JAIST Repository
https://dspace.jaist.ac.jp/

Title	Python Code Verification and Improvement Using Large Language Models
Author(s)	李, 宗珉
Citation	
Issue Date	2025-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/20038
Rights	
Description	Supervisor: BEURAN, Razvan Florin, 先端科学技術研究科, 修士 (情報科学)

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Master's Thesis

Python Code Verification and Improvement Using Large Language Models

2310402 Jongmin Lee

Supervisor	Razvan Florin Beuran
Main Examiner	Razvan Florin Beuran
Examiners	Yasuo Tan
	Yuto Lim
	Naoya Inoue

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)
August 2025

Acknowledgment

Firstly, I would like to express my deepest gratitude to my supervisor, Associate Professor Razvan Florin Beuran. When I first started, I knew nothing about cybersecurity, but he patiently taught me everything. His careful and meticulous guidance helped me avoid many mistakes. Without his support and guidance, none of my achievements would have been possible.

I want to thank my colleagues. Their helpful discussions and support during important times were key in dealing with the more complicated parts of this research. In particular, I am especially grateful to my senior, Khang Mai, from whom I received much advice and support in my research. His help was essential for my progress. I wish him great success in his research after returning to Vietnam. I am also deeply thankful to Zhi Chen, who helped me a lot in daily life. I hope he is enjoying his research and doctoral studies at Kanazawa University. I also wish to thank Surawat Pothong, a member of RebelsNLU. Sharing wine and talking about research and life with him in Kanazawa was a truly sweet break for me. I am also grateful to my fellow Korean students, Samyoung Kim, Junwoo Choi and Jinwoo Kim. During my long journey abroad, they helped me a lot as compatriots, and playing computer games with them was a great way for me to relieve stress, and I would also like to express my sincere gratitude to Professor Donghan Kim from Daegu Gyeongbuk Institute of Science & Technology (DGIST), who is also a JAIST senior and returned this year for research. He generously treated me and other Korean students to meals and drinks while giving us valuable advice on research and life.

I would also like to thank my close friends from my university days, the so-called "The Trio". Jaehyeon Lee, who is happily doing research about solid-state physics at Ulsan National Institute of Science and Technology (UNIST), had many deep conversations with me about research. Seonghoon Oh's cheerful personality always brought me great joy. I hope he becomes a good engineer in the future.

I am also deeply grateful to the members of "Silver Town", a group that began during my undergraduate years. Surrounded by talented developers and outstanding people, this group gave me both emotional and technical support. Especially, I would like to give special thanks to Unmun Lee, an excellent developer at Diquest. When I first started my journey into information science, his insights into artificial intelligence and natural language processing were extremely helpful to me.

Lastly, I would like to dedicate this thesis to my beloved, Len. I hope she can also begin her own joyful journey in research. I may not say it often, but I am truly grateful to my younger brother, Jonghyeon, and I wish him all the best in his life as an officer. I want to send all my love to my parents, who always support me with their love.

Abstract

The explosive growth of artificial intelligence (AI) has propelled large language models (LLMs) from academic curiosities to everyday development companions. By translating natural language descriptions directly into source code, LLMs promise unprecedented boosts in programmer productivity and faster delivery cycles. Yet, this promise is overshadowed by persistent concerns. Empirical studies show that automatically generated code frequently embeds critical security weaknesses, subtle logic flaws, and mismatches between user intent and program behavior. These shortcomings can translate into exploitable vulnerabilities, unexpected production outages, and costly maintenance overhead. Traditional manual reviews cannot keep pace with the volume and velocity of LLM output, underscoring the urgency for scalable, automated safeguards.

This thesis introduces CodeEnhancer, a fully automated, two-stage framework that tackles these challenges holistically. Stage 1 combines best-practice structured prompting with an iterative "generate-verify-refine" loop. Each candidate program is scrutinized by Pylint for syntax compliance, Bandit for CWE-mapped vulnerability patterns, and an LLM-based reasoning module that cross-checks functional intent captured in a mandatory docstring. Structured, tool-specific feedback is fed back to the LLM until the code passes all checks or a configurable iteration budget is exhausted, yielding a corpus of high-quality, self-documented scripts with minimal human oversight.

Stage 2 shifts the focus from fixing problems after the fact to proactively ensuring code quality. In this phase, GPT-4o is fine-tuned using two complementary datasets. Secure code examples authored by experts from the LLMSecEval benchmark, and refined outputs from Stage 1. This combined training approach equips the model with both best-practice secure coding techniques and practical remediation strategies, allowing it to recognize and prevent common vulnerabilities during code generation.

Comprehensive experiments on the security-oriented LLMSecEval and SecurityEval benchmarks confirm CodeEnhancer's effectiveness. A GPT-4o baseline produced vulnerable code in 43.6% of LLMSecEval tasks. A GPT-4o variant further trained with the CodeEnhancer pipeline (framework-tuned GPT-4o) lowered this rate to 18.4% prior to refinement and to only 8% after one refinement cycle. On SecurityEval, the same framework-tuned GPT-4o cut the final vulnerability rate. Functional correctness exhibited similar improvements, with 98.7% prompt adherence after refinement while preserving

or enhancing execution efficiency. Importantly, the framework-tuned GPT-4o achieved these gains in fewer iterations than both the baseline and a fine-tuned model by secure code written by experts. Underscoring the synergistic value of CodeEnhancer's automatically generated training data.

