulnerabilities. Their capabilities are enhanced when integrated with advanced techniques such as prompt tuning (CoT reasoning and ToT prompting), and fine-tuning. These methods allow LLMs to analyze code in a more nuanced way, taking into account context and logical flow, which improves their accuracy and reliability in detecting vulnerabilities.

The use of graph data structures allows LLMs to model code as graphs, capturing relationships and dependencies between code components. This innovative approach facilitates the detection of complex vulnerabilities that might be missed by linear code analysis. By understanding the interconnectedness within code, LLMs can identify vulnerabilities that involve intricate interactions between different parts of the code base.

Hybrid models that combine LLMs with code analysis tools further enhance detection capabilities as well. These models leverage the strengths of AI-driven analysis and established program analysis methods, resulting in higher accuracy and the ability to detect a broader range of vulnerabilities. The combination of LLMs and static analysis tools provides better result for vulnerability detection.

2.4.3 RQ3: What is lacking in the current state of LLMs in code vulnerability detection?

Despite these improvements, current applications of LLMs for code vulnerability detection have some notable limitations. A key problem is the limited variety of datasets used for training and evaluation. Most studies rely on datasets that include only one or two programming languages, limiting the generalizability of the models to other languages such as Java, Python, and JavaScript. Addressing this gap by expanding the diversity of datasets and providing performance comparisons across languages can be benefitial to evaluating the broader applicability of these models in diverse programming environments.

Another area for improvement is the granularity of vulnerability detection. Current studies focus primarily on function-level analysis, potentially missing vulnerabilities at the statement or file level. Evaluating the performance of LLMs in finer-grained contexts, particularly file-level detection, would enable more precise identification and mitigation of security vulnerabilities.

In addition, LLMs face challenges in distinguishing between similar vulnerability types or vulnerable/non-vulnerable, leading to misclassifications. Overlapping vulnerability types can confuse models, reducing the reliability of detection results. Addressing this issue requires more nuanced modeling techniques and possibly more detailed labeling in datasets to help models learn the subtle differences between similar vulnerabilities.

2.5 Conclusion

In conclusion, this literature review establishes a foundation for my master's thesis by analyzing the current capabilities and limitations of LLMs in code vulnerability detection. The findings gathered here will be used to construct a detailed research methodology and framework that will guide subsequent work to advance understanding in this area. Moving forward, my focus will be on designing a structured approach that addresses the identified challenges and sets the stage for a comprehensive exploration of LLMs in enhancing code security.

3. Theoretical Backgrounds

LLMs demonstrate a notable ability to generate creative solutions, reason through complex problems, and explain the logic behind their findings. Their ability to generate patterns from diverse training data allows them to adapt to tasks ranging from abstract ideation to structured technical analysis. A key strength of LLMs is their ability to justify their reasoning, which is a critical trait in fields such as cybersecurity, where transparent decision-making is essential. However, while LLMs demonstrate broad expertise, their generic training can lack their precision in specialized tasks such as code vulnerability detection. This gap motivates the exploration of targeted improvement strategies, including prompt engineering and fine-tuning, to align LLM outputs with domain-specific requirements.

The formulation of input prompts plays a critical role in shaping how LLMs process and generate responses to complex tasks. By refining prompts to guide the model's reasoning framework, it is possible to enhance its capacity for structured problem solving and domain-specific insight. For example, prompts that encourage logical breakdowns of tasks can improve a model's ability to systematically analyze security vulnerabilities. This approach mimics human analytical workflows by requiring the model to break down code evaluation into sequential steps, such as tracing execution paths, validating input sanitation, or prioritizing risks based on potential vulnerabilities. Such structured reasoning not only reduces ambiguity in results, but also facilitates contextual understanding, allowing the model to identify security flaws that might avoid detection.

Prompt engineering shapes how LLMs generate responses, while fine-tuning adjusts their underlying knowledge to detect code vulnerabilities. Pre-trained models may lack exposure to the technical details of secure coding practices, leading to hallucinations such as classifying as irrelevant vulnerabilities (false positives). Fine-tuning with datasets containing labeled examples of code vulnerabilities could address this limitation.

This section details the technical foundations of prompt engineering and fine-tuning, outlining their roles in adapting LLMs for precise and reliable code vulnerability detection.

3.1 Prompt Engineering

Prompt engineering is a critical aspect of optimizing the performance of LLMs. The choice of prompting technique has a strong impact on the quality of the model's output, its accuracy, and its reasoning capabilities in various tasks. This study uses two different prompt styles to assess their comparative strengths, based on existing literature demonstrating their effectiveness.

3.1.1 Chain of Thought (CoT) Prompting

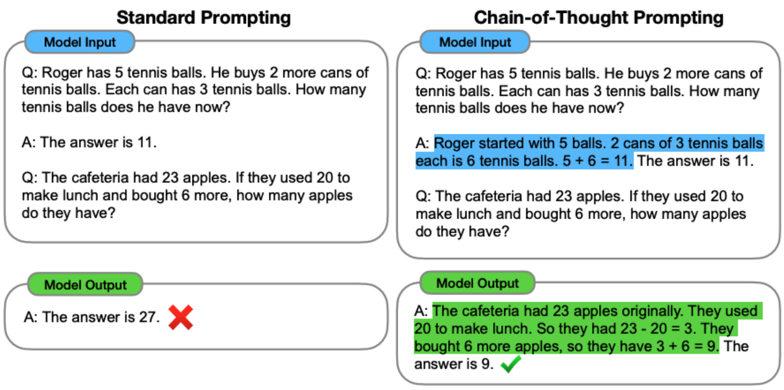

Figure 1. Example of CoT reasoning steps (reference from [20]).

Chain-of-Thought (CoT) is a prompting technique that improve the reasoning capabilities of LLMs by breaking down problems into intermediate steps before reaching a final answer. Recent studies show that CoT significantly improves performance on mathematical, common sense, and symbolic reasoning tasks, especially in complex scenarios where standard prompting fails. [20, 21] In particular cases, CoT's effectiveness scales with model size: larger LLMs, when given step-by-step demonstrations, can decompose problems more effectively without the need for fine-tuning. However, the benefits of CoT are not universal. Research indicates that its advantages are primarily confined to mathematical and logical reasoning tasks [21]. In other domains, such as commonsense reasoning and factual knowledge retrieval, CoT may yield marginal improvements or, in some cases, degrade model performance compared to simpler prompting approaches.

3.1.2 Few-Shots (FS) Prompting

Few-shot prompting takes advantage of context by providing the model with example demonstrations within the prompt. These examples help providing the model to generate more accurate responses to subsequent queries. Studies, such as those using the LLaMA architecture, confirm that increasing the number of in-context examples improves the accuracy of the output [22].

3.1.3 Zero-Shot Prompting

Zero-shot prompting evaluates a model's ability to perform a task without prior examples, relying solely on its pre-trained knowledge. Unlike few-shot or CoT prompting, it requires no task-specific demonstrations, making it a baseline for assessing inherent generalization capabilities. While efficient for straightforward tasks (e.g., classification or translation), its performance often lags behind few-shot or reasoning-based methods in complex domains [23]. For instance, zero-shot prompting may suffice for simple factual queries but struggles with multi-step reasoning, where CoT or few-shot methods excel [20]. This limitation highlights the trade-off between simplicity and accuracy in prompt design.

3.2 Fine-tuning

Fine-tuning, as introduced earlier, refers to the process of adding further training to a pre-trained model using new domain-specific data. The initial phase of training, where the model learns general language patterns and structures from a large and diverse dataset, is known as "pre-training". Fine-tuning, on the other hand, involves adapting the pre-trained model to a specific task or domain by training it on a smaller and more specialized dataset. During this process, the model's parameters are adjusted to different weights according to the characteristics of the new data, making it capable of capturing patterns from the dataset. This additional training allows the model to refine its knowledge and improve its performance in specific domains, making it more accurate and relevant.

3.2.1 Parameter-efficient Fine-tuning (PEFT)

Fine-tuning AI models typically requires expensive resources (storage and GPU with large amount of VRAM), especially for LLMs due to their memory size and large number of parameters. However, PEFT allows training on new data with significantly less resources. PEFT achieves this by fine-tuning only a small subset of additional model parameters while freezing the majority of the pre-trained LLM parameters. This approach not only reduces computational and memory costs, but also addresses the issue of forgetting knowledge, a phenomenon observed during full fine-tuning where the model loses previously learned knowledge.

Quantized Low-Rank Adaptation(QLoRa)

There are various ways of achieving PEFT. QLoRA is most widely used and effective method. In this method, a set of parameters are modularly added to the network, with lower dimensional space. Instead of modifying the whole network, only these modular low-rank network is modified, to achieve the effect. [24]

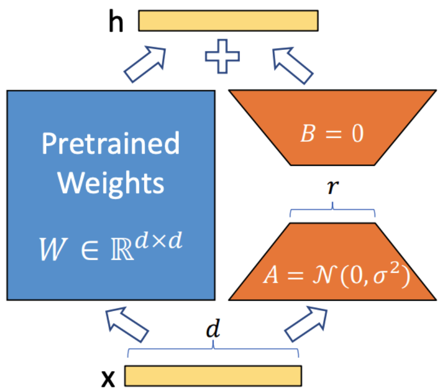

Figure 2. How LoRa works from the original paper [24]

In figure 2, the introduction of an additional layer of trainable parameters (represented as weights on the right) while keeping the original parameters frozen. These trainable parameters are constructed with a significantly reduced rank compared to the dimensions of the original network, allowing for efficient adaptation without extensive modification of the base model.

3.2.2 Instruction Tuning

Instruction tuning is a fine-tuning method that significantly improves the zero-shot learning ability of LLMs by training them to follow natural language instructions across diverse tasks. During instruction tuning, the model learns from structured examples consisting of:

- **Instruction sentences**: natural language descriptions specifying the task (e.g., "Classify the following news headline..."),
- **Input sentences**: the input data to the task, and
- **Output sentences**: the expected correct response.

For example:

- **Instruction Sentence:** *"Classify the following news headline as 'politics,' 'sports,' or 'entertainment."'*
- **Input Sentence:** *"In yesterday's match, the Japanese national soccer team's player XX scored a hat trick."*
- **Output Sentence:** *"Sports"*

The core advantage of instruction tuning is that it explicitly defines each task using natural language. This makes it easier for the model to generalize to unseen tasks during zero-shot inference. This is achieved by fine-tuning on a diverse set of tasks described by instructions. This allows the model to learn to interpret and follow natural language prompts rather than relying on examples or task-specific architectures.

As demonstrated by the Google's FLAN model [25], instruction tuning on a wide variety of natural language processing tasks substantially improves zero-shot performance on tasks the model has never seen during training. This makes the model more flexible and useful across a wide range of applications.

4. Methodology

This study systematically evaluates the capability of LLMs to detect file-level and function-level vulnerabilities in multiple programming languages through two phase exploration. In the first phase, we examines the performance of the model using various prompt engineering techniques, followed by a statistical analysis of the results. Specifically, we examine how results vary across programming languages and vulnerability classifications, including non-vulnerable code and different vulnerability types. In the second phase, we evaluate the effectiveness of fine-tuning by measuring its impact on reducing false-positive/negative rates in the model's outputs.

4.1 Experiment Environment

4.1.1 Model Selection

For this study, we selected **Microsoft Phi-4** [26] as our main LLM. Our decision was based on three criteria: model performance, adaptability to fine-tuning and parameter optimization, and compatibility with our computational in

Phi-4 is a 14-billion-parameter language model developed by Microsoft with a parameter-efficient architecture that achieves state-of-the-art performance despite its relatively small parameter size. Phi-4 shows strong reasoning capabilities and outperforms larger LLMs such as LLaMA-3.3-70B, on coding-related benchmarks. Its computational efficiency enables deployment within our hardware constraints, making it suitable for our experimental setup.

Vulnerability classification requires an understanding of code semantics, logical flow, and the identification of potentially insecure code patterns. Phi-4's training process emphasizes reasoning and uses high-quality synthetic datasets designed to develop structured problem-solving skills.

4.1.2 Hardware Environment

The following hardware configuration was used across all experiments:

- **CPU**: AMD EPYC 7443 (48) @ 2.844GHz

- **GPU**: NVIDIA A100 PCIe 40GB
- **RAM**: 241600 MB (236 GB)
- **CUDA Version**: 12.2

This hardware configuration provided sufficient computational capacity for both model inference and fine-tuning operations, with the A100 GPU offering the necessary memory and processing capabilities for efficient training and evaluation.

4.1.3 Software Environment

List of implemented software stack are:

- **Operating System**: Ubuntu 22.04.5 LTS
- **Programming Language**: Python 3.10.12
- **Frameworks and Libraries**:
  - Ollama: For model deployment and inference
  - Hugging Face Transformers: For model loading and tokenization
  - Unsloth: For efficient fine-tuning optimization
  - PyTorch (2.5.1+cu124): Deep learning framework
  - Jupyter Notebook: For interactive development and analysis

4.2 Dataset: CVEfixes

CVEfixes [10] is a dataset that collected software vulnerabilities in open source projects. This dataset is useful especially for training and evaluating LLMs. The dataset links Common Vulnerabilities and Exposures (CVE) entries to the corresponding code changes that fix them. It captures full file versions before and after fixes, as well as isolated functions, providing LLMs with the ability to analyze vulnerability contexts in granular detail. In addition, metadata such as commit hashes, affected file paths, and categorized vulnerability types come with the dataset. By providing connections between vulnerabilities and their fixes, CVEfixes not only supports model training, but also provides assessments of LLM performance. This combination of file level granularity and contextual metadata makes the dataset uniquely suited for our study.

The dataset contains 12,107 vulnerability-fixing commits from 4,249 open-source projects, covering 11,873 CVEs and 272 distinct CWE types. Figures 3 and 4 present the original CWE distribution and programming language representation of files, respectively, as extracted from the unmodified CVEfixes dataset.

Figure 5 displays the distribution of lines of code per file. Most files contain around 250 lines, and this serve as a filtering criterion.

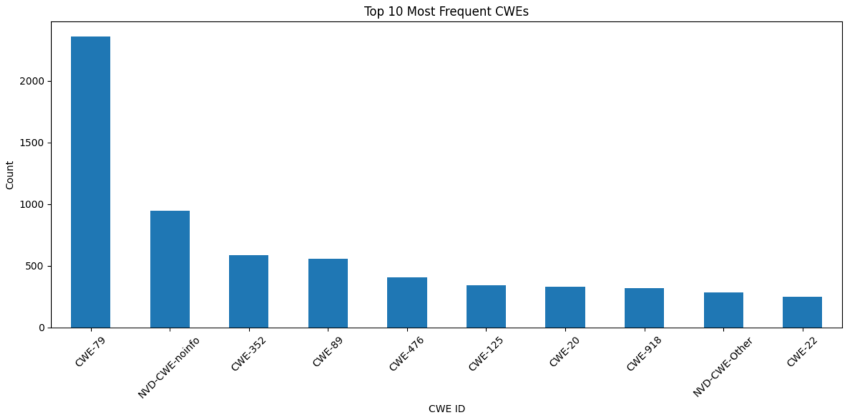

Figure 3. Distribution of CWE Files Numbers

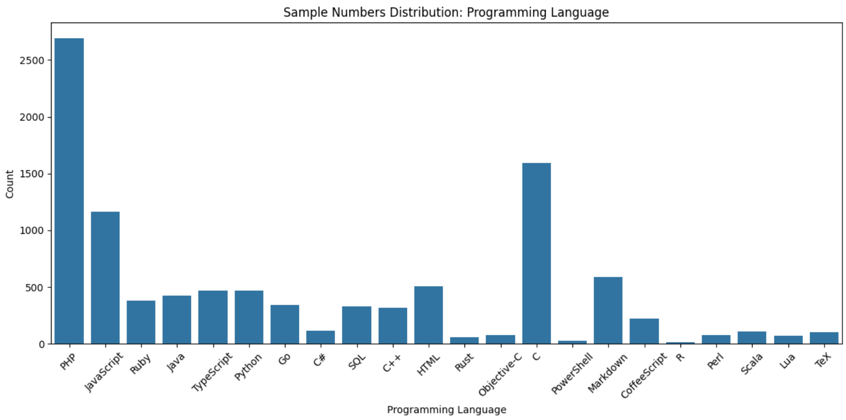

Figure 4. Distribution of Sample Numbers in each Programming Language

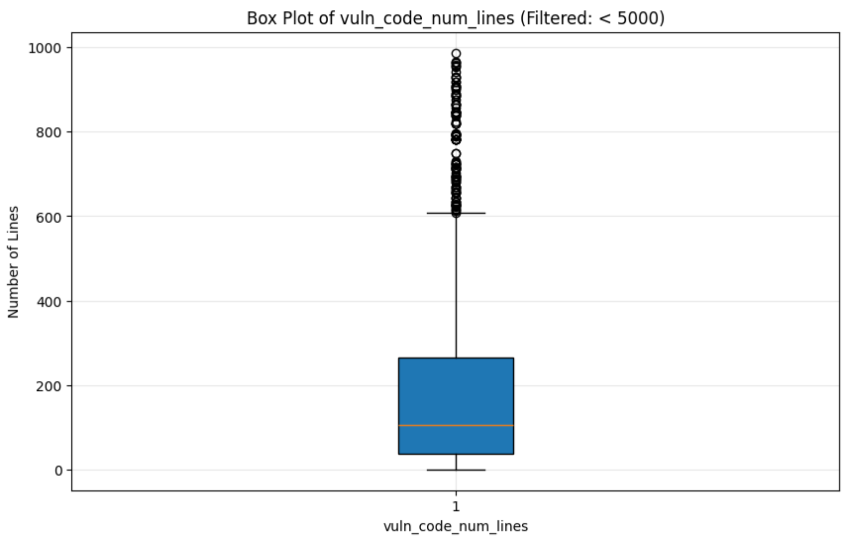

Figure 5. Distribution of code lines

4.3 Dataset Pre-processing

4.3.1 Dataset for File Level Vulnerability Detection

Data Splitting and Filtering Criteria

After analyzing the distribution of samples by programming language and CWE ID, the dataset was filtered and split into training and test subsets using the following criteria:

- Programming Languages: Only examples written in one of six languages were retained: PHP, JavaScript, Java, TypeScript, Ruby, or Python.
- Code Length: Examples exceeding 150 lines or 3,000 tokens have been discarded to ensure compatibility with standard LLM context windows and to keep processing efficient.
- Patch Validity: Patches with fewer than two lines added or deleted have been excluded to make sure that the changes between vulnerable and fixed code are meaningful.

Due to a significant class imbalance in the CWE categories, we focused on four common vulnerability types:

- **CWE-79**: Improper Neutralization of Input During Web Page Generation (XSS).
- **CWE-89**: Incorrect Neutralization of Special Elements in SQL Commands.
- **CWE-434**: Unrestricted upload of malicious file types.
- **CWE-352**: Cross-site request forgery (CSRF).

These categories were selected based on their critical impact on web security and adequate sample sizes across languages.

Dataset Filtering Methodology

The dataset was split using a stratified approach to ensure balanced representation across programming languages and vulnerability types. For vulnerable samples, the stratified_split function grouped the data by both programming language and CWE-ID, then split each group into training (90%) and test (10%) sets while maintaining the original distribution.

Non-vulnerable samples were constructed from patched code segments. The amount of samples matched the number of vulnerable samples (1:1 ratio). Figures 6 and 7 show the distribution of class labels and programming languages, respectively. As can be seen from the figures, both the labels and the languages show a class imbalance.

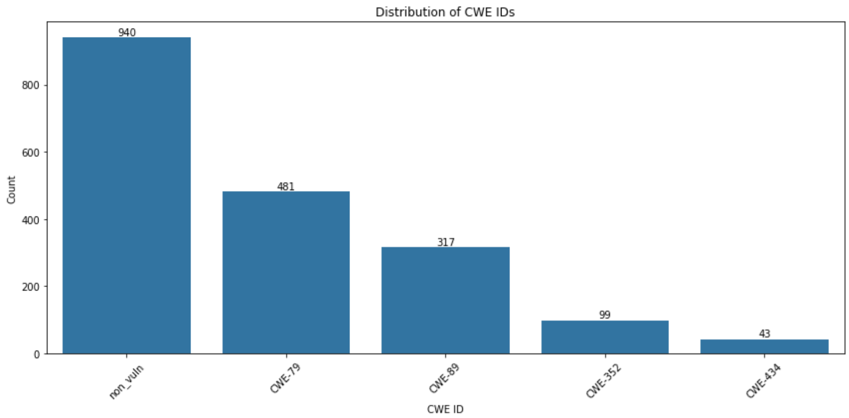

Figure 6. Label Distribution: Train Dataset

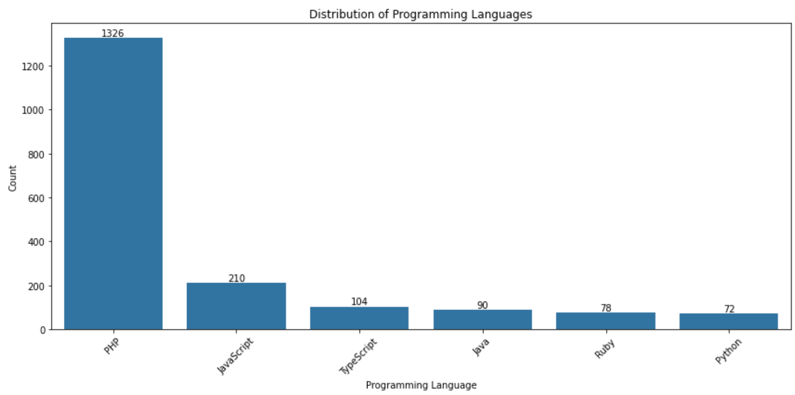

Figure 7. Language Distribution: Train Dataset

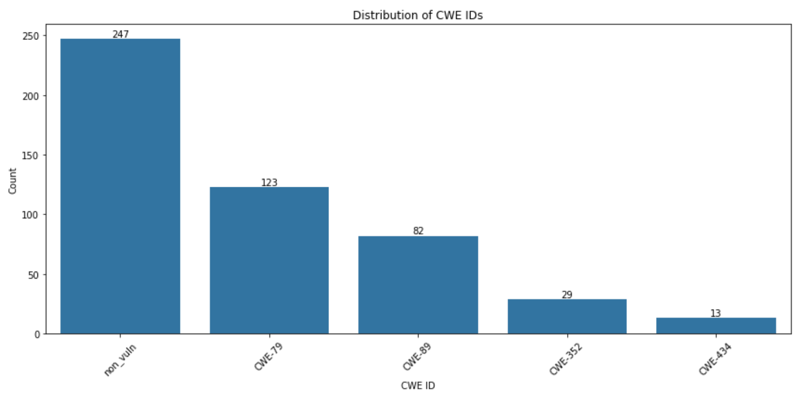

Figure 8. Label Distribution: Test Dataset

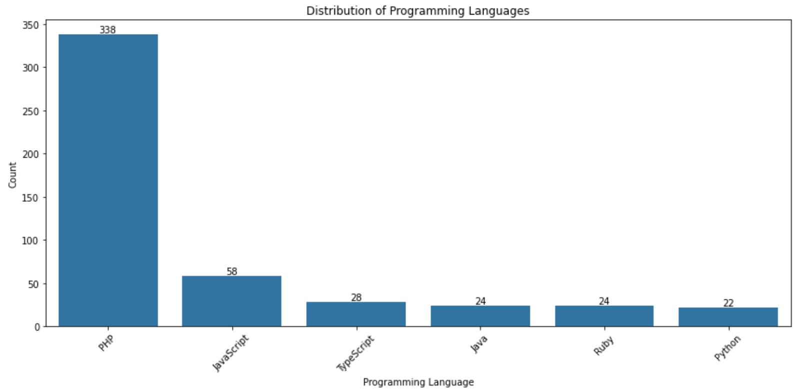

Figure 9. Language Distribution: Test Dataset

4.3.2 Dataset for Function-level Vulnerability Detection

In addition to the file-level classification, we conducted experiments on function-level multi-class vulnerability classification. The extraction criteria remain the same as those used at the file level, maintaining consistency across both granularity. Our dataset includes the same six programming languages and four vulnerability types discussed previously.

Figure 10 shows the distribution of CWE types across our function-level samples. While the dataset contains a substantial number of samples overall, we observe a significant class imbalance among different vulnerability types.

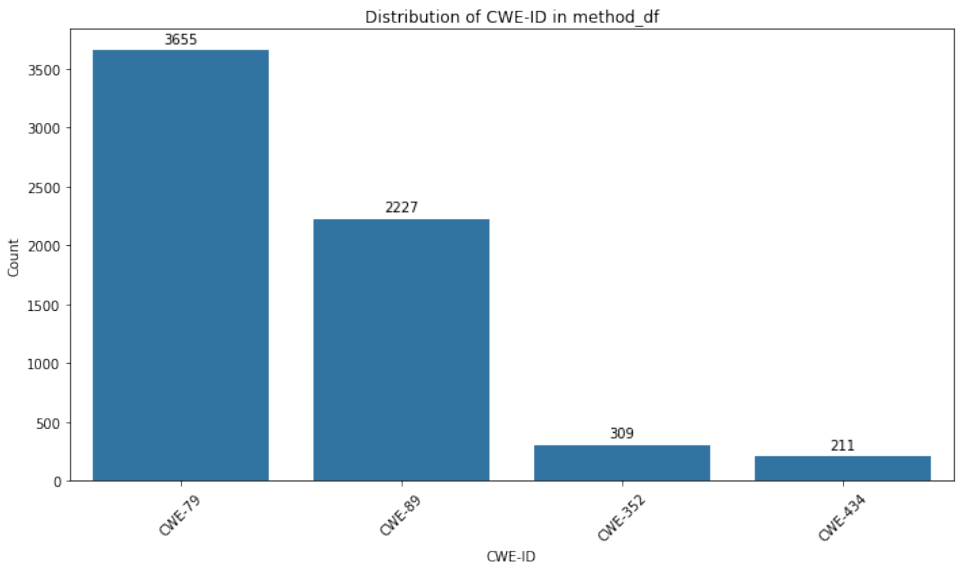

Figure 10. CWE Distribution in Vulnerable Functions

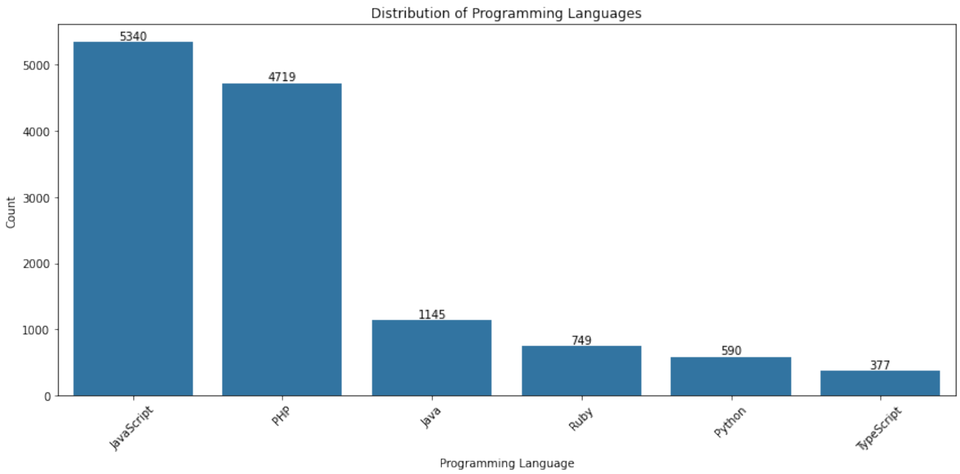

Figure 11. Language Distribution in Vulnerable Functions

Dataset Filtering Methodology

The original dataset was filtered and split using stratified sampling to keep the distribution of languages and vulnerabilities. JavaScript samples were reduced to 5% per vulnerability type, while PHP non-vulnerable code retained only 10% of its original entries.

For both vulnerable and non-vulnerable samples, the data was split into training (90%) and test (10%) sets by grouping by programming language and CWE-ID.

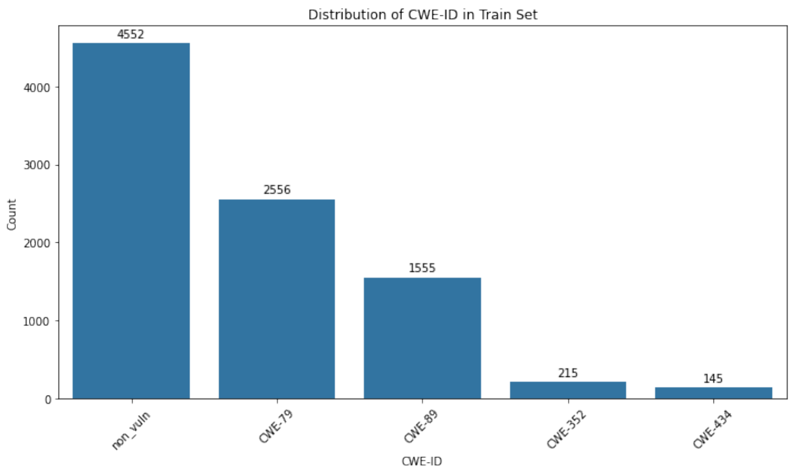

Figure 12. Function-level Label Distribution: Train Dataset

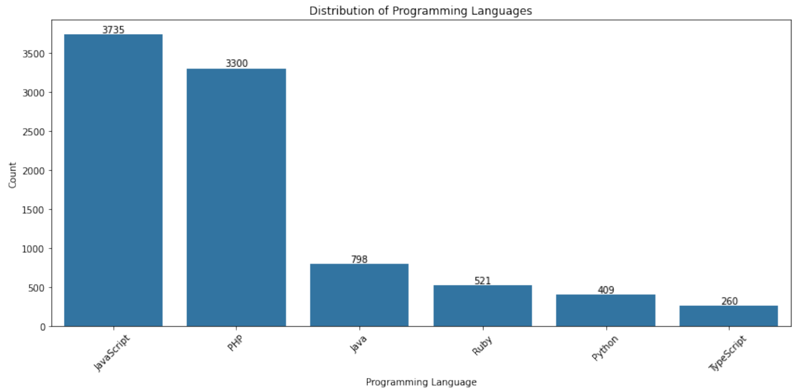

Figure 13. Function-level Language Distribution: Train Dataset

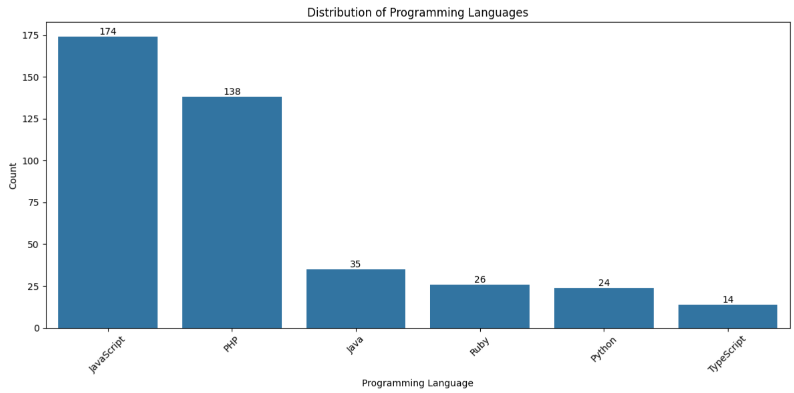

Figure 14. Function-level Label Distribution: Test Dataset

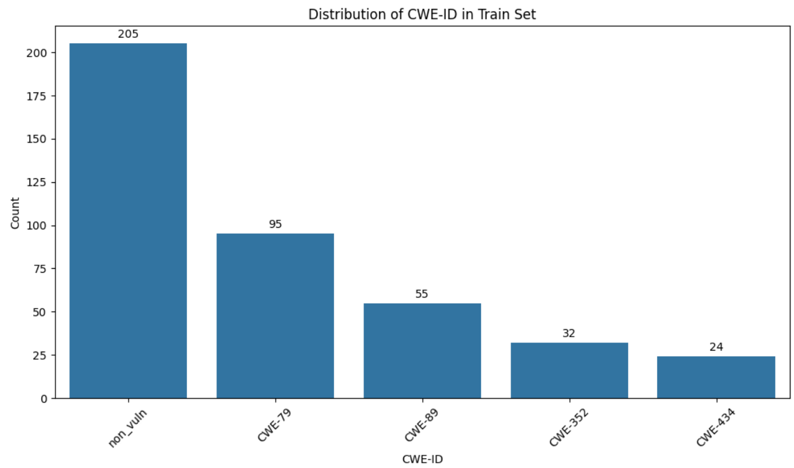

Figure 15. Function-level Language Distribution: Test Dataset

4.4 Baseline Testing

The baseline evaluation begins with unmodified versions of the target language models. The test data set is formatted using three different prompting strategies: CoT prompting, Zero-shot prompting, and Few-Shot prompting. Each prompt includes a set of output class options. To standardize the outputs, all models are instructed to generate responses in JSON format only. This study [27] suggest that presenting the reasoning before the final decision leads to more consistent results. Therefore, each model response is required to include both a reasoning section and the predicted class label.

After experimentation, the model performance is evaluated using standard classification metrics: accuracy, precision, recall, and F1 score. For vulnerability type classification, although the class options are provided in the prompts, any model output that includes a more specific CWE subclass is mapped to its corresponding parent category. A prediction is considered correct if the resolved parent class matches the ground truth annotation in the test dataset. This hierarchical alignment ensures consistent scoring across varying levels of CWE specificity.

Figure 16. Chain-of-Thought Prompt (CoT)

```
You are a security expert tasked with identifying vulnerabilities in a given code.
Carefully analyze the code using CWE descriptions and determine if it contains any
vulnerabilities step by step.

For each step:
1. Examine overall structure to understand purpose and functionality
2. Assess User Input Handling & Data Flow. Determine input processing and track data
flow
3. Analyze for vulnerabilities:
   - If vulnerable: Proceed to steps 4-7
   - If safe: Output must be strictly:
     {
       "reasoning": "Detailed explanation...",
       "cwe": null
     }
4. Identify exploitable function name/code snippet
5. Explain vulnerability with CWE references
6. Identify CWE category (ID, name, description)
7. Respond only with JSON format

Choose one of the following CWE categories:
- CWE-79: Cross-site Scripting
- CWE-89: SQL Injection
- CWE-434: Dangerous File Upload
- CWE-352: CSRF
- None: No vulnerabilities

Output Requirements:
{
  "reasoning": "Detailed explanation...",
  "cwe_id": "CWE-XXX"
}

Code:
[ CODE INPUT ]

Now analyse the code provided and respond accordingly:
```

Figure 16 shows the CoT prompt template. The CoT template makes systematic analysis easier by guiding model through a structured process. It begins with an examination of code structure, followed by an assessment of input handling, then moves on to vulnerability detection, and finally enforces strict JSON response. By forcing step-by-step reasoning before generating formatted output, the template helps ensure a thorough analysis of both vulnerabilities.

Figure 17. Few-shots Prompt (FS)

```
You are a security expert tasked with identifying vulnerabilities in a given code.
Carefully analyze the code using CWE descriptions step by step.


Example 1: Command Injection (CWE-74)
$userName = $_POST["user"];
$command = 'ls -l /home/' . $userName;
system($command);

Response:
{
  "reasoning": "$userName is not checked for malicious input. Attacker could inject OS
  commands like: ;rm -rf /",
  "cwe_id": "CWE-74"
}


Example 2: [Vulnerability Type]
[Code Snippet]

Response:
[JSON response]

... [more examples]

Code:
[ CODE INPUT ]

Now analyse the code provided and respond accordingly:
```

Figure 18. Zero-shot Prompt (ZS)

```
Analyze the code and determine if it contains any vulnerabilities.

Choose one CWE category:
- CWE-79: Cross-site Scripting
- CWE-89: SQL Injection
- CWE-434: Dangerous File Upload
- CWE-352: CSRF
- None: No vulnerabilities found

Code:
[ CODE INPUT ]

Answer in JSON format:
{
  "reasoning": "Detailed explanation of why the code is safe or
  vulnerable",
  "cwe_id": "None" or "CWE-XX"
}

Your JSON response:
```

Figure 17 shows FS prompt template. The FS template provides five vulnerability examples from selected CWEs and one secure example, demonstrating the expected analysis pattern. Each case shows vulnerable code with proper JSON responses containing vulnerability status, code locations, explanations, and CWE references for consistent pattern matching. On the other hand, Zero-shot prompt contains only the first instruction and class choices. The prompt is shown in figure 18

4.5 Fine-tuning Methodology

This section describes the method used to fine-tune LLMs using the previously described training dataset. It is followed by a thorough evaluation of the models' performance using the test dataset.

The primary objective of the fine-tuning process is to systematically evaluate its efficacy in reducing false positives and false negatives. To this end, the model is trained using the dataset described in the dataset section. To optimize resource utilization during fine-tuning, this study employs QLoRA, a parameter-efficient fine-tuning technique described in the theoretical section. QLoRA allows for the targeted adaptation of specific model parameters while preserving most of the pre-trained weights, which significantly reduces computational demands. Integrating the Unsloth Python library further enhances memory efficiency, supporting fine-tuning on hardware with limited capacity and streamlining implementation complexities.

Following fine-tuning, the models are quantized to the Q8_0 format to optimize memory usage while maintaining performance integrity. This process is essential for efficient deployment of models in the Ollama framework.

Two input formats are used during fine-tuning. The first format includes a code snippet and its corresponding class label. The second format includes an intermediate reasoning step between the code snippet and the class label, extending the first format. The GPT-4o model generates this reasoning part, which explains why the code example is associated with a specific class.

4.5.1 Fine-tuning Format

Since the task involves code analysis and requires single-turn text generation based on structured instructions, we adopt the Alpaca-style instruction format [28]. Figure 19 shows the instruction format.

Figure 19. Instruction Input for Fine-tuning Models

```
SYSTEM: You are a highly skilled code analysis assistant specialized
in identifying security vulnerabilities in software code.

USER: Analyze the code and determine if it contains any
vulnerabilities.
Choose one of the following CWE categories:
- CWE-79: Improper Neutralization of Input During Web Page Generation
('Cross-site Scripting')
- CWE-89: Improper Neutralization of Special Elements used in an SQL
Command ('SQL Injection')
- CWE-434: Unrestricted Upload of File with Dangerous Type
- CWE-352: Cross-Site Request Forgery (CSRF)
- None: No vulnerabilities found

Code:
[ VULNERABLE FILE ]

ASSISTANT: [ CWE ID ]
```

4.5.2 Reasoning-Augmented Fine-tuning Approach

While the baseline fine-tuning approach improves the ability of LLMs to classify vulnerabilities using CWE labels, it does not explicitly improve the reasoning capabilities of the models, which can be a effective factor for accurate vulnerability detection according to recent research [29]. To address this limitation, we enhance the fine-tuning process with synthesized reasoning traces that connect code patterns to vulnerability characteristics.

Synthesizing Reasoning Input

We leverage GPT-4o to extract and structure reasoning from both vulnerable and non-vulnerable code samples. For vulnerable samples, the model generates explanations that map specific code patterns to CWE features using the template shown in Figure 20. For secure code examples, we use a template shown in Figure 21 that shows where security controls are present.

Figure 20. Prompt Template for Vulnerability Reasoning Generation

```
Analyze the provided code and explain why it is vulnerable to the
specified CWE-ID.
Your response should consist of only a concise explanation that
clearly links the vulnerable behavior in the code to the
characteristics of the CWE.
Do not include summaries, lists, or step-by-step breakdowns. Focus
only on the reasoning.

CODE: [ VULNERABLE CODE ]
CWE_ID: [ CWE ID and CWE Description]
```

Figure 21. Prompt Template for Secure Code Reasoning Generation

```
Analyze the provided code and explain why it is not vulnerable.
Your response should consist of only a concise explanation that
justifies the code's security with respect to common vulnerabilities.
Do not include summaries or step-by-step breakdowns. Focus only on
the reasoning and security measures present.

CODE: [ NON-VULNERABLE CODE ]
```

Integration of Reasoning into Fine-tuning

We modify the original Alpaca-style instruction format (Figure 19) to include synthesized reasoning before the final CWE classification. As shown in Figure 22, the assistant's response now contains both the vulnerability analysis rationale and the CWE identification. This approach trains the model to simultaneously develop detection capabilities and explainable reasoning patterns.

Figure 22. Instruction Input for Fine-tuning Models

```
SYSTEM: You are a highly skilled code analysis assistant specialized
in identifying security vulnerabilities in software code.

USER: Analyze the code and determine if it contains any
vulnerabilities.
Choose one of the following CWE categories:
- CWE-79: Improper Neutralization of Input During Web Page Generation
('Cross-site Scripting')
- CWE-89: Improper Neutralization of Special Elements used in an SQL
Command ('SQL Injection')
- CWE-434: Unrestricted Upload of File with Dangerous Type
- CWE-352: Cross-Site Request Forgery (CSRF)
- None: No vulnerabilities found

Code:
[ VULNERABLE CODE ]

ASSISTANT:
[ Reasoning ]
This is [ CWE ID ]
```

4.5.3 Fine-tuning Setting

Model Initialization

The pretrained language model was initialized with support for extended sequence lengths by applying Rotary Position Embedding (RoPE) scaling. To reduce computational resource demands without significantly sacrificing performance, the model was loaded using 4-bit quantization.

Parameter-Efficient Adaptation

To enable efficient fine-tuning with limited computational overhead, LoRA was applied to the model architecture. The adaptation was configured with the following parameters:

- **LoRA Rank**: Set to 8 for all adapters to balance efficiency.
- **Target Modules**: LoRA adapters were injected into all transformer projection layers, including query, key, value, and feedforward projections.
- **Scaling Factor** (α): Set to 8 to amplify the low-rank updates.
- **Dropout**: No dropout was applied.

Training Execution

The supervised fine-tuning was conducted using Hugging Face's SFTTrainer over three epochs. The training configuration included several optimizations for stability and efficiency:

- **Loss Computation**: The data collator was configured to compute loss exclusively on the last response part, to make sure that the model focused on optimizing the generation of relevant outputs.
- **Batch Size**: A per-device batch size of 2 was used, with gradient accumulation over 4 steps.
- **Optimizer**: The AdamW optimizer with 8-bit precision (`adamw_8bit`)
- **Learning Rate**: 2×10^{-4}
- **Post-training Conversion**: Upon completion of fine-tuning, the adapted model was quantized to GGUF format and uploaded to the Hugging Face Hub for sharing and deployment.

4.5.4 Fine-tuned Model Testing

After fine-tuning, the models are evaluated using the same methodology as the baseline testing. This includes testing with CoT, Zero-shot, Few-shots prompting strategies. Performance is measured using identical evaluation metrics to ensure consistency with baseline results.

4.6 Result Evaluation

To assess the performance of the classification model, we employ four evaluation metrics: Accuracy, Precision, Recall, and F1-score. These metrics are calculated from the confusion matrix, which tabulates true positive (TP), false positive (FP), true negative (TN), and false negative (FN) predictions.

4.6.1 Per-class Metrics

For each class i in the set of labels L, we calculate:

- **True Positives (TP)**: Correctly predicted instances of class i

$$TP_i = C_{i,i}$$

- **False Positives (FP)**: Instances predicted as class i but belonging to other classes

$$FP_i = \left(\sum_{j=1}^{|L|} C_{j,i} \right) - TP_i$$

- **False Negatives (FN)**: Instances of class i predicted as other classes

$$FN_i = \left(\sum_{j=1}^{|L|} C_{i,j} \right) - TP_i$$

- **True Negatives (TN)**: Instances not belonging to class i and correctly not predicted as class i

$$TN_i = N_{\text{total}} - (TP_i + FP_i + FN_i)$$

Where C is the confusion matrix and N_{total} is the total number of instances.

4.6.2 Performance Metrics

The metrics are computed as follows:

- **Precision** (Positive Predictive Value) for class i:

$$\text{Precision}_i = \frac{TP_i}{TP_i + FP_i}$$

- **Recall** (True Positive Rate, Sensitivity) for class i:

$$\text{Recall}_i = \frac{TP_i}{TP_i + FN_i}$$

- **F1-score** (Harmonic mean of Precision and Recall) for class i:

$$\text{F1}_i = 2 \times \frac{\text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}$$

- **Accuracy** (Overall correctness across all classes):

$$\text{Accuracy} = \frac{\sum_{i=1}^{|L|} TP_i}{N_{\text{total}}}$$

4.6.3 Weighted-averaged Metrics

For multi-class evaluation with class imbalance, we compute weighted-averaged metrics by taking the support-weighted mean of per-class metrics:

- **Weighted Precision**:

$$\text{Weighted-Precision} = \frac{\sum_{i=1}^{|L|} \text{Precision}_i \times w_i}{\sum_{i=1}^{|L|} w_i}$$

  where w_i is the number of true instances (support) for class i.
- **Weighted Recall**:

$$\text{Weighted-Recall} = \frac{\sum_{i=1}^{|L|} \text{Recall}_i \times w_i}{\sum_{i=1}^{|L|} w_i}$$

- **Weighted F1-score**:

$$\text{Weighted-F1} = \frac{\sum_{i=1}^{|L|} \text{F1}_i \times w_i}{\sum_{i=1}^{|L|} w_i}$$

- **Weighted False Negative Rate (FNR)**:

$$\text{Weighted-FNR} = \frac{\sum_{i=1}^{|L|} \text{FNR}_i \times w_i}{\sum_{i=1}^{|L|} w_i}, \quad \text{where} \quad \text{FNR}_i = \frac{\text{FN}_i}{\text{TP}_i + \text{FN}_i}$$

- **Weighted False Positive Rate (FPR)**:

$$\text{Weighted-FPR} = \frac{\sum_{i=1}^{|L|} \text{FPR}_i \times w_i}{\sum_{i=1}^{|L|} w_i}, \quad \text{where} \quad \text{FPR}_i = \frac{\text{FP}_i}{\text{FP}_i + \text{TN}_i}$$

These weighted metrics provide class distribution-aware evaluation, where each class's contribution is proportional to its frequency in the data set. This approach is particularly useful for unbalanced datasets because it automatically adjusts for class frequencies, giving more influence to majority classes while still accounting for the performance of minority classes. The weighted average thus provides a more realistic performance estimate that reflects the actual data distribution, maintaining interpretability while being sensitive to the relative importance of different classes based on their frequency of occurrence.

4.6.4 Macro Metrics: F1, FNR, FPR

Macro-averaged metrics (F1, FNR, FPR) complement weighted metrics by treating all classes equally, ensuring that minority classes are not lost in imbalanced datasets. While weighted metrics reflect performance biased toward common classes, macro metrics penalize inconsistent performance across classes, which is critical in areas where rare classes demand equal attention. Reporting on both provides a holistic view: weighted metrics prioritize dominant trends, while macro metrics expose robustness gaps in classes that are less common.

- **Macro F1-score**:

$$\text{Macro-F1} = \frac{1}{|L|}\sum_{i=1}^{|L|} \text{F1}_i, \quad \text{where} \quad \text{F1}_i = 2 \cdot \frac{\text{Precision}_i \cdot \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}$$

- **Macro False Negative Rate (FNR)**:

$$\text{Macro-FNR} = \frac{1}{|L|}\sum_{i=1}^{|L|} \text{FNR}_i, \quad \text{where} \quad \text{FNR}_i = \frac{\text{FN}_i}{\text{TP}_i + \text{FN}_i}$$

- **Macro False Positive Rate (FPR)**:

$$\text{Macro-FPR} = \frac{1}{|L|}\sum_{i=1}^{|L|} \text{FPR}_i, \quad \text{where} \quad \text{FPR}_i = \frac{\text{FP}_i}{\text{FP}_i + \text{TN}_i}$$

5. Experiment Result

This section presents the evaluation results for baseline and fine-tuned code vulnerability detection models across three prompt strategies: Zero-Shot (ZS), Chain-of-Thought (CoT), and Few-Shot (FS). They are applied to both file-level and function-level code inputs. The analysis is structured into four comparisons:

1. Baseline file-level results
2. Fine-tuned file-level results
3. Baseline function-level results
4. Fine-tuned function-level results

As detailed in section 4, two fine-tuning approaches were applied:

- **FT**: Fine-tuning with code samples and their corresponding CWE labels.
- **FT-explain**: Fine-tuning with code samples, reasoning steps, and CWE labels.

For each comparison, we assess performance variations across prompt types, CWE classes, and programming languages, with emphasis on detection accuracy and error rates. To ensure a precise evaluation under class imbalance, all metrics are computed using:

- **Class-weighted statistics** (prioritizing dominant classes like non-vulnerable code, CWE-89, and CWE-79).
- **Macro-averaged statistics** (F_1 score, False Negative Rate (FNR), False Positive Rate (FPR)) to equally weight all classes.

In results tables, macro-averaged metrics are prefixed with “Ma” and fine-tuned results are labeled “FT.”

5.1 File Level Vulnerability Detection

5.1.1 Baseline Test Result

Zero-shot (ZS) Prompt

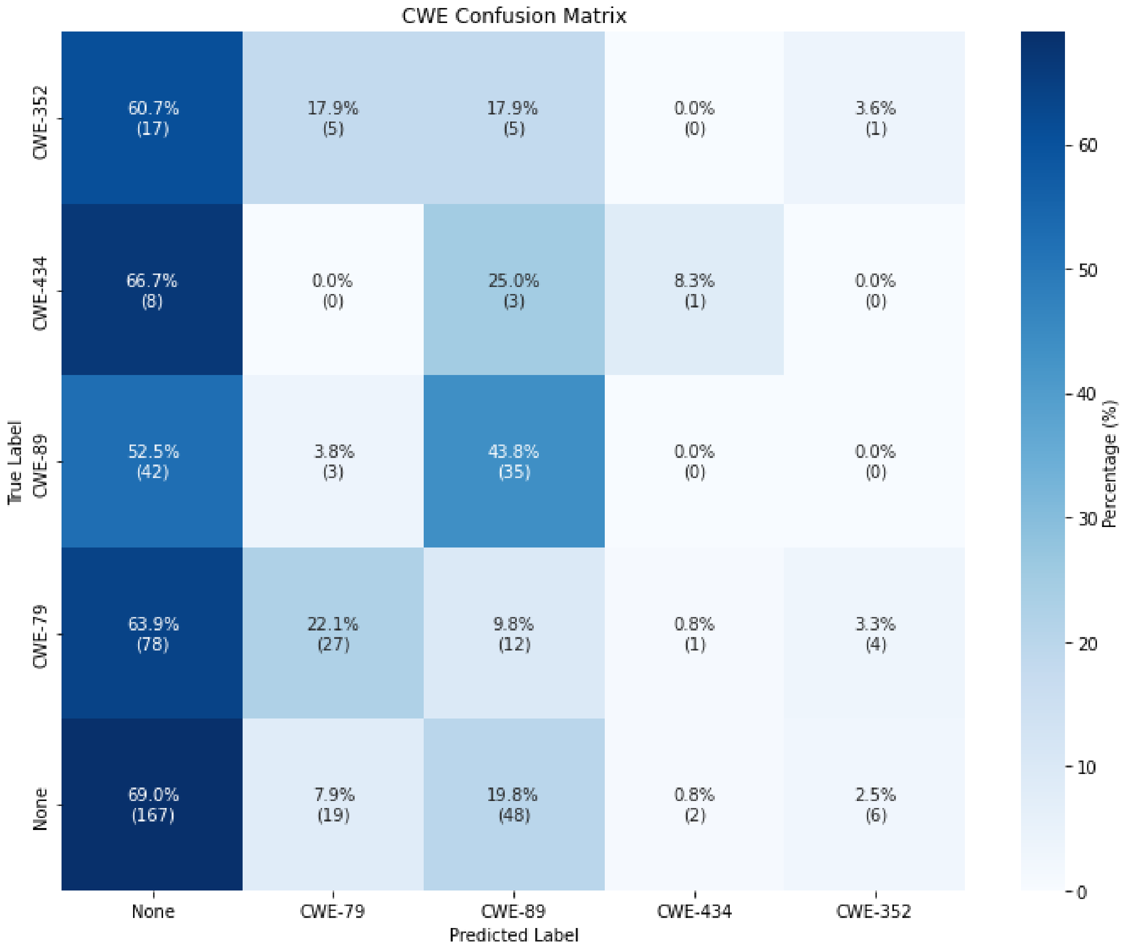

Figure 23. Confusion Matrix: File Level Vulnerability Detection with Zero-shot Prompt

Figure 23 shows the result of ZS prompt-based file-level vulnerability detection. Over 50% of the vulnerable code samples were misclassified as non-vulnerable. This misclassification was particularly marked for samples tagged with CWE-434 and CWE-352, which were mostly classified as non-vulnerable or incorrectly assigned to different CWE classes.

In contrast, a moderate proportion of samples from CWE-79 and CWE-89 were correctly identified (approximately 20-30%), although the majority were still misclassified as non-vulnerable. It is also notable that non-vulnerable samples had the highest classification accuracy, but about 30% of them were misclassified as vulnerable.

Chain of Thought (CoT) Prompt

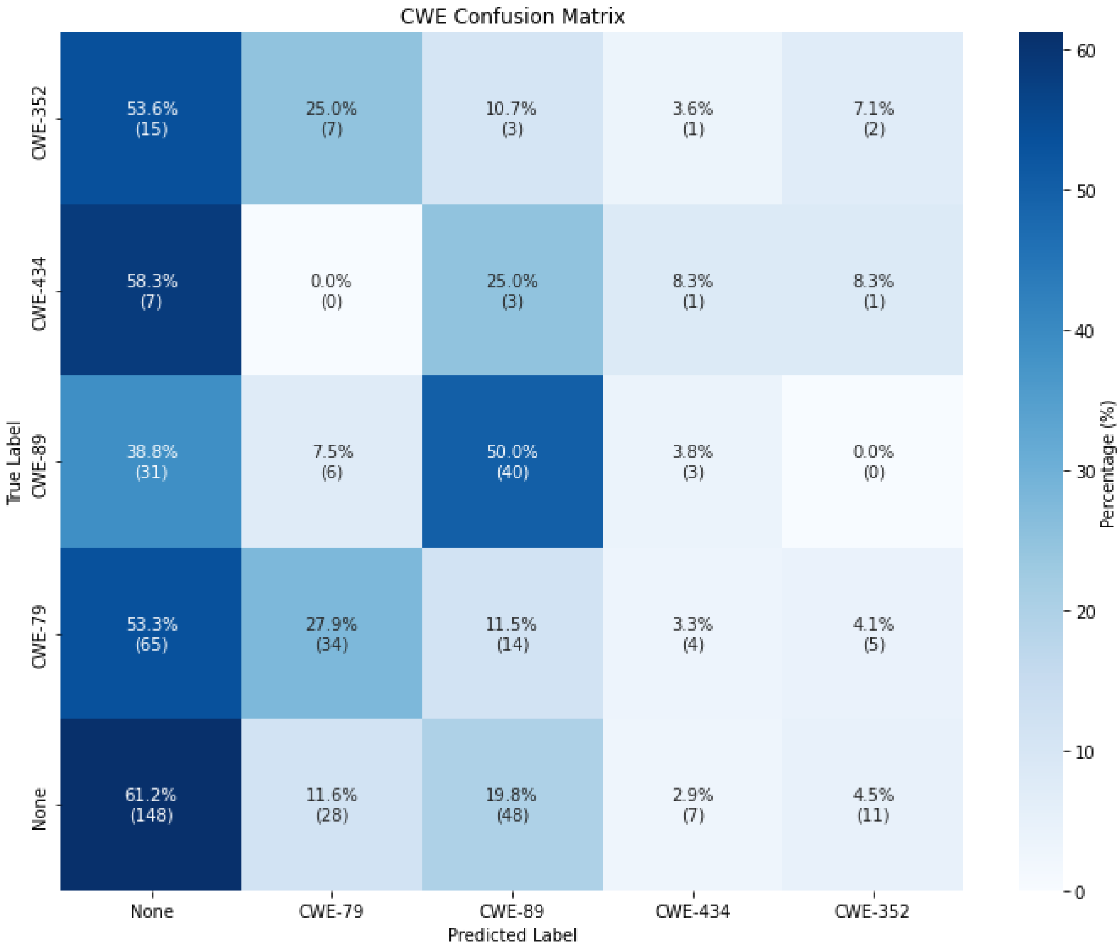

Figure 24. Confusion Matrix: File Level Vulnerability Detection with CoT Prompt

Figure 24 shows the result of CoT prompt based file level vulnerability detection. The true positive rates for the CWE-79 and CWE-89 samples improved compared to the ZS prompt results. In particular, the CWE-89 samples achieved a true positive identification rate of over 50%. However, the misclassification rate of non-vulnerable samples increased slightly by 1-2%. In addition, the model continued to struggle to correctly identify CWE-434 and CWE-352, indicating difficulties in detecting these vulnerability types.

Few-shots (FS) Prompt

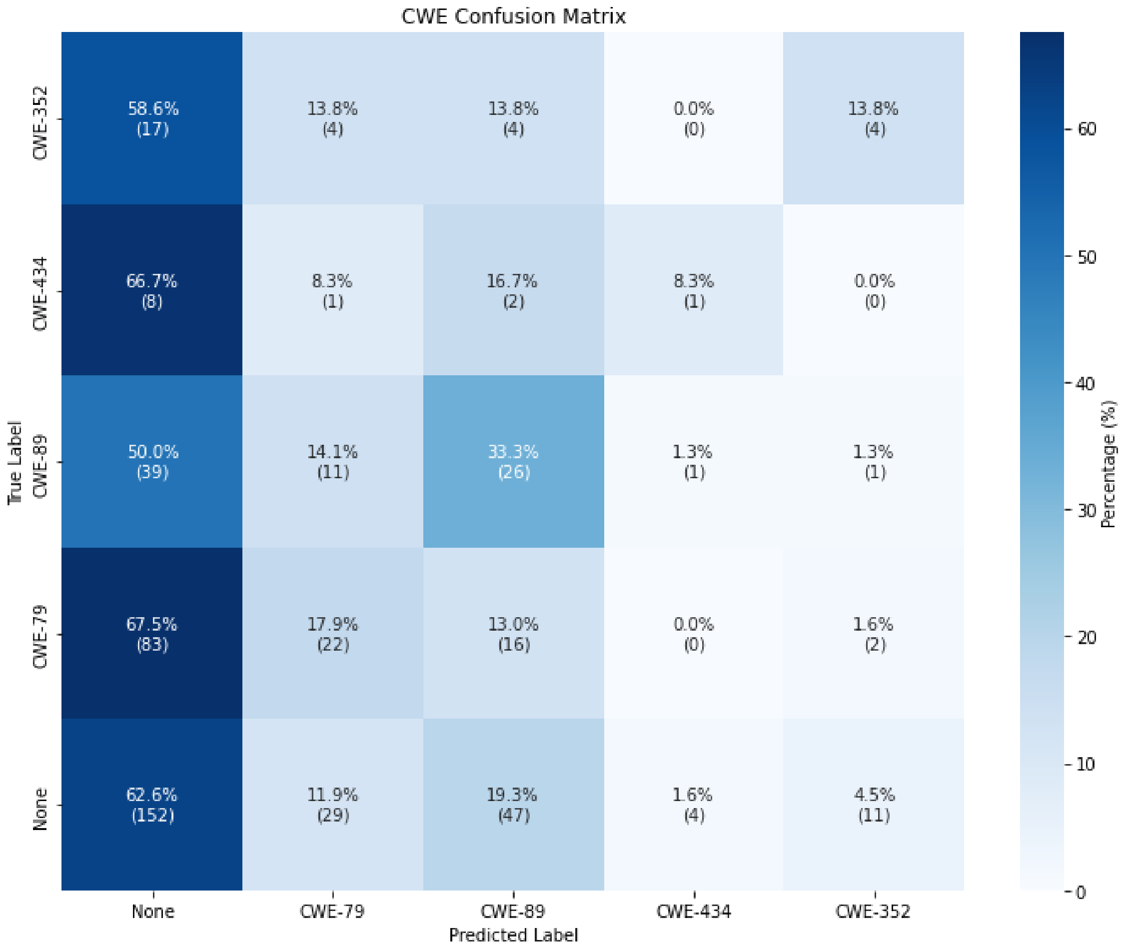

Figure 25. Confusion Matrix: File Level Vulnerability Detection with FS Prompt

Figure 25 shows the results of the FS prompt based file level vulnerability detection. The result shows a severe misclassifications tendency, with more than 50% of vulnerable samples incorrectly classified as non-vulnerable across all tested cases. Among the prompt types evaluated, CWE-352 has the highest amount of true positives in this result.

5.1.2 Fine-tuned Model Test Result

Zero-shot (ZS) Prompt

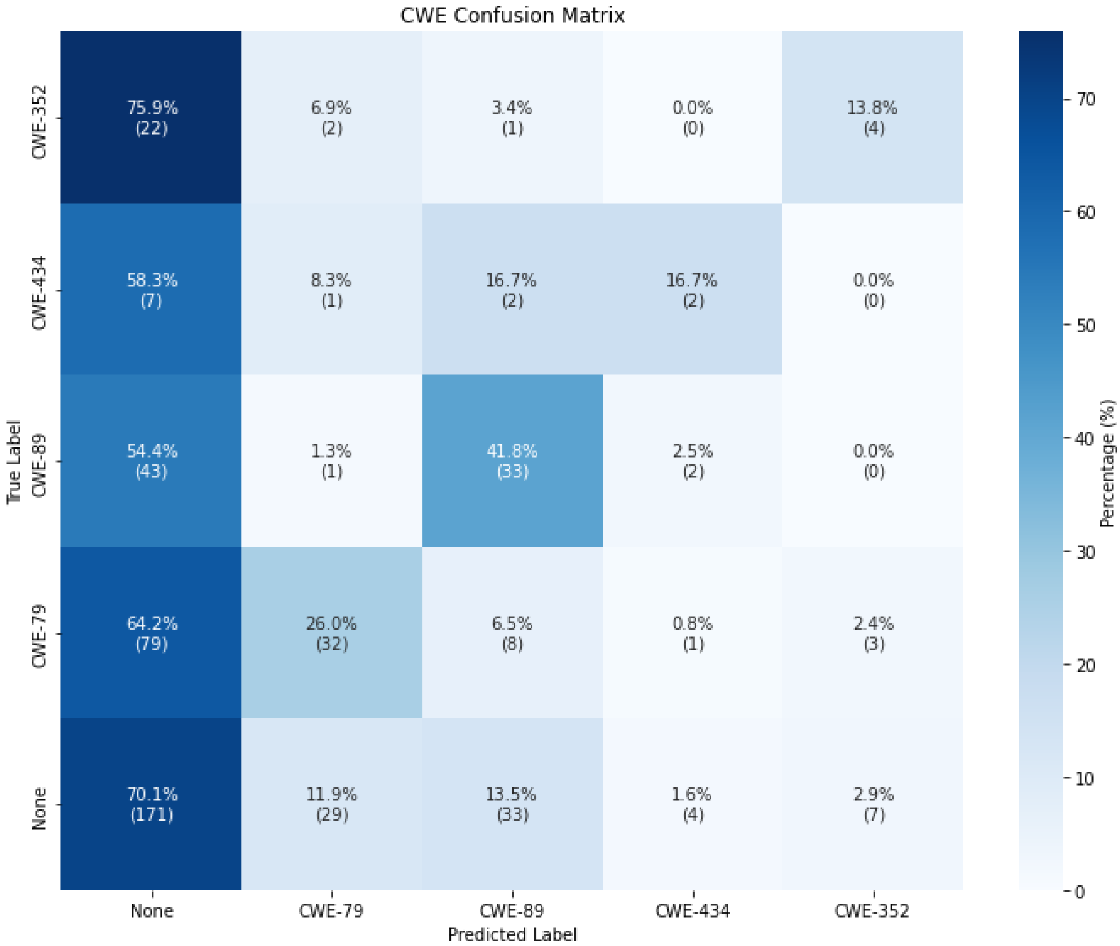

Figure 26. Confusion Matrix: File Level Vulnerability Detection with Zero-shot Prompt with Fine-tuned Model

Figure 26 presents the confusion matrix for ZS prompt based file-level vulnerability identification using our fine-tuned model. The model tends to misclassify vulnerable classes as non-vulnerable classes. However, slight improvements in true positive rates are observed for CWE-79, CWE-434, and CWE-352, as well as for the non-vulnerable class. Compared to the baseline zero-shot prompting approach, there are increases of 2–4%. Overall, the number of false positives varies by class. Some classes show an increase, while others show a decrease.

Chain of Thought (CoT) Prompt

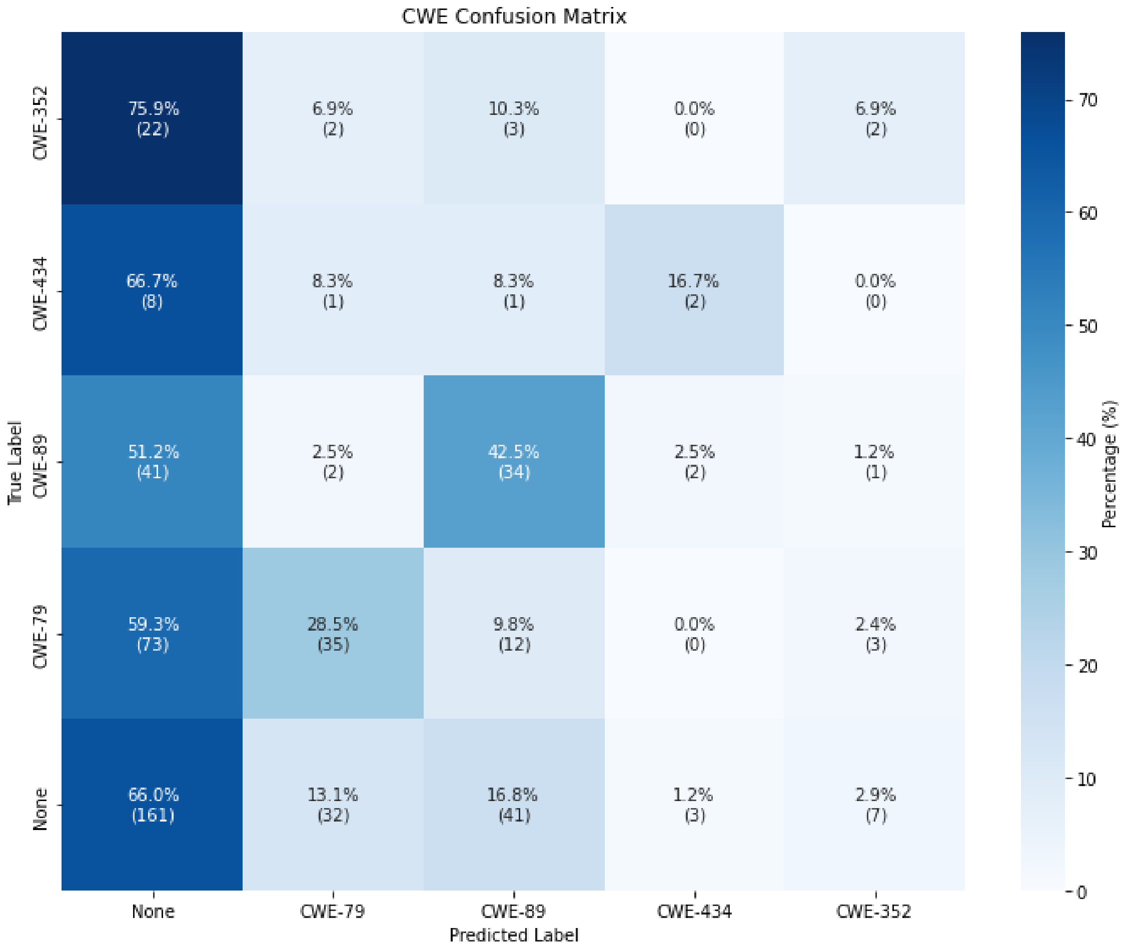

Figure 27. Confusion Matrix: File Level Vulnerability Detection with CoT Prompt with Fine-tuned Model

Figure 27 shows the confusion matrix for the fine-tuned model using the CoT prompt. The results show a different classification pattern than other prompt styles. While the model maintains similar detection accuracy for vulnerable instances, there is an increased misclassification rate of vulnerable samples as non-vulnerable compared to the baseline CoT prompt model. This suggests that the fine-tuned model may have a more sensitive or shifted decision boundary. Overall, there is no significant improvement in terms of false positives or false negatives.

Few-shots (FS) Prompt

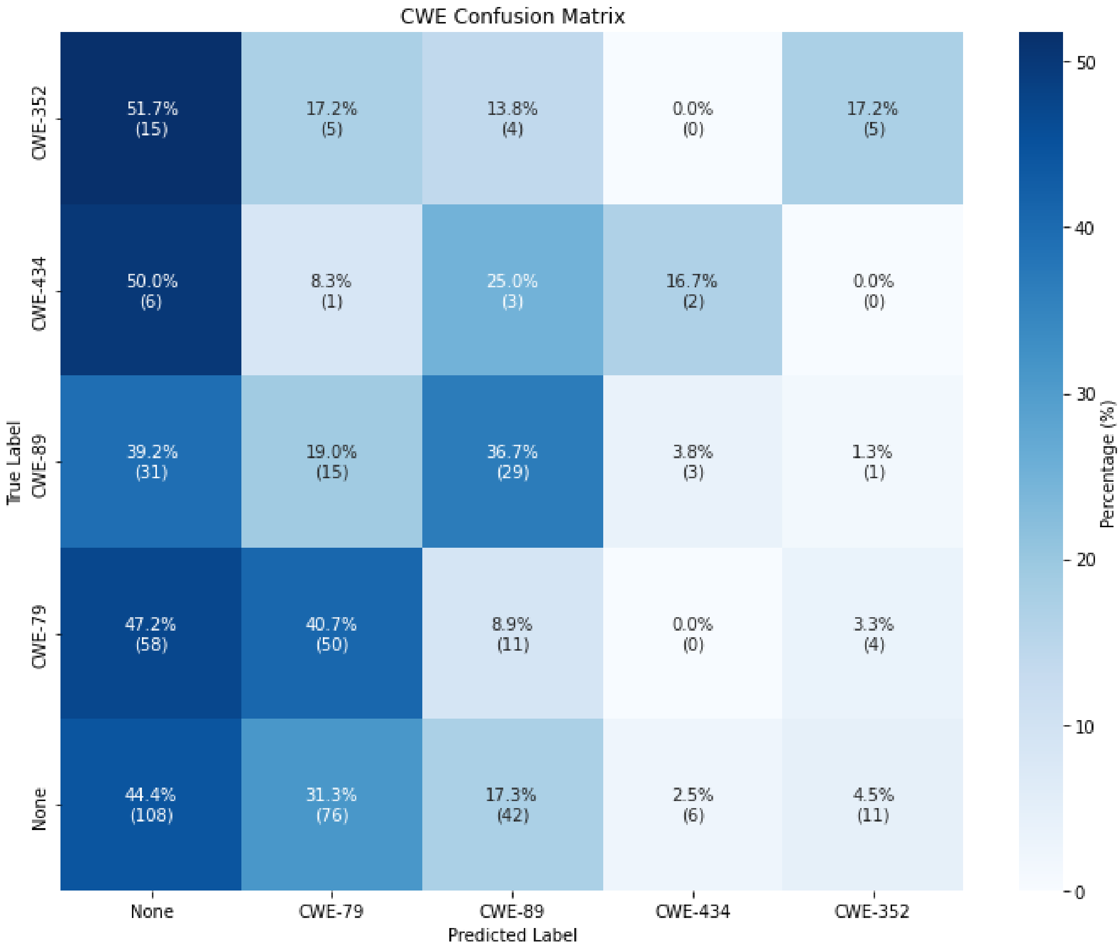

Figure 28. Confusion Matrix: File Level Vulnerability Detection with FS Prompt with Fine-tuned Model

Figure 28 shows the performance characteristics of our FS prompt approach. The results show a significant improvement in the detection of CWE-79 vulnerabilities, with a true positive rate of 40%. However, for non-vulnerable and CWE-89 class samples, the true positive rate decreases compared to other fine-tuned models. It is also notable that misclassifications do not only occur as non-vulnerable. Some are misclassified as other vulnerability types instead.

5.1.3 Fine-tuning with Synthesized Input

Zero-shot (ZS) Prompt

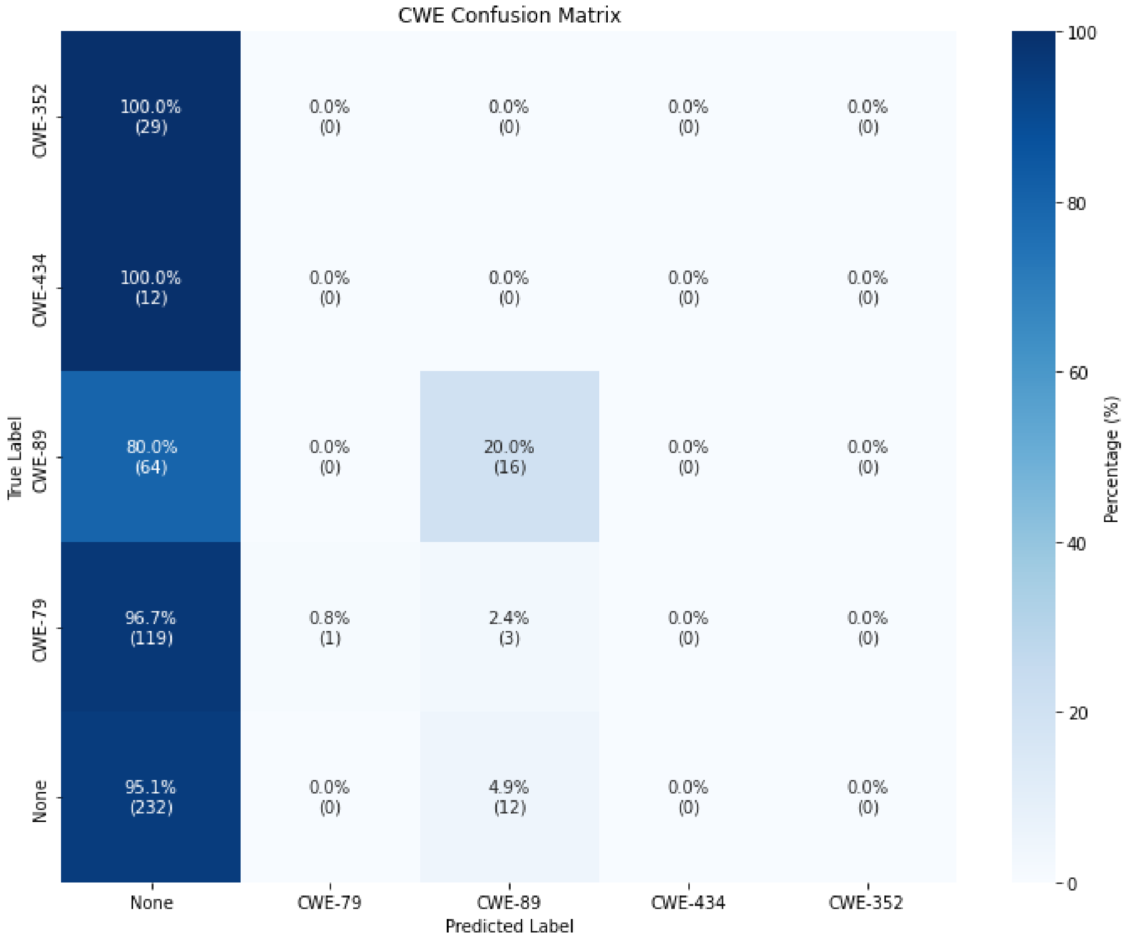

Figure 29. File-Level Vulnerability Detection Performance (ZS Prompting with Reasoning-Augmented Fine-Tuned Model)

Figure 29 shows the results of the Reasoning-Augmented Fine-Tuned Model with the ZS prompt. This approach is only slightly effective, as the model misclassifies most samples as non-vulnerable across all CWE categories. While there are a few correct classifications for CWE-79 and CWE-89, the model fails to detect vulnerabilities in the remaining classes. Unlike the simplified fine-tuned results, the ZS approach has difficulty distinguishing between vulnerable and non-vulnerable samples.

Chain of Thought (CoT) Prompt

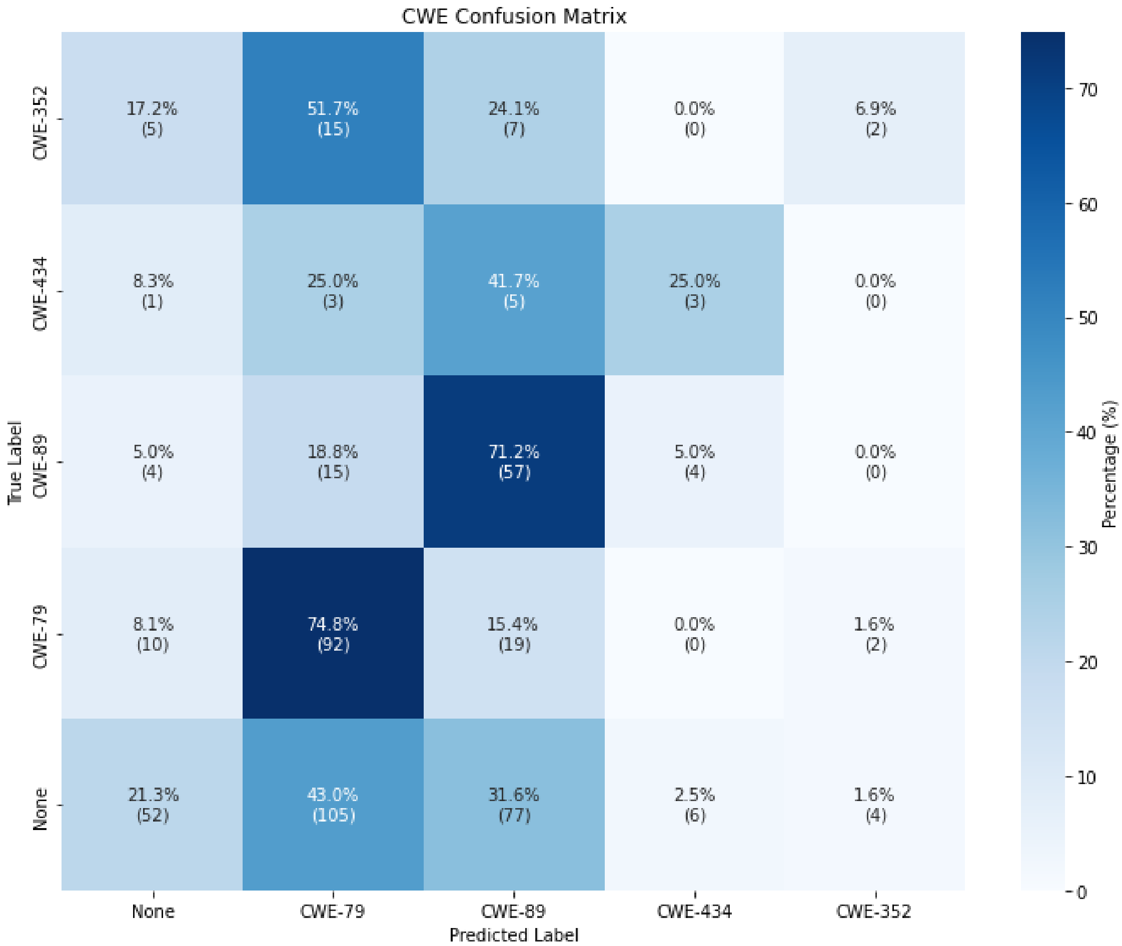

Figure 30. File-Level Vulnerability Detection Performance (CoT Prompting with Reasoning-Augmented Fine-Tuned Model)

Figure 30 shows the confusion matrix for the Reasoning-Augmented Fine-Tuned Model using the CoT prompt. Compared to the same model with ZS prompt, the tendency to classify samples as not vulnerable is reduced. Instead, the number of true positives for vulnerable classes increases. For example, approximately 70% of the samples in CWE-79 and CWE-89 are classified correctly. However, samples in other classes are often misclassified. Most of these misclassified samples are labeled as CWE-79 or CWE-89, suggesting the model tends to falsely predict these two classes. Overall, the model appears to struggle in capturing the distinguishing characteristics of classes other than CWE-79 and CWE-89.

Few-shots (FS) Prompt

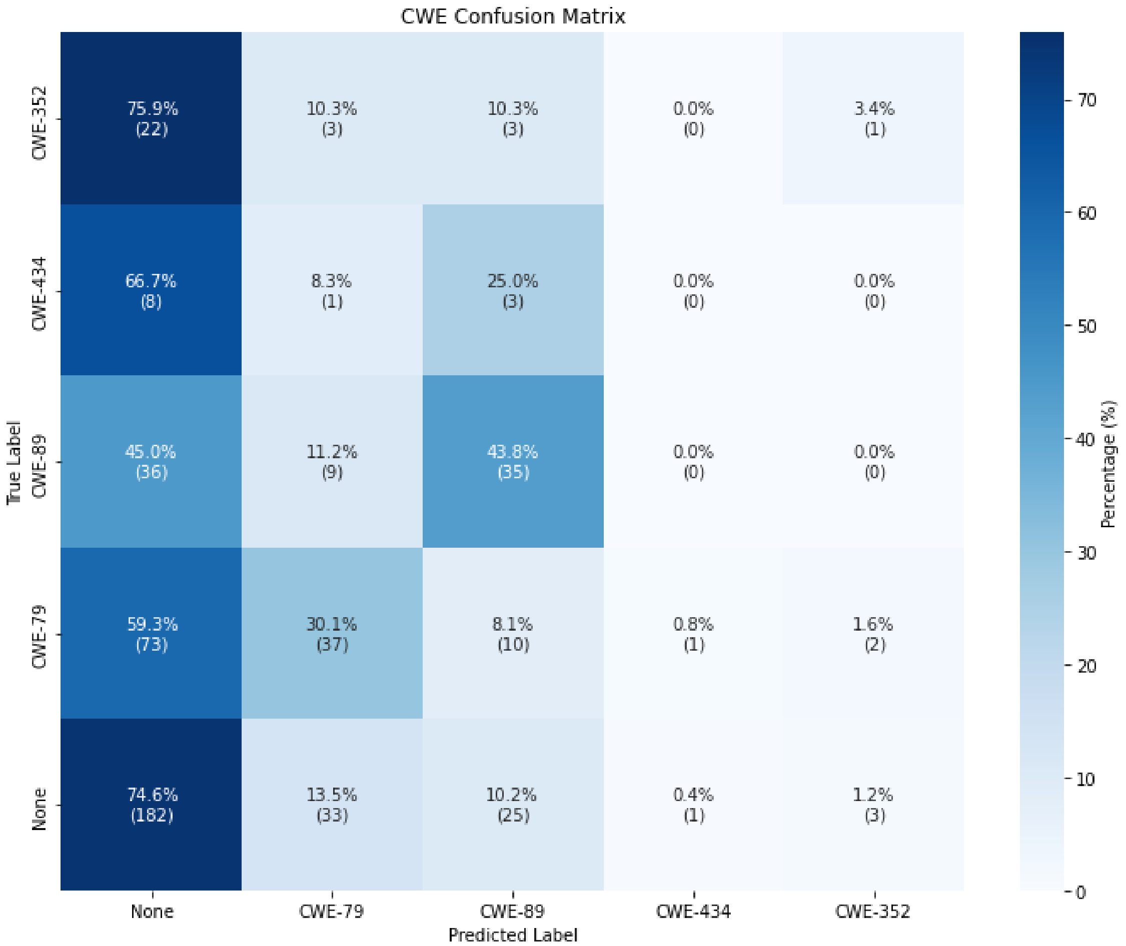

Figure 31. File-Level Vulnerability Detection Performance (FS Prompting with Reasoning-Augmented Fine-Tuned Model)

Figure 31 shows the confusion matrix for the Reasoning-Augmented Fine-Tuned Model using the FS prompt. Although there are some true positives for CWE-79 and CWE-89, as well as for non-vulnerable samples, a significant number of vulnerable samples are misclassified as non-vulnerable. In particular, there are nearly zero true positives for CWE-434 and CWE-352.

Compared to the simplified version of the fine-tuned model, overall performance has degraded. Though this model outperforms the baseline approach with the same prompt in some categories, such as non-vulnerable, CWE-79, and CWE-89, it struggles to detect minority cases effectively.

5.1.4 Statistics Overall: Comparison of Baseline and Fine-tuned Model

Table 2. Class Performance Metrics: File Level

CWE-Class	Prompt	FNR	FPR	Acc	Precision	Recall	F1
None	ZS	0.31	0.60	0.55	0.54	0.69	0.60
	ZS-FT	0.30	0.62	0.54	0.53	0.70	0.60
	ZS-FT-explain	0.00	0.98	0.51	0.51	1.00	0.67
	CoT	0.39	0.49	0.56	0.56	0.61	0.58
	CoT-FT	0.34	0.59	0.53	0.53	0.66	0.59
	CoT-FT-explain	0.79	0.08	0.57	0.72	0.21	0.33
	FS	0.37	0.61	0.51	0.51	0.63	0.56
	FS-FT	0.56	0.45	0.50	0.50	0.44	0.47
	FS-FT-explain	0.25	0.57	0.59	0.57	0.75	0.64
CWE-79	ZS	0.78	0.07	0.75	0.50	0.22	0.31
	ZS-FT	0.74	0.09	0.75	0.49	0.26	0.34
	ZS-FT-explain	1.00	0.00	0.75	0.00	0.00	0.00
	CoT	0.72	0.11	0.73	0.45	0.28	0.35
	CoT-FT	0.72	0.10	0.74	0.49	0.28	0.36
	CoT-FT-explain	0.25	0.38	0.65	0.40	0.75	0.52
	FS	0.82	0.12	0.70	0.33	0.18	0.23
	FS-FT	0.59	0.27	0.65	0.34	0.41	0.37
	FS-FT-explain	0.70	0.13	0.73	0.45	0.30	0.36
CWE-89	ZS	0.56	0.17	0.77	0.34	0.44	0.38
	ZS-FT	0.58	0.11	0.82	0.43	0.42	0.42
	ZS-FT-explain	0.95	0.00	0.84	0.80	0.05	0.09
	CoT	0.50	0.17	0.78	0.37	0.50	0.43
	CoT-FT	0.57	0.14	0.79	0.37	0.42	0.40
	CoT-FT-explain	0.29	0.26	0.73	0.35	0.71	0.47
	FS	0.67	0.17	0.75	0.27	0.33	0.30
	FS-FT	0.63	0.15	0.77	0.33	0.37	0.35
	FS-FT-explain	0.56	0.10	0.82	0.46	0.44	0.45
CWE-434	ZS	0.92	0.01	0.97	0.25	0.08	0.12
	ZS-FT	0.83	0.01	0.97	0.22	0.17	0.19
	ZS-FT-explain	1.00	0.00	0.98	0.00	0.00	0.00
	CoT	0.92	0.03	0.95	0.06	0.08	0.07
	CoT-FT	0.83	0.01	0.97	0.29	0.17	0.21
	CoT-FT-explain	0.75	0.02	0.96	0.23	0.25	0.24
	FS	0.92	0.01	0.97	0.17	0.08	0.11
	FS-FT	0.83	0.02	0.96	0.18	0.17	0.17
	FS-FT-explain	1.00	0.00	0.97	0.00	0.00	0.00
CWE-352	ZS	0.96	0.02	0.92	0.09	0.04	0.05
	ZS-FT	0.86	0.02	0.93	0.29	0.14	0.19
	ZS-FT-explain	1.00	0.00	0.94	0.00	0.00	0.00
	CoT	0.93	0.04	0.91	0.11	0.07	0.09
	CoT-FT	0.93	0.02	0.92	0.15	0.07	0.10
	CoT-FT-explain	0.93	0.01	0.93	0.25	0.07	0.11
	FS	0.86	0.03	0.92	0.22	0.14	0.17
	FS-FT	0.83	0.04	0.92	0.24	0.17	0.20
	FS-FT-explain	0.97	0.01	0.93	0.17	0.03	0.06

Table 4. Language Performance Metrics: File Level Detection

Language	Method	Recall	Precision	F1	$F1_{Ma}$	FNR	FPR
PHP	ZS	0.45	0.44	0.44	0.30	0.55	0.32
	ZS-FT	0.47	0.45	0.45	0.32	0.53	0.35
	ZS-FT-explain	0.51	0.37	0.35	0.15	0.49	0.49
	CoT	0.43	0.45	0.43	0.29	0.57	0.27
	CoT-FT	0.46	0.45	0.45	0.34	0.54	0.34
	CoT-FT-explain	0.40	0.52	0.36	0.29	0.60	0.17
	FS	0.39	0.38	0.38	0.27	0.61	0.34
	FS-FT	0.35	0.37	0.35	0.28	0.65	0.31
	FS-FT-explain	0.51	0.45	0.47	0.28	0.49	0.33
JavaScript	ZS	0.56	0.60	0.48	0.25	0.44	0.43
	ZS-FT	0.63	0.65	0.59	0.47	0.37	0.35
	ZS-FT-explain	0.50	0.25	0.33	0.17	0.50	0.50
	CoT	0.56	0.58	0.51	0.27	0.44	0.41
	CoT-FT	0.56	0.56	0.52	0.28	0.44	0.37
	CoT-FT-explain	0.54	0.59	0.48	0.27	0.46	0.33
	FS	0.52	0.54	0.41	0.21	0.48	0.44
	FS-FT	0.50	0.52	0.49	0.49	0.50	0.43
	FS-FT-explain	0.63	0.60	0.62	0.43	0.37	0.29
Java	ZS	0.54	0.41	0.46	0.31	0.46	0.38
	ZS-FT	0.54	0.43	0.48	0.32	0.46	0.37
	ZS-FT-explain	0.58	0.52	0.48	0.30	0.42	0.42
	CoT	0.54	0.41	0.46	0.31	0.46	0.38
	CoT-FT	0.50	0.38	0.43	0.29	0.50	0.39
	CoT-FT-explain	0.46	0.48	0.37	0.45	0.54	0.15
	FS	0.54	0.41	0.46	0.31	0.46	0.38
	FS-FT	0.46	0.41	0.43	0.33	0.54	0.36
	FS-FT-explain	0.54	0.41	0.46	0.31	0.46	0.38
TypeScript	ZS	0.57	0.72	0.49	0.37	0.43	0.36
	ZS-FT	0.54	0.57	0.47	0.36	0.46	0.39
	ZS-FT-explain	0.50	0.25	0.33	0.17	0.50	0.50
	CoT	0.54	0.71	0.47	0.36	0.46	0.36
	CoT-FT	0.54	0.64	0.54	0.36	0.46	0.28
	CoT-FT-explain	0.50	0.55	0.46	0.50	0.50	0.29
	FS	0.50	0.28	0.36	0.27	0.50	0.40
	FS-FT	0.54	0.57	0.54	0.40	0.46	0.29
	FS-FT-explain	0.50	0.56	0.45	0.23	0.50	0.39
Ruby	ZS	0.50	0.33	0.38	0.20	0.50	0.47
	ZS-FT	0.50	0.48	0.48	0.37	0.50	0.36
	ZS-FT-explain	0.50	0.25	0.33	0.13	0.50	0.50
	CoT	0.58	0.40	0.47	0.29	0.42	0.38
	CoT-FT	0.42	0.32	0.36	0.23	0.58	0.41
	CoT-FT-explain	0.38	0.44	0.34	0.33	0.63	0.19
	FS	0.46	0.30	0.36	0.21	0.54	0.44
	FS-FT	0.50	0.41	0.45	0.30	0.50	0.33
	FS-FT-explain	0.46	0.35	0.40	0.25	0.54	0.38
Python	ZS	0.50	0.48	0.40	0.24	0.50	0.46
	ZS-FT	0.55	0.41	0.45	0.29	0.45	0.42

Continued on next page

Table 4. Weighted Performance Metrics for Different Languages (Continued)

Language	Method	Recall	Precision	F1	$F1_{Ma}$	FNR	FPR
	ZS-FT-explain	0.50	0.25	0.33	0.17	0.50	0.50
	CoT	0.45	0.44	0.41	0.30	0.52	0.42
	CoT-FT	0.50	0.39	0.43	0.30	0.50	0.40
	CoT-FT-explain	0.45	0.54	0.41	0.36	0.55	0.18
	FS	0.45	0.39	0.40	0.34	0.55	0.44
	FS-FT	0.55	0.61	0.53	0.49	0.45	0.35
	FS-FT-explain	0.50	0.43	0.46	0.34	0.50	0.37

Table 5. Overall Performance Metrics: File Level Vulnerability Detection

Prompt	Recall	Precision	F1	$F1_{Ma}$	FNR	FNR_{Ma}	FPR	FPR_{Ma}
ZS	0.48	0.46	0.45	0.29	0.52	0.71	0.35	0.17
ZS-FT	0.50	0.48	0.47	0.35	0.50	0.66	0.35	0.17
ZS-FT-explain	0.51	0.38	0.35	0.15	0.49	0.79	0.49	0.20
CoT	0.46	0.46	0.46	0.30	0.54	0.69	0.30	0.17
CoT-FT	0.48	0.46	0.46	0.33	0.52	0.68	0.35	0.17
CoT-FT-explain	0.42	0.54	0.38	0.33	0.58	0.60	0.18	0.15
FS	0.42	0.40	0.40	0.27	0.58	0.73	0.37	0.19
FS-FT	0.40	0.41	0.40	0.31	0.60	0.69	0.32	0.18
FS-FT-explain	0.52	0.48	0.49	0.30	0.48	0.70	0.33	0.16

Table 5 presents the overall scores of each prompting technique and the fine-tuned models using identical prompt styles in file level input. Among the unmodified model settings, CoT achieves the most balanced performance, with the highest F1 score (Weighted F1:0.46 and Macro F1: 0.30). On the other hand, FS prompt result performs the lowest F1 scores (Weighted F1:0.40 and Macro F1: 0.27).

When comparing the fine-tuned models, FS-FT-explain achieved the highest weighted F1 score (0.49). However, in terms of macro F1 score, ZS-FT performed best (0.35). This difference occurs because the ZS-FT model correctly classified more samples from the minority classes (CWE-434 and CWE-352), while FS-FT-explain classified more samples from the majority class (non-vulnerable, CWE-79 and CWE-89). Another interesting observation from this comparison is that CoT-FT-explain achieved the lowest false positive rate for both weighted and macro scores (Weighted FPR:0.18 and Macro FPR:0.15).

As shown in Table 2, which presents performance metrics across different classes, the results vary depending on the prompting strategy and the CWE category.

For the unmodified model, the highest F1 scores are achieved by all prompts for the Non-vulnerable class, followed by CWE-89 and CWE-79. Among the minority classes, the ZS prompt performs better on CWE-434 than on CWE-352. In contrast, the CoT prompt slightly favors CWE-352 over CWE-434. The FS prompt also performs better on CWE-352 than on CWE-434.

Overall, the CoT prompt perfoems better performance than other prompting strategies across most classes, while the FS prompt shows improved performance on the minority classes (CWE-434 and CWE-352). Nevertheless, both minority classes exhibit low performance across all metrics, indicating the need for more specialized or targeted approaches. Additionally, the non-vulnerable class has a high false positive rate (FPR) with all prompting techniques.

The fine-tuning process resulted in noticeable performance improvements across multiple metrics, most notably a reduction in FNR (CWE-352 ZS: 0.96 $\rightarrow$ 0.86) and an increase in recall (CWE-79 FS: 0.18 $\rightarrow$ 0.41), which in many cases led to F1 score gains (CWE-79 FS: 0.23 $\rightarrow$ 0.37, CWE-434 CoT: 0.07 $\rightarrow$ 0.21). However, these benefits came with trade-offs, including increased FPR (None ZS: 0.60 $\rightarrow$ 0.62, CWE-79 FS: 0.12 $\rightarrow$ 0.27) and occasional decreases in precision (CWE-434 ZS: 0.25 $\rightarrow$ 0.22). Among the results, CWE-352 achieved the largest ZS F1 gain (+0.14) through balanced improvements in precision and recall. These findings shows the effectiveness of fine-tuning in enhancing detection capabilities, while also highlighting the associated performance trade-offs. Additionally, the reason-augmented model shows improvement the best when used with CoT prompts. For example, in Java, CoT-FT-explain significantly decreases the false positive rate (FPR) from 0.39 (CoT-FT) to 0.15. Similarly, in Python, CoT-FT-explain achieves a lower FPR (0.18) than CoT (0.42) and CoT-FT (0.40) while maintaining a similar F1 score.

Table 4 shows the result of language specific performance. ZS works best with PHP and TypeScript. It finds more vulnerabilities (high recall), but has a slightly higher FPR. CoT is better for JavaScript, Ruby and Python. With a better F1 score, it strikes a balance between finding vulnerabilities and being accurate. FS struggles in all languages, misses vulnerabilities more often (low F1), and performs poorly overall. Uniquely, Java performs across the different prompts, suggesting that it's harder to find vulnerabilities. Overall, CoT performs best with balanced results.

Fine-tuning had a positive effect on the results for most programming languages. It often reduced the FPR, as seen in JavaScript (ZS: 0.43 $\rightarrow$ 0.35) and TypeScript (CoT: 0.36 $\rightarrow$ 0.28). However, an exception was observed in Ruby (ZS: 0.36 $\rightarrow$ 0.47). Fine-tuning also reduced the FNR in several cases, including PHP (CoT: 0.57 $\rightarrow$ 0.54), JavaScript (ZS: 0.44

→ 0.37), and Python (ZS: 0.50 → 0.45).

Reason-augmented models, particularly when combined with CoT prompting, performed further improvements. For instance, the CoT-FT-explain model significantly reduced the FPR in Java from 0.39 to 0.15. Similarly, CoT-FT-explain achieved a lower FPR (0.18) in Python while maintaining a similar F1 score.

5.2 Function Level Vulnerability Detection

5.2.1 Baseline Test Result

Zero-shot (ZS) Prompt

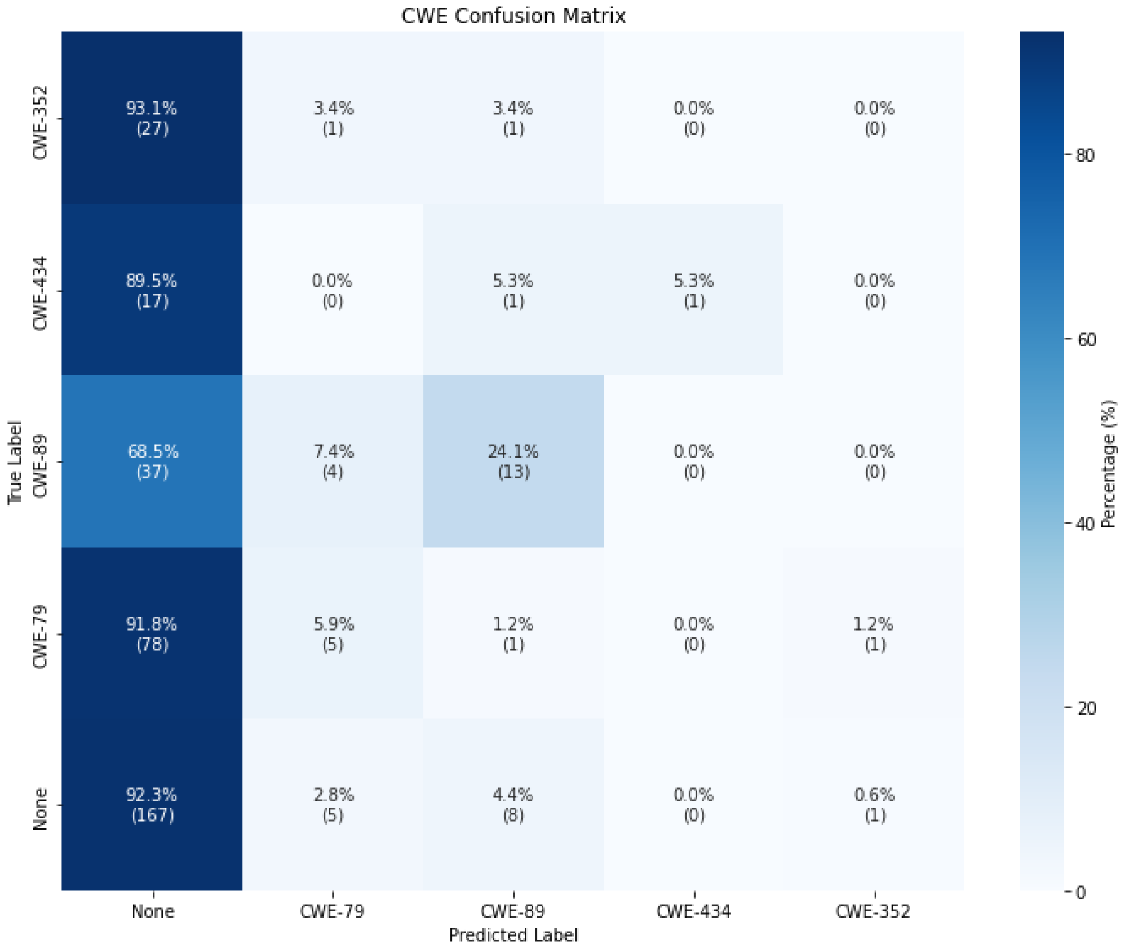

Figure 32. Confusion Matrix: Function Level Vulnerability Detection with Zero-shot Prompt

Figure 32 shows the results of the function-level vulnerability detection with a use of ZS prompt, expressed as a confusion matrix. The matrix shows a strong bias toward classifying most of the test samples as non-vulnerable code. Among the vulnerability classes, CWE-89 achieves the highest detection rate with 13 true positives (24.1% of all samples in this class). The remaining vulnerability classes show minimal detection success, with only 1-5 samples correctly identified in each CWE class. True positive of CWE-352 is remained zero in this prompt result as well.

Chain of Thought (CoT) Prompt

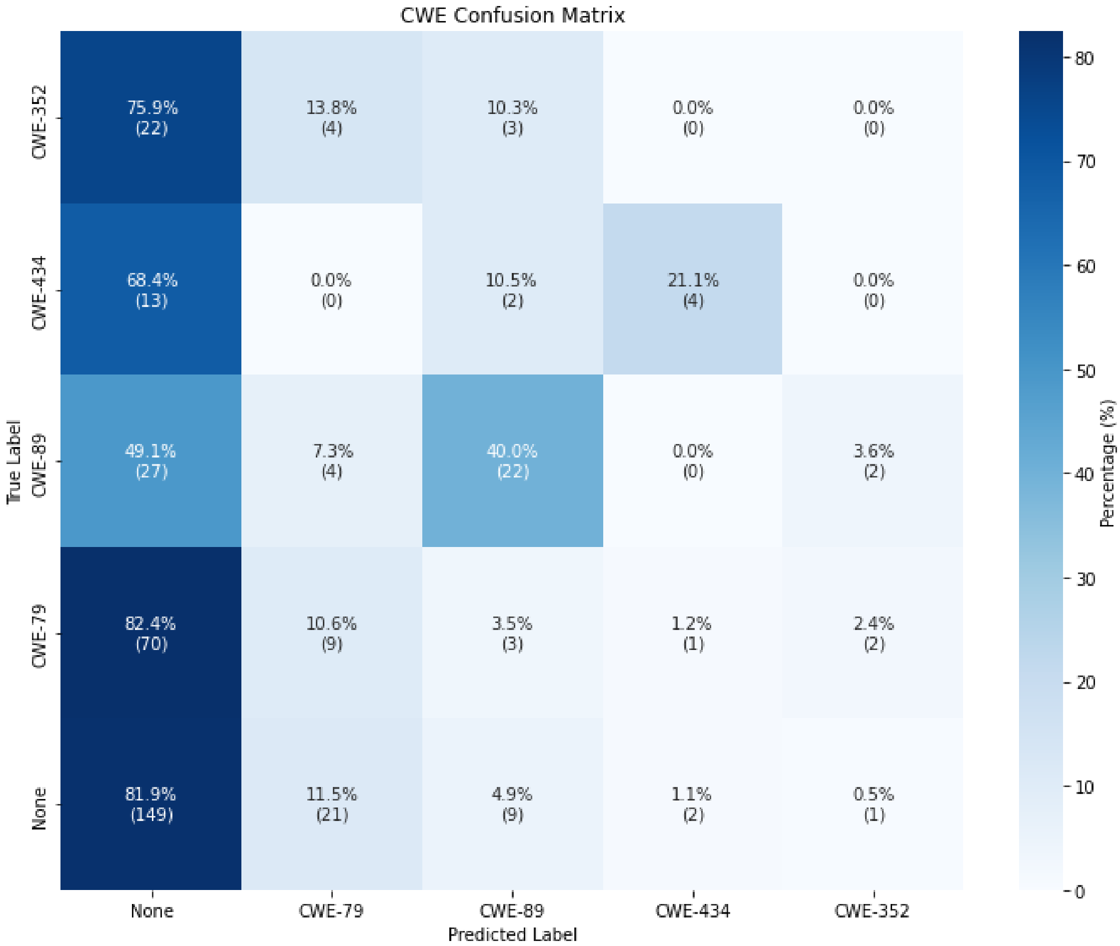

Figure 33. Confusion Matrix: Function Level Vulnerability Detection with CoT Prompt

Figure 33 shows the results of function-level vulnerability detection using CoT prompting. Similar to the ZS results, the model largely misclassifies vulnerable code samples as non-vulnerable as well. However, we observe slight improvements in certain vulnerability classes: CWE-89 shows increased true positive rates, while CWE-79 and CWE-434 show small improvements.

Few-shots (FS) Prompt

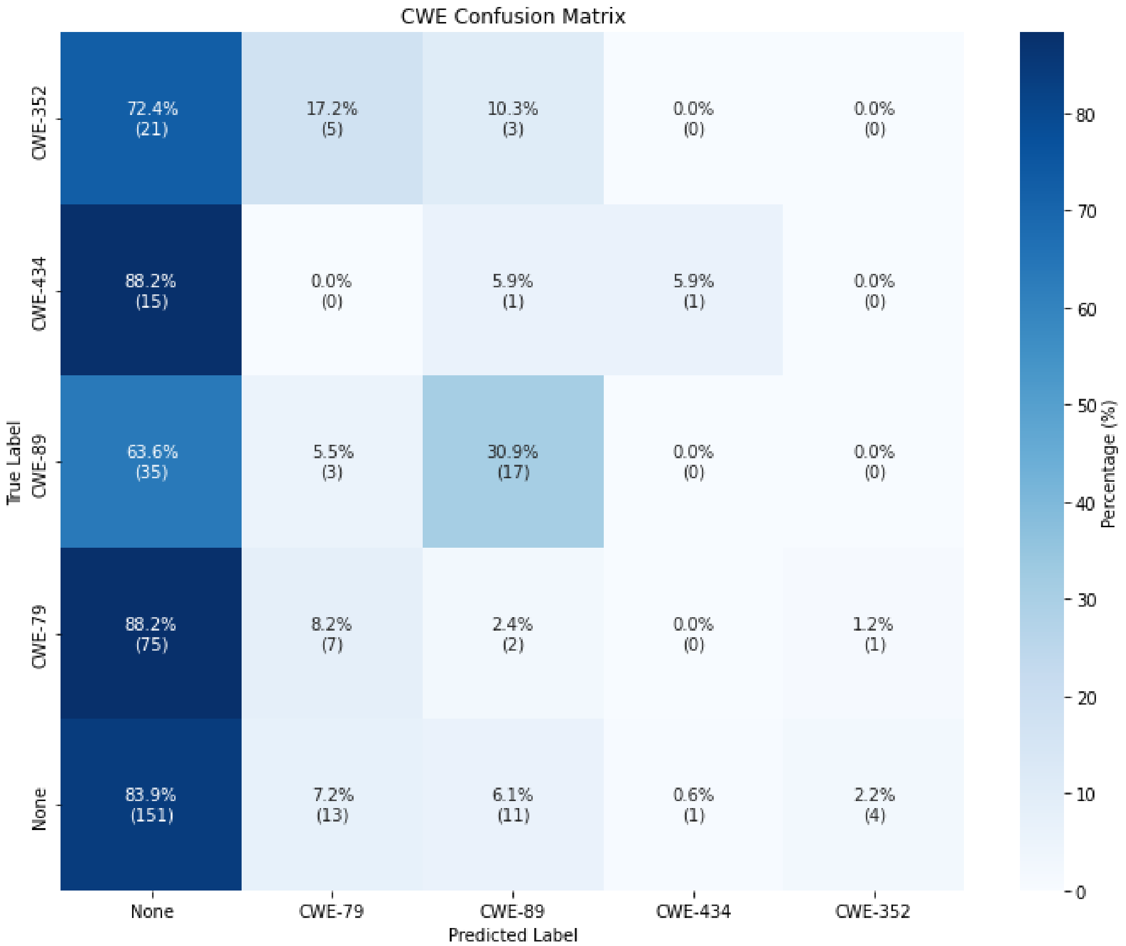

Figure 34. Confusion Matrix: Function Level Vulnerability Detection with FS Prompt

Figure 34 shows the function-level detection results using the FS prompt approach. The results show a similar classification trend to the ZS result, with one exception: CWE-89 has a higher number of true positives compared to other vulnerability classes. Of all the categories tested, CWE-89 remains the most accurately detected vulnerability, while detection performance for other CWEs remains comparable to the ZS approach.

5.2.2 Fine-tuned Model Test Result

Zero-shot (ZS) Prompt

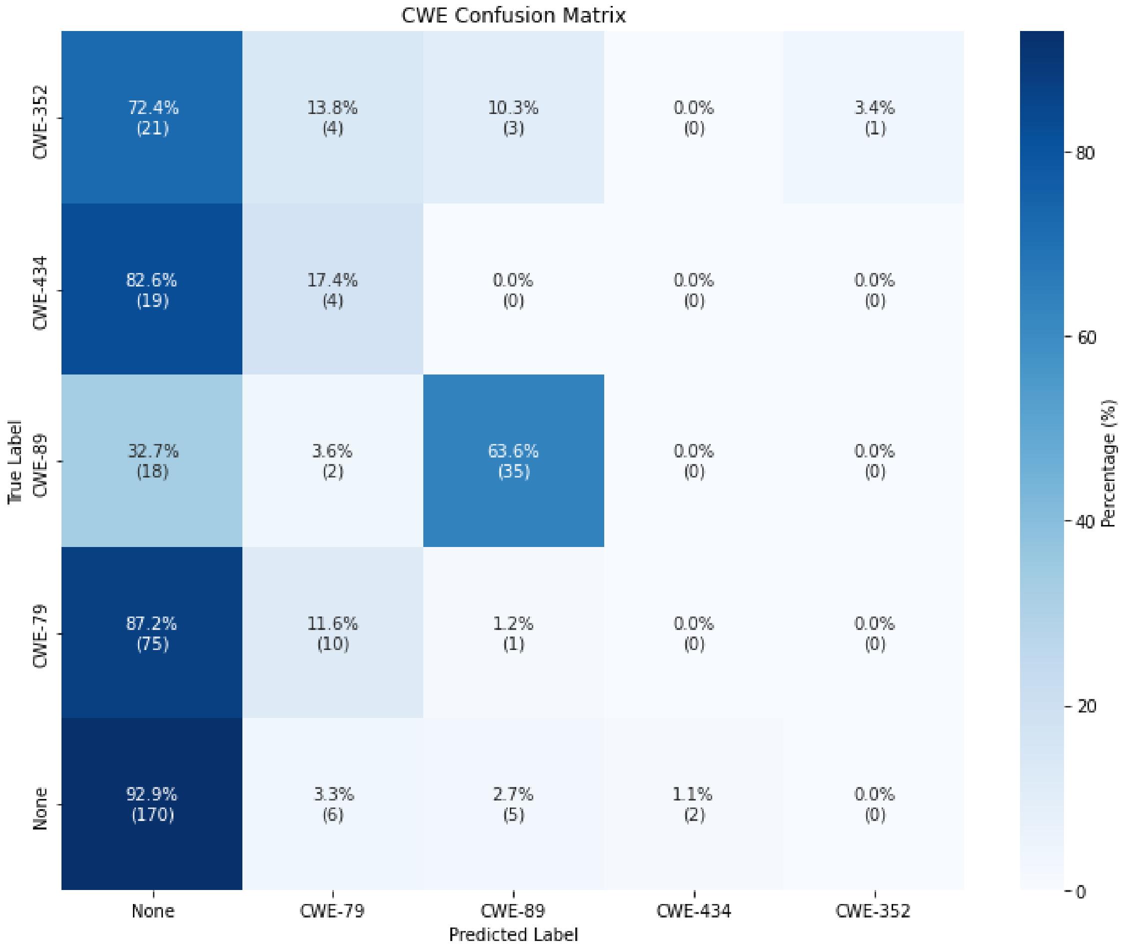

Figure 35. Confusion Matrix: Function Level Vulnerability Detection with Zero-shot Prompt with Fine-tuned Model

Figure 35 shows the ZS prompt results evaluated on the fine-tuned model. While most samples are misclassified as non-vulnerable, the model shows improved detection performance for CWE-79 and CWE-89. There is a noticeable increase in true positive classifications compared to previous configurations. However, for minor classes such as CWE-434 and CWE-352, the number of correctly classified samples is low, 0 and 1, respectively.

Chain of Thought (CoT) Prompt

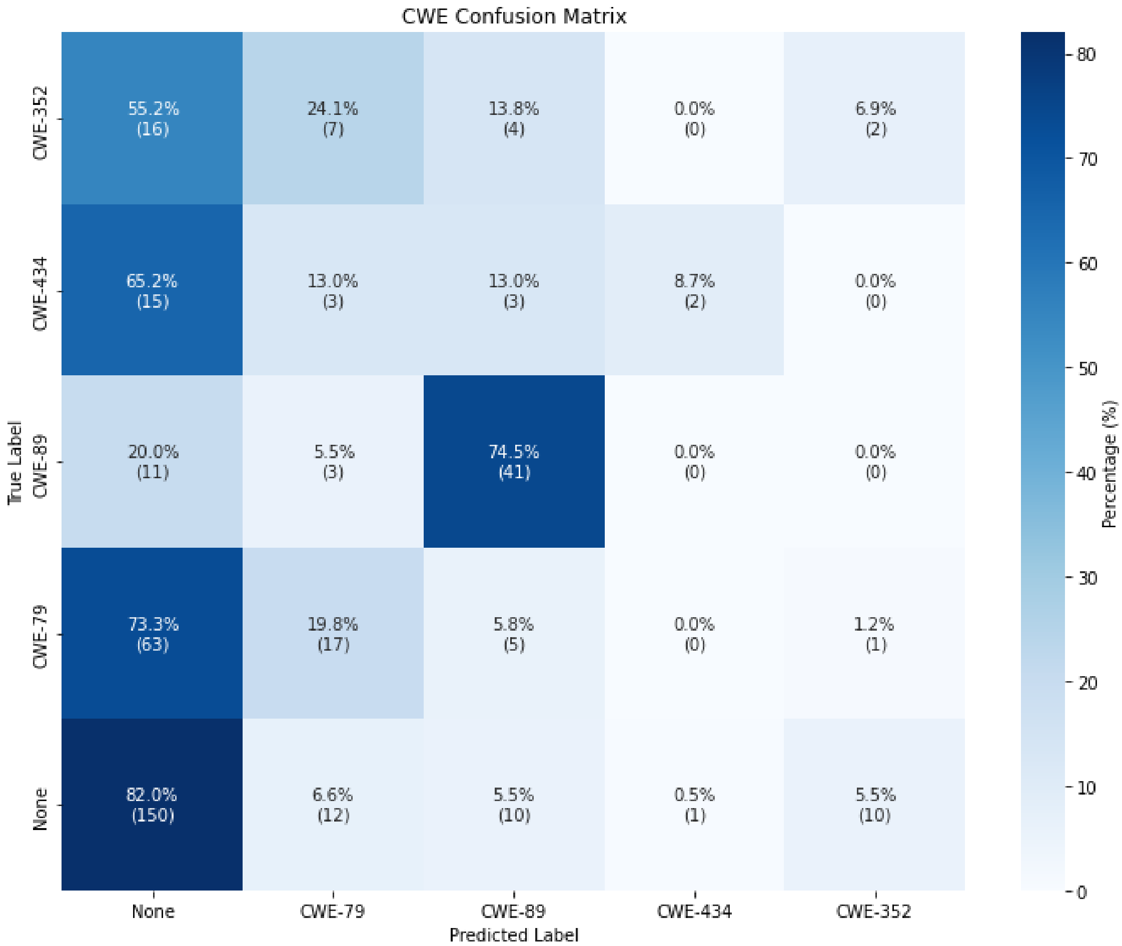

Figure 36. Confusion Matrix: Function Level Vulnerability Detection with CoT Prompt with Fine-tuned Model

Figure 36 shows the vulnerability classification results using the fine-tuned model with CoT prompting. Most samples are misclassified as non-vulnerable here still, this result shows a clear improvement in classification performance. Specifically, CWE-89 shows a consistent increase in true positives compared to previous experiments. CWE-79 also shows improved detection rates relative to the baseline CoT approach. Furthermore, minor classes, such as CWE-434 and CWE-352, achieved a few increase in true positives.

Few-shots (FS) Prompt

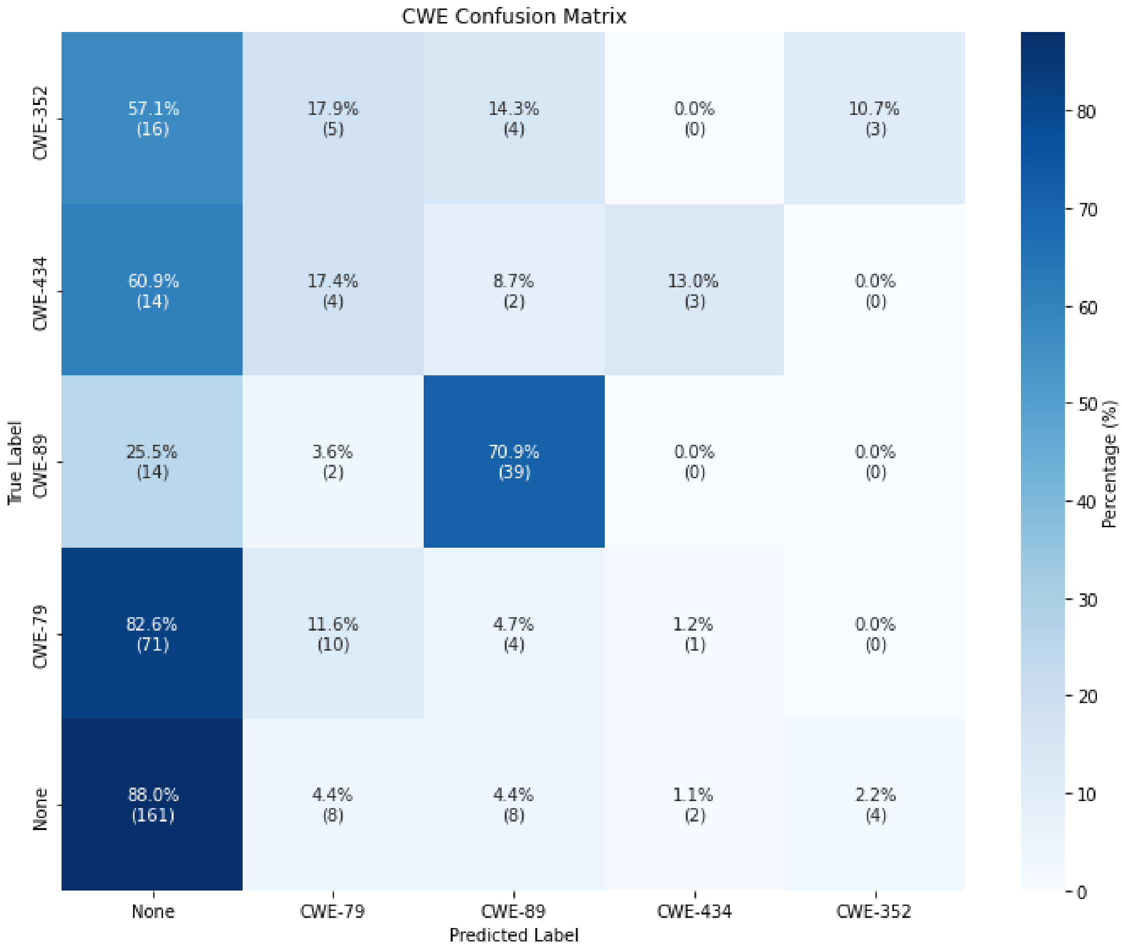

Figure 37. Confusion Matrix: Function Level Vulnerability Detection with FS Prompt with Fine-tuned Model

Figure 37 shows the FS prompt results on the fine-tuned model. The result shows that most samples are misclassified as non-vulnerable still. However, the result shows improved true positives for CWE-89 and small gains for CWE-434 and CWE-352. The model continues to struggle with distinguishing CWE-79 similar to the fine-tuned ZS result.

5.2.3 Fine-tuning with Synthesized Input

Zero-shot (ZS) Prompt

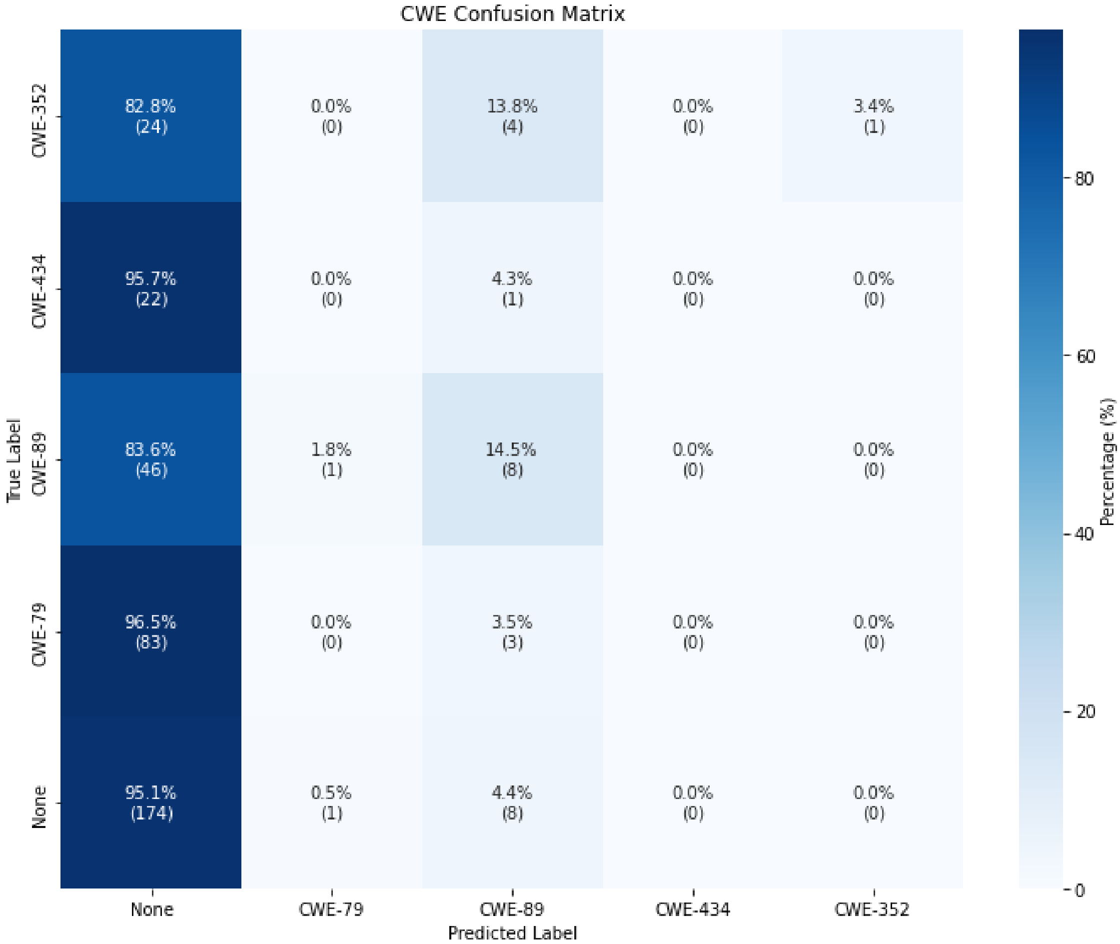

Figure 38. Confusion Matrix: Function Level Vulnerability Detection with ZS Prompt with Fine-tuned Model

Figure 38 shows the zero-shot prompt results of the reasoning-augmented fine-tuned model. The result has similar characteristic to the result observed at the file level with the same setting, with most samples misclassified as not vulnerable, and a few samples in CWE-89 and CWE-352 are classified correctly. The result shows that reasoning-augmented fine-tuning does not work well with the ZS prompt.

Chain of Thought (CoT) Prompt

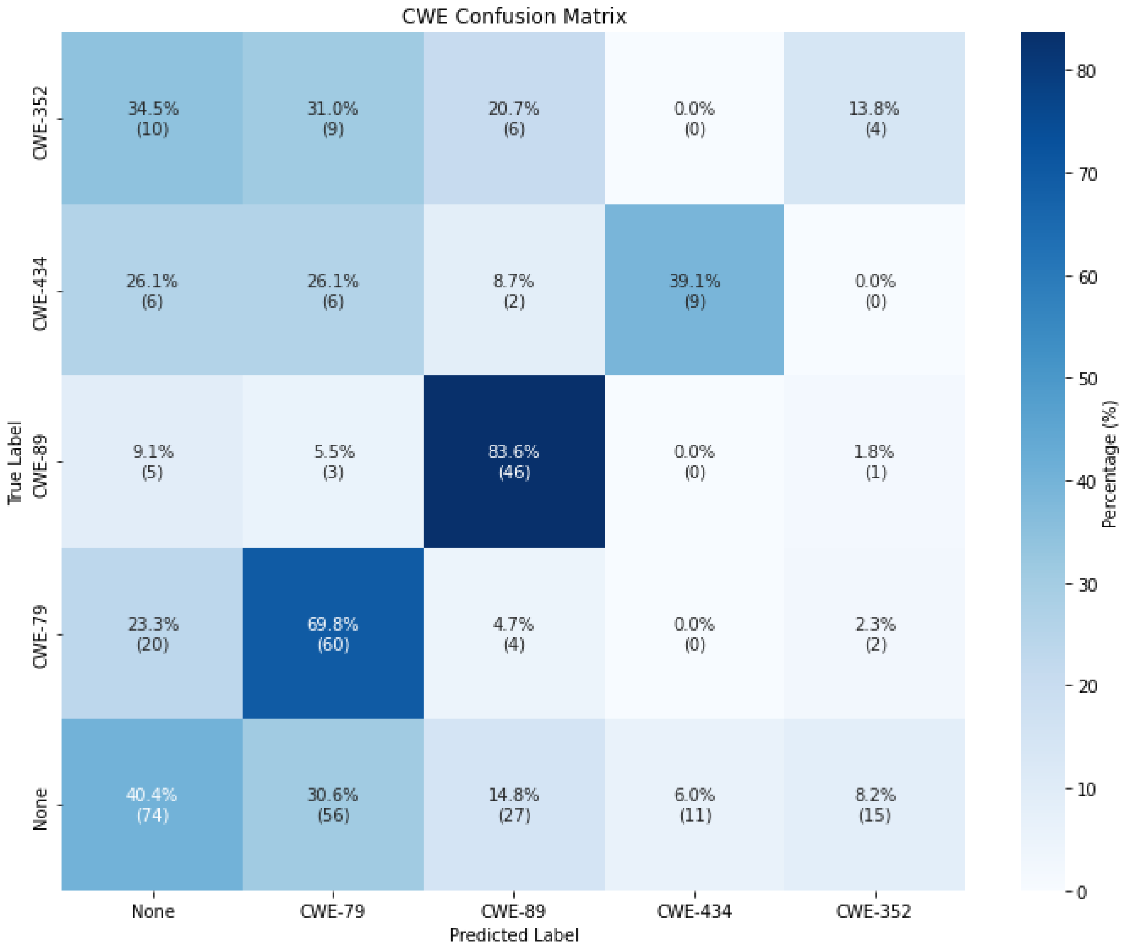

Figure 39. Confusion Matrix: Function Level Vulnerability Detection with CoT Prompt with Fine-tuned Model (Reasoning)

Figure 39 shows the CoT prompt results from the reasoning-augmented fine-tuned model. This pattern is similar to the results observed at the file level under the same setting. Misclassifying vulnerable samples as non-vulnerable has decreased significantly, and the number of true positives in vulnerable classes has increased. However, a large number of non-vulnerable samples are misclassified as various vulnerable classes. Despite this drawback, the model achieves the highest number of true positives across all vulnerability classes, including the minor classes.

Few-shots (FS) Prompt

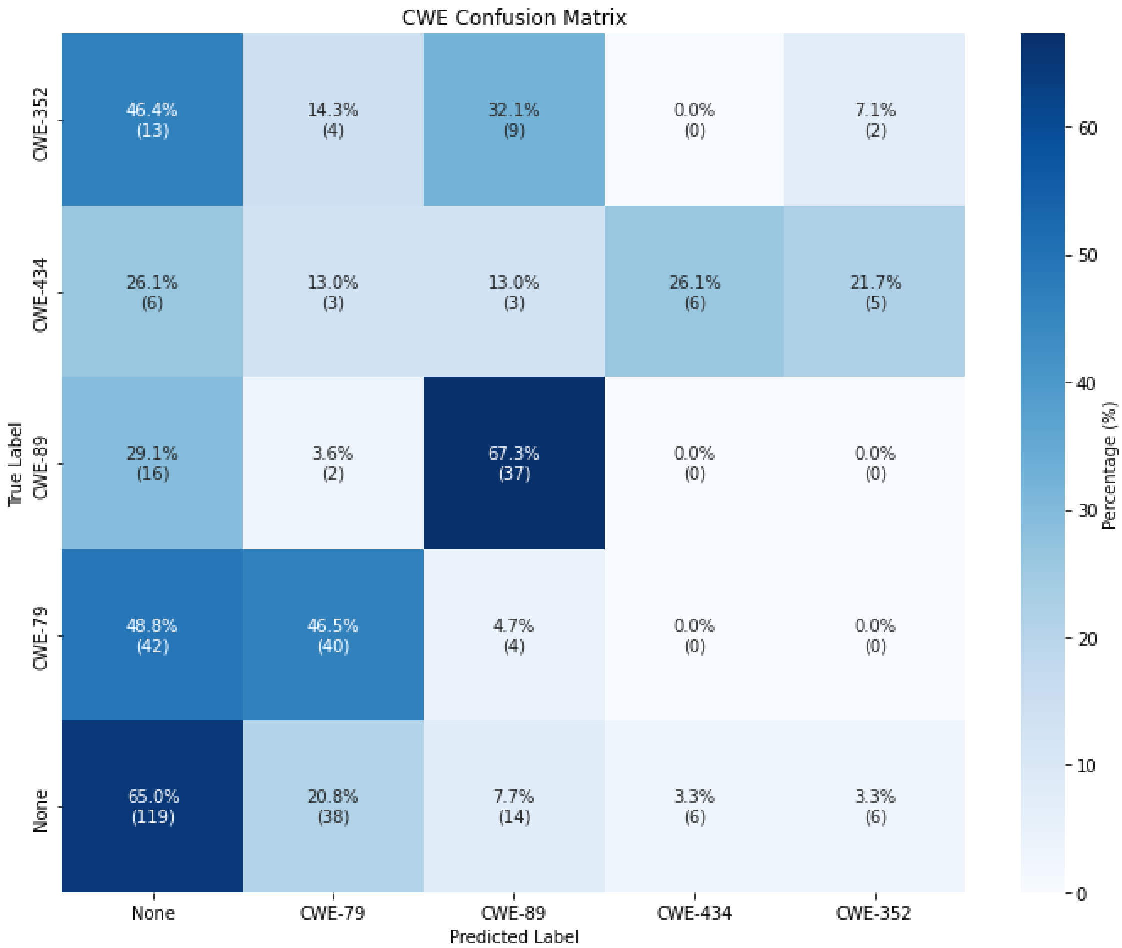

Figure 40. Confusion Matrix: Function Level Vulnerability Detection with FS Prompt with Fine-tuned Model (Reasoning)

Figure 40 shows the FS prompt results of the reasoning-enhanced fine-tuned model. The results show similar improvements to the CoT prompt. However, many vulnerable samples are still misclassified as non-vulnerable, and the misclassification of non-vulnerable code as vulnerable has also increased.

5.2.4 Statistics Overall: Comparison of Baseline and Fine-tuned Model

Table 6. Class Performance Metrics: Function Level

CWE-Class	Prompt	FNR	FPR	Acc	Precision	Recall	F1
None	ZS	0.08	0.85	0.53	0.51	0.92	0.66
	ZS-FT	0.07	0.69	0.61	0.56	0.93	0.70
	ZS-FT-explain	0.05	0.91	0.51	0.50	0.95	0.65
	CoT	0.18	0.70	0.55	0.53	0.82	0.64
	CoT-FT	0.18	0.54	0.63	0.59	0.82	0.68
	CoT-FT-explain	0.60	0.21	0.60	0.64	0.40	0.50
	FS	0.16	0.78	0.52	0.51	0.84	0.63
	FS-FT	0.12	0.60	0.63	0.58	0.88	0.70
	FS-FT-explain	0.35	0.40	0.62	0.61	0.65	0.63
CWE-79	ZS	0.94	0.04	0.76	0.33	0.06	0.10
	ZS-FT	0.88	0.06	0.76	0.38	0.12	0.18
	ZS-FT-explain	1.00	0.01	0.77	0.00	0.00	0.00
	CoT	0.89	0.10	0.72	0.24	0.11	0.15
	CoT-FT	0.80	0.09	0.75	0.40	0.20	0.27
	CoT-FT-explain	0.30	0.26	0.73	0.45	0.70	0.55
	FS	0.92	0.07	0.73	0.25	0.08	0.12
	FS-FT	0.88	0.07	0.75	0.34	0.12	0.17
	FS-FT-explain	0.53	0.16	0.75	0.46	0.47	0.46
CWE-89	ZS	0.76	0.04	0.86	0.54	0.24	0.33
	ZS-FT	0.36	0.03	0.92	0.80	0.64	0.71
	ZS-FT-explain	0.85	0.05	0.83	0.33	0.15	0.20
	CoT	0.60	0.05	0.86	0.56	0.40	0.47
	CoT-FT	0.25	0.07	0.90	0.65	0.75	0.69
	CoT-FT-explain	0.16	0.12	0.87	0.54	0.84	0.66
	FS	0.69	0.05	0.85	0.50	0.31	0.38
	FS-FT	0.29	0.06	0.91	0.68	0.71	0.70
	FS-FT-explain	0.33	0.09	0.87	0.55	0.67	0.61
CWE-434	ZS	0.95	0.00	0.95	1.00	0.05	0.10
	ZS-FT	1.00	0.01	0.93	0.00	0.00	0.00
	ZS-FT-explain	1.00	0.00	0.94	0.00	0.00	0.00
	CoT	0.79	0.01	0.95	0.57	0.21	0.31
	CoT-FT	0.91	0.00	0.94	0.67	0.09	0.15
	CoT-FT-explain	0.61	0.03	0.93	0.45	0.39	0.42
	FS	0.94	0.00	0.95	0.50	0.06	0.11
	FS-FT	0.87	0.01	0.94	0.50	0.13	0.21
	FS-FT-explain	0.74	0.02	0.94	0.50	0.26	0.34
CWE-352	ZS	1.00	0.01	0.92	0.00	0.00	0.00
	ZS-FT	0.97	0.00	0.93	1.00	0.03	0.07
	ZS-FT-explain	0.97	0.00	0.93	1.00	0.03	0.07
	CoT	1.00	0.01	0.91	0.00	0.00	0.00
	CoT-FT	0.93	0.03	0.90	0.15	0.07	0.10
	CoT-FT-explain	0.86	0.05	0.89	0.18	0.14	0.16
	FS	1.00	0.01	0.91	0.00	0.00	0.00
	FS-FT	0.89	0.01	0.92	0.43	0.11	0.17
	FS-FT-explain	0.93	0.03	0.90	0.15	0.07	0.10

Table 8. Language Performance Metrics: Function Level

Language	Method	Recall	Precision	F1	$F1_{Ma}$	FNR	FNR_{Ma}	FPR	FPR_{Ma}
PHP	ZS	0.40	0.41	0.33	0.23	0.60	0.75	0.31	0.18
	ZS-FT	0.62	0.65	0.57	0.40	0.38	0.58	0.18	0.11
	ZS-FT-explain	0.38	0.28	0.28	0.16	0.62	0.79	0.34	0.19
	CoT	0.45	0.45	0.41	0.35	0.55	0.64	0.25	0.16
	CoT-FT	0.59	0.55	0.54	0.40	0.41	0.59	0.18	0.12
	CoT-FT-explain	0.52	0.57	0.51	0.46	0.48	0.54	0.17	0.13
	FS	0.39	0.38	0.33	0.25	0.61	0.74	0.31	0.18
	FS-FT	0.64	0.64	0.61	0.51	0.36	0.51	0.16	0.10
	FS-FT-explain	0.54	0.55	0.52	0.41	0.46	0.60	0.20	0.13
JavaScript	ZS	0.52	0.69	0.38	0.15	0.48	0.80	0.49	0.19
	ZS-FT	0.49	0.33	0.34	0.14	0.51	0.80	0.50	0.20
	ZS-FT-explain	0.49	0.24	0.33	0.13	0.51	0.80	0.50	0.20
	CoT	0.46	0.39	0.37	0.15	0.54	0.82	0.50	0.21
	CoT-FT	0.50	0.52	0.40	0.30	0.50	0.68	0.46	0.19
	CoT-FT-explain	0.49	0.53	0.43	0.36	0.51	0.48	0.29	0.16
	FS	0.48	0.42	0.38	0.16	0.52	0.81	0.49	0.20
	FS-FT	0.46	0.32	0.34	0.23	0.54	0.65	0.48	0.25
	FS-FT-explain	0.47	0.49	0.48	0.29	0.50	0.59	0.40	0.23
Java	ZS	0.74	0.64	0.67	0.48	0.26	0.55	0.57	0.21
	ZS-FT	0.71	0.59	0.64	0.30	0.26	0.59	0.57	0.21
	ZS-FT-explain	0.66	0.50	0.57	0.20	0.34	0.77	0.72	0.27
	CoT	0.66	0.58	0.62	0.39	0.34	0.58	0.51	0.21
	CoT-FT	0.71	0.71	0.71	0.59	0.29	0.41	0.38	0.17
	CoT-FT-explain	0.71	0.68	0.68	0.58	0.29	0.45	0.51	0.20
	FS	0.71	0.67	0.67	0.50	0.29	0.56	0.58	0.22
	FS-FT	0.74	0.75	0.74	0.64	0.26	0.40	0.37	0.16
	FS-FT-explain	0.74	0.58	0.64	0.38	0.26	0.63	0.64	0.23
TypeScript	ZS	0.64	0.57	0.54	0.32	0.36	0.67	0.48	0.21
	ZS-FT	0.57	0.33	0.42	0.18	0.43	0.75	0.57	0.25
	ZS-FT-explain	0.57	0.35	0.44	0.15	0.43	0.75	0.62	0.25
	CoT	0.64	0.57	0.54	0.32	0.36	0.67	0.48	0.21
	CoT-FT	0.57	0.55	0.51	0.30	0.43	0.70	0.49	0.23
	CoT-FT-explain	0.43	0.37	0.39	0.25	0.57	0.72	0.44	0.25
	FS	0.57	0.33	0.42	0.18	0.43	0.75	0.57	0.25
	FS-FT	0.57	0.33	0.42	0.18	0.43	0.75	0.57	0.25
	FS-FT-explain	0.64	0.57	0.54	0.32	0.36	0.67	0.48	0.21
Ruby	ZS	0.54	0.42	0.47	0.23	0.46	0.74	0.58	0.26
	ZS-FT	0.58	0.57	0.51	0.26	0.42	0.73	0.56	0.25
	ZS-FT-explain	0.58	0.38	0.46	0.19	0.42	0.77	0.57	0.25
	CoT	0.65	0.57	0.60	0.37	0.35	0.56	0.29	0.16
	CoT-FT	0.65	0.65	0.65	0.40	0.32	0.48	0.31	0.16
	CoT-FT-explain	0.46	0.47	0.43	0.22	0.52	0.63	0.32	0.21
	FS	0.54	0.51	0.50	0.31	0.46	0.61	0.31	0.19
	FS-FT	0.58	0.49	0.51	0.26	0.42	0.73	0.51	0.23
	FS-FT-explain	0.54	0.50	0.51	0.30	0.46	0.64	0.36	0.21
Python	ZS	0.58	0.38	0.46	0.18	0.42	0.77	0.63	0.26
	ZS-FT	0.58	0.38	0.46	0.18	0.42	0.77	0.63	0.26

Continued on next page

Table 8 – continued from previous page

Language	Method	Recall	Precision	F1	$F1_{Ma}$	FNR	FNR_{Ma}	FPR	FPR_{Ma}
	ZS-FT-explain	0.67	0.57	0.56	0.30	0.33	0.69	0.56	0.22
	CoT	0.50	0.36	0.42	0.17	0.50	0.80	0.65	0.29
	CoT-FT	0.46	0.39	0.42	0.21	0.54	0.77	0.60	0.28
	CoT-FT-explain	0.42	0.39	0.40	0.17	0.57	0.78	0.59	0.29
	FS	0.54	0.37	0.44	0.18	0.46	0.78	0.64	0.28
	FS-FT	0.58	0.38	0.46	0.18	0.42	0.77	0.63	0.26
	FS-FT-explain	0.62	0.39	0.48	0.19	0.38	0.75	0.63	0.25

Table 9. Overall Performance Metrics: Function Level Vulnerability Detection

Prompt	Recall	Precision	F1	$F1_{Ma}$	FNR	FNR_{Ma}	FPR	FPR_{Ma}
ZS	0.51	0.46	0.40	0.24	0.49	0.75	0.43	0.19
ZS-FT	0.55	0.55	0.49	0.33	0.43	0.66	0.35	0.16
ZS-FT-explain	0.49	0.37	0.35	0.18	0.51	0.77	0.45	0.19
CoT	0.50	0.43	0.44	0.31	0.50	0.69	0.38	0.18
CoT-FT	0.56	0.53	0.51	0.38	0.44	0.62	0.30	0.15
CoT-FT-explain	0.51	0.54	0.50	0.45	0.49	0.51	0.19	0.13
FS	0.48	0.41	0.40	0.25	0.52	0.74	0.41	0.19
FS-FT	0.58	0.53	0.51	0.39	0.42	0.61	0.32	0.15
FS-FT-explain	0.54	0.52	0.53	0.43	0.46	0.58	0.25	0.14

Table 9 shows the overall scores for the function-level input. Similar to the file-level input results, CoT achieves the highest F1 score among the unmodified model settings, having weighted F1 score of 0.44. However, ZS and FS demonstrate comparable performance in this setting.

Most configurations of the fine-tuned models outperform their unmodified model results, except for ZS-FT-explain. FS-FT-explain achieves the highest weighted F1 score, while CoT-FT-explain achieves the highest macro F1 score. CoT-FT-explain obtains a high number of true positives in vulnerable classes, while FS-FT-explain performs well in non-vulnerable classes and captures subtle differences in vulnerable ones. FS-FT achieves the lowest weighted FNR (0.42), and CoT-FT-explain achieves the lowest macro FNR (0.51), and the lowest FPR (0.19 weighted and 0.13 macro).

Table 6 evaluates prompt techniques across vulnerability classes for baseline and fine-tuned models. Baseline techniques show clear performance differences: ZS achieves the highest F1 scores for non-vulnerable (0.60) and CWE-434 (0.12), while CoT performs best for CWE-79 (0.35) and CWE-89 (0.43). FS performs best for CWE-352 (0.17). Fine-tuning reduces FNR and FPR for the non-vulnerable and CWE-89 classes for all techniques.

However, CWE-79 performance degrades after fine-tuning. CoT-FT improves F1 scores for most classes except CWE-79. It is also notable that reason-augmented fine-tuning improves performance significantly across all classes with CoT propmt and FS propmt. The most substantial gain is observed for CWE-79, where the F1 score increases significantly (F1 CoT: 0.15 $\rightarrow$ CoT-FT-explain: 0.55).

Table 8 shows language specific differences in different prompt types. The best-performing combinations among the unmodified model prompts are: ZS in Java (F1 score of 0.67), CoT in Java (F1 score of 0.62), and FS in Java (F1 score of 0.67). Overall, Java appears to be the most favorable language across prompt types. On the other hand, the lowest performance is observed with the PHP ZS and FS prompts (both F1: 0.33) and the JavaScript CoT prompt (F1: 0.37).

Java and PHP consistently benefit from fine-tuning, with PHP showing significant F1 improvements (ZS: 0.33 $\rightarrow$ ZS-FT: 0.57). In contrast, JavaScript and TypeScript show performance degradation (JavaScript ZS-FT F1 drops: 0.38 $\rightarrow$ 0.34). CoT-FT performs the best in Ruby (F1: 0.65).

Fine-tuning generally improves language-level metrics, lowering FPR and FNR in cases such as PHP ZS (FNR: 0.60 $\rightarrow$ 0.38) and Java CoT (FNR: 0.34 $\rightarrow$ 0.29). However, there are trade-offs, such as increased FPR in Ruby CoT-FT (0.29 $\rightarrow$ 0.31) and increased FNR/FPR in TypeScript ZS-FT (FNR: 0.36 $\rightarrow$ 0.43 and FPR: 0.48 $\rightarrow$ 0.57). These inconsistencies shows that the effect of fine-tuning varies by language and technique.

6. Discussion

This chapter discusses the key findings presented in Section 5 and evaluates their implications in relation to the research questions presented in Section 1. And then we compare the results with those of similar studies. Finally, the methodological limitations of this work are critically examined to frame the scope of the conclusions and suggest directions for future research.

6.1 RQ1: How effectively can LLMs detect vulnerabilities across different programming languages?

Table 10. File and Function Level Input: F1 Score Rankings by Language (CoT)

File Level Input					Function Level Input				
Rank	Language	F1	FNR	FPR	Rank	Language	F1	FNR	FPR
1	JavaScript	0.51	0.44	0.41	1	Java	0.62	0.34	0.51
2	TypeScript	0.47	0.46	0.36	2	Ruby	0.60	0.35	0.29
2	Ruby	0.47	0.42	0.38	3	TypeScript	0.54	0.36	0.48
4	Java	0.46	0.46	0.38	4	Python	0.42	0.50	0.65
5	PHP	0.43	0.57	0.27	5	PHP	0.41	0.55	0.25
6	Python	0.41	0.52	0.42	6	JavaScript	0.37	0.54	0.50

Table 10 shows F1 score rankings of languages with CoT prompt style, which is prioritized due to its superior performance in most file and function level classifications. These comparisons are based on the unmodified version of the model, without any fine-tuning.

The comparative analysis reveals significant variations in the model's ability to identify vulnerabilities across different programming languages and input granularity levels:

- **JavaScript** performs best at the file level but worst at the function level. This suggests that JavaScript vulnerability detection significantly benefits from broader contextual information, such as function interactions, variable scopes, and module dependencies.
- **TypeScript** and **Ruby** consistently rank in positions two and three in both file and function level classifcations. This stability suggests that these languages have consistent vulnerability patterns regardless of input granularity.
- **Java** achieves the highest performance in function-level classification, yet it ranks

fourth in file-level detection. This is an opposite pattern compared to JavaScript, which suggests that Java's strongly typed structure may provide clearer vulnerability signals in isolated functions. However, additional context may introduce confounding factors.

- **PHP** consistently ranks fifth in both evaluation contexts. It has the lowest FPR (file: 0.27 and function: 0.25) and the highest FNR (file: 0.57 and function: 0.55). These results suggest that the model has high precision but low recall when classifying PHP code. The model rarely misclassifies other classes as true class (low false positives), but often fails to correctly identify true class (high false negatives).
- **Python** ranks lowest in file-level analysis and fourth in function-level classificatoin. Python samples showed the highest FPR values (file: 0.42 and function: 0.65), suggesting that the model often misclassifies Python code samples as different class.

These cross-language performance differences highlight important factors to consider when developing vulnerability detection models. The findings suggest that input granularity interacts differently with various language granularities, and that the trade-off between precision and recall varies significantly across programming languages.

6.2 RQ2: How does detection performance vary between file-level and function-level analysis?

Table 11. Comparison of Overall Results in Different Context Granularity

Level & Prompt	F1	$F1_{Ma}$	FNR	FNR_{Ma}	FPR	FPR_{Ma}
File ZS	**0.45**	**0.29**	0.52	**0.71**	**0.35**	**0.17**
Func ZS	0.40	0.24	**0.49**	0.75	0.43	0.19
File CoT	**0.46**	0.30	0.54	0.69	**0.30**	**0.17**
Func CoT	0.44	**0.31**	**0.50**	0.69	0.38	0.18
File FS	0.40	**0.27**	0.58	**0.73**	**0.37**	0.19
Func FS	0.40	0.25	**0.52**	0.74	0.41	0.19

Table 12. CWE-Class Input: F1 Score Rankings (CoT)

File Level Input					**Function Level Input**				
Rank	CWE-Class	F1	FNR	FPR	Rank	CWE-Class	F1	FNR	FPR
1	None	0.58	0.39	0.49	1	None	0.64	0.18	0.70
2	CWE-89	0.43	0.50	0.17	2	CWE-89	0.47	0.60	0.05
3	CWE-79	0.35	0.72	0.11	3	CWE-434	0.31	0.79	0.01
4	CWE-352	0.09	0.93	0.04	4	CWE-79	0.15	0.89	0.10
5	CWE-434	0.07	0.92	0.03	5	CWE-352	0.00	1.00	0.01

Comparison of results in different context granularity (Table 11) shows that file-level input results slightly better vulnerability detection performance than function-level input in case of use with unmodified model. Under the ZS and CoT prompt strategies, file-level context results in modest improvements in F1 scores ranging from 0.02 to 0.05, suggesting that broader code context enhances the model's ability to analyze cross-function data flow, variable scope, and component interactions. These factors are particularly important for detecting vulnerabilities that involve cross-function logic, such as improper input sanitization or insecure data handling.

In addition to F1 scores, file-level input also correlates with a lower FPR, reducing the likelihood of false vulnerability alerts. However, this approach has a slightly higher FNR, meaning that excessive context can occasionally introduce ambiguity into the classification. This represents a fundamental trade-off between precision and recall.

In contrast, function-level input achieves a lower FNR, likely because concise code snippets allow for more straightforward classification. However, this granularity increases the FPR, resulting in more false positives. The choice between file-level and function-level input should depend on the specific application requirements.

For scenarios where minimizing false positives is critical, such as developer review tools, file-level inputs can be beneficial. On the other hand, function-level inputs may be more appropriate for scenarios where analysis speed is prioritized, such as CI/CD pipelines since inputs are substantially smaller than file-level inputs.

Table 12 shows the F1 score rankings for each CWE class across file- and function-level classifications. This comparison shows patterns in the model's ability to detect specific vulnerability types under different code granularities. See Section 7.1 for code samples and model outputs (CoT prompt).

- **Non-vulnerable code** and **CWE-89 (SQL Injection)** consistently rank as the top two classes across both granularity levels. This stability suggests that the distinctive patterns of these classes remain equally detectable regardless of whether the full file context or isolated function context is provided. For SQL injection vulnerabilities, this may mean that the typical query construction patterns can be identified even in limited code contexts. Figure 41 shows a function-level classification example.
- **CWE-79 (Cross-site Scripting)** demonstrates stronger performance in file-level analysis (ranked 3rd and F1: 0.35) compared to function-level analysis (ranked 4th and F1: 0.15). This suggests that XSS vulnerabilities benefit from broader contextual information. This is potentially because their detection often requires understanding

how user inputs traverse through multiple functions before reaching output rendering points. A sample of the file's code and output is shown in Figure 42.

- **CWE-352 (CSRF)** is difficult to classify, as shown by its low F1 scores at the function and file levels. However, it shows slightly better relative performance at the file level (ranked fourth with an F1 score of 0.09) than at the function level (ranked fifth with an F1 score of 0.0). This slight improvement suggests that, while still limited, file-level input can occasionally provide enough contextual information to support correct classification. CSRF vulnerabilities often span multiple functions and rely on broader context, such as request validation and session management logic. Figure 43 shows a relevant example.
- **CWE-434 (Unrestricted Upload of File)** shows the opposite pattern, performing better at the function level (ranked 3rd with an F1 score of 0.09) than at the file level (ranked 4th with an F1 score of 0.31). This suggests that this vulnerabilities may be more distinctive when isolated within individual functions. A sample function and its output are shown in Figure 44.

6.3 RQ3: Does fine-tuning reduce false-positive and false-negative rates in multi-class vulnerability classification?

Table 13. Comparison of Overall Results in Different Fine-tuning Methods

Level & Prompt	F1	$F1_{Ma}$	FNR	FNR_{Ma}	FPR	FPR_{Ma}
File ZS	0.45	0.29	0.52	0.71	**0.35**	**0.17**
File ZS-FT	**0.47**	**0.35**	0.50	**0.66**	**0.35**	**0.17**
File ZS-FT-explain	0.35	0.15	**0.49**	0.79	0.49	0.20
File CoT	0.46	0.30	0.54	0.69	0.30	0.17
File CoT-FT	**0.46**	**0.33**	**0.52**	0.68	0.35	0.17
File CoT-FT-explain	0.38	**0.33**	0.58	**0.60**	**0.18**	**0.15**
File FS	0.40	0.27	0.58	0.73	0.37	0.19
File FS-FT	0.40	**0.31**	0.60	**0.69**	**0.32**	0.18
File FS-FT-explain	**0.49**	0.30	**0.48**	0.70	0.33	**0.16**
Func ZS	0.40	0.24	0.49	0.75	0.43	0.19
Func ZS-FT	**0.49**	**0.33**	**0.43**	**0.66**	**0.35**	**0.16**
Func ZS-FT-explain	0.35	0.18	0.51	0.77	0.45	0.19
Func CoT	0.44	0.31	0.50	0.69	0.38	0.18
Func CoT-FT	**0.51**	0.38	**0.44**	0.62	0.30	0.15
Func CoT-FT-explain	0.50	**0.45**	0.49	**0.51**	**0.19**	**0.13**
Func FS	0.40	0.25	0.52	0.74	0.41	0.19
Func FS-FT	0.51	0.39	**0.42**	0.61	0.32	0.15
Func FS-FT-explain	**0.53**	**0.43**	0.46	**0.58**	**0.25**	**0.14**

Table 13 presents a comprehensive comparison of various fine-tuning methods across different prompt types. The results clearly demonstrate that fine-tuning substantially improves model performance across all configurations. In particular, the improvement observed with function-level inputs is significant, as these inputs have a much larger training dataset (approximately 9,000 samples) than file-level inputs (approximately 1,900 samples).

In terms of performance metrics, the highest weighted F1 scores were achieved using the FS prompt configuration for both input types: 0.49 for file-level (FS-FT-explain) and 0.53 for function-level (FS-FT-explain). However, when evaluated based on macro F1 scores, the CoT prompt yielded superior results: 0.33 for file-level (CoT-FT) and 0.45 for function-level (CoT-FT-explain). As shown in Table 13 and the confusion matrices in Section 5, this difference in performance is due to the strengths of each approach. CoT prompts are better at identifying vulnerable code samples, while FS prompts are better at classifying non-vulnerable instances with similar characteristics.

Both prompting strategies effectively reduced false positives and false negatives. The most significant improvements were observed in the function-level input setting when using CoT prompts with the reasoning-augmented fine-tuned model. This configuration achieved the lowest error rates (FNR: 0.49, FNR_{Ma}: 0.51, FPR: 0.19, FPR_{Ma}: 0.13). Similar patterns were observed in the file-level setting, where the CoT prompt with reasoning-augmented fine-tuning produced the lowest macro FNR and FPR (FNR_{Ma}: 0.60, FPR: 0.18, FPR_{Ma}: 0.15). These results suggest that combining reasoning-augmented fine-tuning with CoT prompting, which enables the LLM to process information logically, improves the model's ability to accurately classify code vulnerabilities.

An interesting observation was found regarding models that were fine-tuned with reasoning-augmented input but evaluated using ZS prompts. In these configurations (File ZS-FT-explain and Func ZS-FT-explain), the models' classification accuracy was significantly lower. This suggests a potential mismatch between the reasoning-oriented training approach and direct classification in zero-shot scenarios. It also highlights the importance of proper prompt training to optimize model performance.

6.4 Comparison to Related Papers

Due to the limited number of studies that explore code vulnerability detection using LLMs on multi-class, multi-language datasets, it is somewhat challenging to directly compare our results to existing work. Therefore, we focus on papers that utilize the same dataset as ours (CVEFixes) and analyze the similarities and differences in approach and results.

This paper [30] fine-tuned four models, including encoder-decoder, encoder-only, and decoder-only architectures. Among these, the decoder model used was DeepSeek-Coder (1.3B parameters), and it was evaluated on the CVEFixes dataset, classifying function-level code snippets from multiple programming languages into binary classes (vulnerable or non-vulnerable). Their fine-tuning and evaluation approach is similar to ours, including language-specific performance analysis.

One important difference is how well different languages work, especially JavaScript. In the binary classification setting, JavaScript ranked second with an F1 score of 0.65. In contrast, in our multi-class classification setting, JavaScript ranked last, with our best score at the function level being only 0.48. However, Java performed similarly in both studies, ranking first in the first study (F1: 0.69) and among the best in the second study (F1: 0.74). This suggests that Java vulnerability classification may be easier for LLMs to classify, regardless of the setup. Overall, their results show that smaller LLMs can still be competitive in certain languages. However, it's important to be cautious when directly comparing metrics because of differences in class size.

This paper [31] studied LLM performance in detecting vulnerabilities across multiple programming languages using a combined dataset of CVEFixes, CWE-snippets, and JVE. Their dataset, which contains 378 test samples at the file level, is comparable in size to our file-level test set (492 samples). They evaluated five languages: Python, C, C++, Java, and JavaScript using zero-shot and few-shot prompts that instructed models to output only the class label. This differs from our approach, which required models to provide both reasoning and classification. Their evaluation included both binary and multi-class (CWE Top 25) classification tasks using models such as GPT-3.5-Turbo, GPT-4 Turbo, Gemini 1.5 Pro, and CodeLlama. Their results suggest that GPT-4o performs best on Python, C++, and Java, while Gemini performs the best in Python, and GPT-4 Turbo is particularly effective on C and JavaScript. These findings indicate that each model has language-specific strengths.

Another study [5] studied LLMs for vulnerability detection in binary class, with function level inputs, using the same CVEFixes dataset. One of the models they used is GPT-3.5-Turbo, which has about 100 billion parameters. Despite the size of the model, their zero-shot approach had limited effectiveness, with precision and recall falling below 15% and 5%, respectively, for both vulnerability classes. In contrast, our method uses a much smaller LLM with about 16 billion parameters and targets multi-class classification task. Despite the increased complexity, our approach achieves significantly better performance.

This comparison suggests two important observations. First, larger model size alone does not necessarily lead to better performance in vulnerability detection tasks. Second, model effectiveness is highly dependent on methodological choices, such as the use of task-specific prompting strategies and fine-tuning procedures. Taken together, these results demonstrate that smaller models, when guided by targeted design decisions based on domain knowledge, can outperform much larger models with some enhancement.

In summary, these studies suggest that while model size influences performance, it is not the sole determinant. Model architecture, prompt design, and language-specific characteristics also play critical roles. Our findings highlight ongoing gaps in the field, including the lack of standardized benchmarks for multi-class, multi-language vulnerability detection and the underexplored impact of input granularity.

6.5 Limitations

This section outlines the limitations of our study.

First, the dataset used in this study was highly imbalanced. While the dataset contained mainly examples in PHP and JavaScript, other languages such as Python, Java, Ruby, and TypeScript were underrepresented. This language imbalance may have affected the model's ability to generalize across programming languages when fine-tuning the model. Although we did not observe significant overfitting characteristics during fine-tuning, having a more balanced dataset would likely improve performance better, especially for underrepresented languages.

Second, we were only able to fine-tune a relatively small parameter model (14B) due to time and computational resource limitations. Our initial attempts to fine-tune a state-of-the-art LLM with relatively larger parameter size were limited by frequent CUDA out-of-memory errors. As a result, we resorted to using a smaller and more resource efficient model. This limitation may have limited the potential performance gains from fine-tuning.

Third, the evaluation in this study was limited to comparing the fine-tuned model to its base version. We did not include comparisons with other state-of-the-art LLMs. A more comprehensive evaluation using multiple models would provide a deeper understanding of the comparative strengths and weaknesses of the Phi-4 model. Such a comparison could reveal how the parameter size and the model's specialty affect the effectiveness of fine-tuning in detail.

Finally, our study did not compare the performance of fine-tuned LLMs with traditional static/dynamic analysis tools because we could not find a tool that supports all the languages used in the dataset. Previous studies [11, 32] suggest that LLMs can outperform traditional tools, depending on the model, experimental setup, and CWE class. Therefore we encourage readers to use our findings to improve LLM-based vulnerability detection rather than for direct comparison with conventional methods.

6.6 Future Work

Building on the limitations discussed above, the following directions are proposed for future research on using LLMs for code vulnerability detection.

6.6.1 Balanced Dataset Generation

As discussed, the dataset used in this study was highly imbalanced. Future work should focus on collecting or synthesizing a more balanced dataset, especially including more samples from underrepresented languages. A more diverse dataset would allow for a fairer evaluation of cross-language generalization and help prevent overfitting during fine-tuning.

6.6.2 Exploring Alternative Models

In this study, we fine-tuned a single model, which is made for generic purposes to assess performance improvements. Future research should experiment with a wider range of models, including those with varying parameter sizes and training objectives. This would help identify which architectures are most suitable for vulnerability detection tasks and how different models respond to fine-tuning.

6.6.3 Language-Specific Fine-Tuning and Attribution Analysis

While our dataset included multiple programming languages, we did not isolate the performance contribution of each individual language. Future work could involve fine-tuning separate models on single language subsets to investigate how language-specific training affects performance. This may lead to better language specific optimization strategies and insights into cross-language transferability.

7. Conclusion

This thesis systematically investigated the ability of LLMs to detect code vulnerabilities across multiple programming languages and classification categories. We conducted evaluations at both the file level and the function level, using a multi-class classification setting. The study focused on six widely used programming languages: PHP, JavaScript, Java, TypeScript, Ruby, and Python, and four major CWE types, such as CWE-79, CWE-89, CWE-434, CWE-352, and non-vulnerable code retrieved from the CVEfixes dataset.

To address the limitations of previous research, we compared 3 prompting strategies, including zero-shot, few-shot, and chain-of-thought, to assess baseline performance. We then fine-tuned the LLM using the QLoRA method to further optimize detection accuracy and minimize false positives and false negatives. The results demonstrate that fine-tuning and prompt engineering improve the model's ability to generalize across languages and vulnerability types.

7.1 Contribution

- **Multilingual Vulnerability Detection Enhancement**: This study provides a comprehensive evaluation of LLM-based code vulnerability detection across six programming languages. By benchmarking performance at both the file and function levels and under different prompt styles, the analysis reveals language-specific trade-offs in detection performance. These trade-offs vary depending on input granularity and prompting strategy. The findings offer insights into improving cross-language performance of LLMs and underscore the importance of adapting detection strategies to the unique characteristics of each programming language.
- **Granularity Analysis for Improved Precision**: The study demonstrates that the granularity of code input significantly affects detection performance. Analysis of unmodified model results suggests that file-level analysis captures broader context and reduces false positives by identifying vulnerabilities that span multiple functions. However, this comes at the cost of increased false negatives. In contrast, function-level analysis enhances recall by focusing on localized patterns but tends to produce more false positives.
- **Fine-Tuning Strategies for Error Reduction**: This study investigates fine-tuning strategies aimed at reducing classification errors in a multi-class setting. Fine-tuning improved the model's performance across different configurations. Improvements

were observed with function-level inputs more than with file-level inputs, likely because a larger training dataset was available for the former input size. Different prompt types exhibited varying strengths. CoT prompts are better at identifying vulnerable code, while FS prompts are more effective at classifying non-vulnerable instances with similar characteristics. Most prompt styles reduced false positives and false negatives. The best results came from combining reasoning-augmented fine-tuning with CoT prompting. This combination helped the model process information more logically.

References

[1] CVE Details. *Number of common IT security vulnerabilities and exposures (CVEs) worldwide from 2009 to 2024 YTD*. https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures/. Accessed: 2024-10-06. 2024.

[2] Juyong Jiang et al. "A Survey on Large Language Models for Code Generation". In: *arXiv preprint arXiv:2406.00515* (2024). URL: https://arxiv.org/abs/2406.00515.

[3] Jing Liu et al. *Can LLM Generate Regression Tests for Software Commits?* 2025. DOI: 10.48550/arXiv.2501.11086. arXiv: 2501.11086 [cs.SE]. URL: https://arxiv.org/abs/2501.11086.

[4] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. "RepairAgent: An Autonomous, LLM-Based Agent for Program Repair". In: *arXiv preprint arXiv:2403.17134* (2024). URL: https://arxiv.org/abs/2403.17134.

[5] Moumita Das Purba et al. "Software Vulnerability Detection using Large Language Models". In: *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE. IEEE, 2023, pp. 112–119. DOI: 10.1109/ISSREW60843.2023.00058.

[6] Xin Yin, Chao Ni, and Shaohua Wang. "Multitask-based Evaluation of Open-Source LLM on Software Vulnerability". In: *IEEE Transactions on Software Engineering* (2024). DOI: 10.1109/TSE.2024.3470333.

[7] Zoltán Szabó and Vilmos Bilicki. "A New Approach to Web Application Security: Utilizing GPT Language Models for Source Code Inspection". In: *Future Internet* 15.10 (2023), p. 326. DOI: 10.3390/fi15100326. URL: https://www.mdpi.com/1999-5903/15/10/326.

[8] Niklas Risse, Jing Liu, and Marcel Böhme. "Top Score on the Wrong Exam: On Benchmarking in Machine Learning for Vulnerability Detection". In: *arXiv preprint arXiv:2408.12986* (2024). Accepted at the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2025). DOI: 10.48550/arXiv.2408.12986. URL: https://arxiv.org/abs/2408.12986.

[9] Sihao Hu et al. "Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives". In: *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE. 2023, pp. 297–306. DOI: 10.1109/TPS-ISA58951.2023.00044.

[10] Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. “CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software”. In: *arXiv preprint arXiv:2107.08760* (2021). DOI: 10.48550/arXiv.2107.08760. arXiv: 2107.08760 [cs.SE].

[11] Karl Tamberg and Hayretdin Bahsi. “Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study”. In: *arXiv preprint arXiv:2405.15614* (2024). URL: http://arxiv.org/abs/2405.15614.

[12] Lei Yu et al. “PSCVFinder: A Prompt-Tuning Based Framework for Smart Contract Vulnerability Detection”. In: *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2023, pp. 555–566. DOI: 10.1109/ISSRE59848.2023.00030.

[13] Zhihong Liu et al. “Software Vulnerability Detection with GPT and In-Context Learning”. In: *2023 8th International Conference on Data Science in Cyberspace (DSC)*. IEEE. Shenzhen, China: IEEE, 2023, pp. 229–236. DOI: 10.1109/DSC59305.2023.00041.

[14] Yuqiang Sun et al. “GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis”. In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE ’24)*. ACM, 2024. DOI: 10.1145/3597503.3639117.

[15] Guilong Lu et al. “GRACE: Empowering LLM-based Software Vulnerability Detection with Graph Structure and In-Context Learning”. In: *Journal of Systems and Software* 212 (2024), p. 112031. DOI: 10.1016/j.jss.2024.112031.

[16] Xin-Cheng Wen et al. “SCALE: Constructing Structured Natural Language Comment Trees for Software Vulnerability Detection”. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA ’24)*. ACM, 2024, p. 13. DOI: 10.1145/3650212.3652124.

[17] Zhihong Liu, Zezhou Yang, and Qing Liao. “Exploration on Prompting LLM with Code-Specific Information for Vulnerability Detection”. In: *IEEE International Conference on Software Services Engineering (SSE)*. IEEE. 2024.

[18] Chenyuan Zhang et al. “Prompt-Enhanced Software Vulnerability Detection Using ChatGPT”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion ’24)*. ACM. Lisbon, Portugal: ACM, 2024, pp. 276–277. DOI: 10.1145/3639478.3643065. URL: https://doi.org/10.1145/3639478.3643065.

[19] Xiaohu Du et al. "Generalization-Enhanced Code Vulnerability Detection via Multi-Task Instruction Fine-Tuning". In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Available at https://github.com/CGCL-codes/VulLLM. Association for Computational Linguistics. 2024.

[20] Jason Wei et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *arXiv preprint arXiv:2201.11903* (2022). URL: https://arxiv.org/abs/2201.11903.

[21] Zayne Sprague et al. "To COT or Not to COT? Chain-of-Thought Helps Mainly on Math and Symbolic Reasoning". In: *arXiv preprint arXiv:2310.09783* (2023). The University of Texas at Austin, Johns Hopkins University, Princeton University. URL: https://arxiv.org/abs/2409.12183.

[22] Hugo Touvron et al. "LLaMA: Open and Efficient Foundation Language Models". In: *arXiv preprint arXiv:2302.13971* (2023). URL: https://arxiv.org/abs/2302.13971.

[23] Jason Wei et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *arXiv preprint arXiv:2201.11903* (2022). DOI: 10.48550/arXiv.2201.11903. URL: https://arxiv.org/abs/2201.11903.

[24] Edward Hu et al. "LoRA: Low-Rank Adaptation of Large Language Models". In: *arXiv preprint arXiv:2106.09685* (2021). Version 2. URL: https://arxiv.org/abs/2106.09685.

[25] Jason Wei et al. "Finetuned Language Models Are Zero-Shot Learners". In: *arXiv preprint arXiv:2206.07682* (2022).

[26] Marah Abdin et al. *Phi-4 Technical Report*. 2024. arXiv: 2412.08905 [cs.CL]. URL: https://arxiv.org/abs/2412.08905.

[27] Zikai Xie. *Order Matters in Hallucination: Reasoning Order as Benchmark and Reflexive Prompting for Large-Language-Models*. 2024. arXiv: 2408.05093 [cs.CL]. URL: https://arxiv.org/abs/2408.05093.

[28] Unsloth Team. *Tutorial: How to Finetune LLaMA-3 and Use in Ollama*. Blog post. Accessed: Sunday 18th May, 2025. 2024. URL: https://docs.unsloth.ai/basics/tutorial-how-to-finetune-llama-3-and-use-in-ollama#id-6.-alpaca-dataset.

[29] Qiheng Mao et al. "Towards Effectively Detecting and Explaining Vulnerabilities Using Large Language Models". In: *arXiv preprint arXiv:2406.09701* (2024). Version 3. URL: https://arxiv.org/abs/2406.09701.

[30] Syafiq Al Atiiq, Christian Gehrmann, and Kevin Dahlén. "Vulnerability Detection in Popular Programming Languages with Language Models". In: *arXiv preprint arXiv:2412.15905* (2024). URL: https://arxiv.org/abs/2412.15905v2.

[31] Tiago Espinha Gasiba Kohei Dozono and Andrea Stocco. *Large Language Models for Secure Code Assessment: A Multi-Language Empirical Study*. 2024. DOI: 10.48550/arXiv.2408.06428. arXiv: 2408.06428 [cs.SE]. URL: https://arxiv.org/abs/2408.06428.

[32] Avishree Khare et al. *Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities*. 2023. DOI: 10.48550/arXiv.2311.16169. arXiv: 2311.16169 [cs.CR]. URL: https://arxiv.org/abs/2311.16169.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Keisuke Konno

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Evaluating and Fine-tuning Large Language Model for Code Vulnerability Detection Across Programming Languages and Code Granularities", supervised by Hayretdin Bahsi and Danilo Gligoroski
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

18.05.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - Source Code Repository

The source code and experiments related to this thesis are available on GitHub:

`https://github.com/kei5uke/llm-code-vuln`

This repository contains all scripts used in the experiments presented in this thesis.

Appendix 3 – Sample Codes and Output

Figure 41. CWE-89 (SQL Injection) Sample and LLM Output

Code:

```
function getTextColumns($table) {
                $sql = "SHOW COLUMNS FROM " . $this->prefix.$table . " WHERE type =
    'text' OR type like 'varchar%'";
                $res = @mysqli_query($this->connection, $sql);
                if ($res == null)
            return array();
                $records = array();
                while($row = mysqli_fetch_object($res)) {
                        $records[] = $row->Field;
                }

                return $records;
        }
```

LLM Output:

```
The code is vulnerable to SQL injection (CWE-89) because it directly concatenates
    user-controlled input, specifically the `$table` variable, into the SQL query
    without any validation or sanitization. This allows an attacker to manipulate the
    input to execute arbitrary SQL commands, potentially compromising the database's
    integrity and security. The lack of prepared statements or parameterized queries
    further exacerbates this vulnerability, as it fails to neutralize special
    characters that could alter the intended SQL command.
```

Figure 42. CWE-79 (XSS) Sample and LLM Output

Code:

```
function loadSessionsTable(sessions) {
  $.each(sessions, function(index, session) {
    $("#interactive-sessions .sessions-table-body").append(
      "<tr>" +
        tdWrap(uiLink("session/" + session.id, session.id)) +
        tdWrap(appIdLink(session)) +
        tdWrap(session.name) +
        tdWrap(session.owner) +
        tdWrap(session.proxyUser) +
        tdWrap(session.kind) +
        tdWrap(session.state) +
        tdWrap(logLinks(session, "session")) +
        "</tr>"
    );
  });
}

function loadBatchesTable(sessions) {
  $.each(sessions, function(index, session) {
    $("#batches .sessions-table-body").append(
      "<tr>" +
        tdWrap(session.id) +
```

```
        tdWrap(appIdLink(session)) +
        tdWrap(session.name) +
        tdWrap(session.owner) +
        tdWrap(session.proxyUser) +
        tdWrap(session.state) +
        tdWrap(logLinks(session, "batch")) +
        "</tr>"
    );
  });
}

var numSessions = 0;
var numBatches = 0;

$(document).ready(function () {
  var sessionsReq = $.getJSON(location.origin + prependBasePath("/sessions"),
    function(response) {
    if (response && response.total > 0) {
      $("#interactive-sessions").load(prependBasePath("/static/html/sessions-table.html
    .sessions-template"), function() {
        loadSessionsTable(response.sessions);
        $("#interactive-sessions-table").DataTable();
        $('#interactive-sessions [data-toggle="tooltip"]').tooltip();
      });
    }
    numSessions = response.total;
  });

  var batchesReq = $.getJSON(location.origin + prependBasePath("/batches"),
    function(response) {
    if (response && response.total > 0) {
      $("#batches").load(prependBasePath("/static/html/batches-table.html
    .sessions-template"), function() {
        loadBatchesTable(response.sessions);
        $("#batches-table").DataTable();
        $('#batches [data-toggle="tooltip"]').tooltip();
      });
    }
    numBatches = response.total;
  });

  $.when(sessionsReq, batchesReq).done(function () {
    if (numSessions + numBatches == 0) {
      $("#all-sessions").append('<h4>No Sessions or Batches have been created
    yet.</h4>');
    }
  });
});
```

LLM Output:

```
The code provided is a JavaScript snippet that dynamically loads session and batch data
    into HTML tables on a web page. It uses jQuery to make AJAX requests to fetch this
    data from server endpoints (/sessions and /batches). The fetched data is then used
    to populate table rows within the DOM.

1. **Overall Structure**: The code consists of two main functions, loadSessionsTable
    and loadBatchesTable, which are responsible for appending session and batch
    information into respective HTML tables. It also includes a jQuery document ready
    function that initiates AJAX requests to fetch data when the page loads.
```

```
2. **User Input Handling & Data Flow**: The code does not directly handle user input;
    instead, it processes server responses containing session and batch data. The
    critical part of this analysis is how the data from these responses is used in
    constructing HTML elements. Specifically, the functions uiLink, appIdLink, and
    logLinks are called with parameters derived from the response data.

3. **Vulnerability Analysis**:
   - The primary concern here is the potential for Cross-Site Scripting (XSS)
    vulnerabilities. This arises because the code constructs HTML elements using string
    concatenation, incorporating data directly from server responses without any
    explicit sanitization or escaping.
   - Functions like uiLink, appIdLink, and logLinks are used to generate parts of the
    HTML content. If these functions do not properly escape or sanitize their inputs,
    they could inadvertently allow malicious scripts to be injected into the page if an
    attacker can control the data being returned by the server.
   - The use of jQuery's .append() method with unsanitized input is a common vector for
    XSS attacks, as it directly inserts HTML content into the DOM.

4. **Vulnerable Code Identification**:
The code could be vulnerable to XSS if any of the functions (uiLink, appIdLink,
    logLinks) do not properly escape or sanitize their inputs before incorporating them
    into HTML strings.

5. **Explanation of Vulnerability**:
   - Cross-Site Scripting (XSS) occurs when an application includes untrusted data in a
    web page without proper validation or escaping, allowing attackers to execute
    scripts in the context of another user's browser session.
   - In this code, if any of the functions used to generate HTML content do not escape
    special characters like <, >, &, etc., it could lead to XSS vulnerabilities. This
    is because an attacker might be able to inject malicious JavaScript into these
    fields through server-side manipulation or other means.

6. **Conclusion**:
The code is potentially vulnerable to Cross-Site Scripting (XSS) due to the direct
    inclusion of untrusted data in HTML content without explicit sanitization or
    escaping. This aligns with CWE-79, which describes XSS vulnerabilities.
```

Figure 43. CWE-352 (CSRF) Sample and LLM Output

Code:

```
from flask import Flask, render_template, request, session, redirect
import os, sys
import requests
import jwt

CLIENT_ID = 'valtech.idp.testclient.local'
CLIENT_SECRET = os.environ.get('CLIENT_SECRET')

if CLIENT_SECRET is None:
  print 'CLIENT_SECRET missing. Start using "CLIENT_SECRET=very_secret_secret python
    main.py"'
  sys.exit(-1)

app = Flask(__name__, static_url_path='')

@app.route('/')
def index():
  signed_in = session.get('signed_in') != None
```

```
  header = 'Not signed in'
  text = 'Click the button below to sign in.'

  if signed_in:
    header = 'Welcome!'
    text = 'Signed in as %s.' % session['email']

  return render_template('index.html', header=header, text=text)

@app.route('/sign-in')
def sign_in():
  if session.get('signed_in') != None: return redirect('/')
  authorize_url =
    'https://stage-id.valtech.com/oauth2/authorize?response_type=%s&client_id=%s&scope=%s'
    % ('code', CLIENT_ID, 'email openid')
  return redirect(authorize_url)

@app.route('/sign-in/callback')
def sign_in_callback():
  code = request.args.get('code')

  # as both scope openid and email was requested on authorize request above, the client
  # will receive both an access_token (according to OAuth 2) AND an id_token (according
    to OpenID Connect)
  tokens = exchange_code_for_tokens(code)

  # if the client only need authentication (and not authorization), the access token
    can be ignored
  # (but it is still possible to use it if client wants to, and is left here for
    documentation)
  #user_info = fetch_user_info(tokens['access_token'])

  # as this example app is only interested in who logged in, we will parse the id_token.
  # currently, IDP does not sign id_tokens, but as IDP uses https this is no problem
  # (but the id_token should not be passed around in plaintext where it can be modified
    by a man-in-the-middle)
  user_info = jwt.decode(tokens["id_token"], verify=False)

  session['signed_in'] = True
  session['email'] = user_info['email']

  return redirect('/')

@app.route('/sign-out')
def sign_out():
  session.clear()
  return redirect('https://stage-id.valtech.com/oidc/end-session?client_id=%s' %
    CLIENT_ID)

def exchange_code_for_tokens(code):
  data = {
    'grant_type': 'authorization_code',
    'code': code,
    'client_id': CLIENT_ID,
    'client_secret': CLIENT_SECRET
  }

  res = requests.post('https://stage-id.valtech.com/oauth2/token', data=data)
  return res.json()
```

```
def fetch_user_info(access_token):
  res = requests.get('https://stage-id.valtech.com/api/users/me', headers={
    'Authorization': 'Bearer %s' % access_token })
  return res.json()

if __name__ == '__main__':
  app.secret_key = 'someverysecretkey'
  app.run(host='0.0.0.0', debug=True)
```

LLM Output:

```
The code is a Flask web application that handles user authentication using OAuth2 and
    OpenID Connect protocols. It includes routes for signing in, handling callbacks,
    and signing out. The main areas of concern are input handling and session
    management.

1. **Input Handling**:
   - The `code` parameter from the URL query string is directly used without validation
    or sanitization when calling `exchange_code_for_tokens`. This could potentially
    lead to issues if the code is manipulated, although OAuth2 typically handles this
    securely.
   - JWT tokens are decoded with `verify=False`, which means they are not
    cryptographically verified. This can be dangerous as it allows tampering with the
    token data without detection.

2. **Session Management**:
   - The session secret key (`app.secret_key`) is hardcoded as 'someverysecretkey'. This
    is a security risk because if this value is exposed, an attacker could potentially
    manipulate session cookies.

3. **Security Vulnerabilities Identified**:
   - **JWT Verification**: The JWT token is decoded without verification, which can
    lead to tampering with the token data (CWE-352: Cross-Site Request Forgery).
   - **Hardcoded Secret Key**: Using a hardcoded secret key for session management
    poses a risk of session hijacking.

Given these issues, the code contains vulnerabilities related to JWT handling and
    session security.
```

Figure 44. CWE-434 (Unrestricted Upload of File) Sample and LLM Output

Code:

```
public function __construct($page, $file = null) {

    $this->page      = $page;
    $this->file      = $file;
    $this->blueprint = $page->blueprint();
    $this->filename  = $this->blueprint->files()->sanitize() ? '{safeFilename}' :
    '{filename}';

    if($this->file) {
      $this->replace();
    } else {
      $this->upload();
    }

  }
```

LLM Output:

```
The provided code is vulnerable to CWE-434 (Unrestricted Upload of File with Dangerous
    Type) due to its handling of file uploads without adequate validation or
    restrictions on the types of files that can be uploaded. Specifically, the function
    `check_file_writable_user` checks if a directory is writable but does not enforce
    any checks on the file type being uploaded. This lack of validation allows an
    attacker to upload potentially harmful files (e.g., executable scripts) to the
    server, which could then be executed in the context of the web application or
    system, leading to unauthorized access or control over the server. The absence of
    strict controls on file types directly correlates with the characteristics of
    CWE-434, where the vulnerability arises from allowing uploads without proper
    restrictions.
```