Beyond empirical metrics, CodeEnhancer offers practical benefits. It integrates seamlessly into CI/CD pipelines, requires no language-specific test harnesses, and remains agnostic to future advances in static code analysis (SAST) tools. Nevertheless, limitations persist. Current evaluations target Python and static analysis. Extending coverage to languages with stronger type systems and incorporating dynamic or formal constitutes next steps. Additionally, broader datasets that capture domain-specific security requirements will help generalize the approach.

In summary, CodeEnhancer demonstrates that pairing LLMs with iterative static analysis, verification, and feedback-informed fine-tuning creates a virtuous cycle that progressively decreases vulnerabilities and increases trustworthiness. The framework lays a scalable foundation for safely harnessing AI-powered code generation across modern software engineering workflows, paving the way for future research on multi-language support, dynamic analysis, and continuous self-improvement loops.

Keywords: Large Language Models (LLMs), Code Generation, Software Security, Static Analysis, Functional Correctness, Fine-Tuning

Contents

Acknowledgment **1**

Abstract **3**

Contents **5**

List of Figures **7**

List of Tables **9**

Chapter 1 Introduction **11**
1.1 Motivation 11
1.2 Problem Definition 12
1.3 Research Approach 13
1.4 Contributions 13
1.5 Thesis Structure 14

Chapter 2 Background and Related Work **15**
2.1 Large Language Models for Code Generation 15
2.2 Static Application Security Testing 16
2.3 Large Language Models for Code Verification 17
2.4 Fine-Tuning Large Language Models 18
2.5 Functional Correctness Evaluation 20
2.6 Comparative Analysis 20

Chapter 3 CodeEnhancer Framework **23**
3.1 Overall Architecture 23
3.2 Stage 1: Code Generation, Verification & Refinement 24
3.2.1 Code Generation with Structured Prompting 25
3.2.2 Syntax, Security & Functional Correctness Verification 25
3.2.3 Iterative Refinement 27
3.3 Stage 2: LLM Fine-Tuning for Code Enhancement 28
3.3.1 Dataset Preparation and Model Training 29

3.3.2 Fine-Tuning Strategy 29

Chapter 4 Experimental Results **31**
4.1 Preliminary Experiment 31
4.1.1 Datasets 31
4.1.2 Methodology 32
4.1.3 Results and Analysis 32
4.2 Experiment Setup 34
4.2.1 Datasets 34
4.2.2 LLM Configurations 35
4.2.3 Evaluation Metrics 36
4.3 Experiment #1: Code Generation, Verification & Refinement . 36
4.3.1 Syntax Error Verification 37
4.3.2 Vulnerability Verification 37
4.3.3 Functional Correctness Verification 38
4.3.4 Quantitative Results Using Code-Level Metrics 39
4.4 Experiment #2: LLM Fine-Tuning for Code Enhancement . . 42
4.4.1 Syntax Error Verification 43
4.4.2 Vulnerability Verification 43
4.4.3 Functional Correctness Verification 50

Chapter 5 Discussion **53**
5.1 Vulnerability Mitigation 53
5.2 Functional Correctness Evaluation 53
5.3 Static Analysis Capabilities 54
5.4 Dataset Representativeness 54

Chapter 6 Conclusion **55**
6.1 Summary 55
6.2 Future Work 55

Publications **57**

List of Figures

2.1 Workflow of fine-tuning a large language model. 19

3.1 Overall architecture of CodeEnhancer, illustrating the relationship between the code generation & verification refinement pipeline (Stage 1) and the fine-tuning methodology (Stage 2). 23
3.2 Example of a code syntax issue detected via Pylint. 26
3.3 Example of a code vulnerability issue detected via Bandit. . . 27
3.4 Example of a functional correctness issue detected via LLM . . 28

4.1 Example of a natural language prompt from the LLMSecEval dataset. 32
4.2 Percentage of code snippets with vulnerability issues for the baseline and enhanced groups (LLMSecEval, 150 samples). . 33
4.3 Distribution of vulnerabilities per CWE for the original and enhanced prompt groups (LLMSecEval, 150 samples). 33
4.4 Example of a natural language prompt from the SecurityEval dataset. 35
4.5 Percentage of code snippets with vulnerability issues before and after iterative refinement (LLMSecEval, 150 samples). . . 37
4.6 Percentage of code snippets with functional correctness issues before and after iterative refinement (LLMSecEval, 150 samples). 39
4.7 Experiment #2: Comparison of the percentage of code snippets with vulnerability issues before and after iterative refinement for each tested model when using the LLMSecEval (a) and SecurityEval (b) datasets; error bars indicate the standard deviation across five experiment runs. 44
4.8 Experiment #2: Cumulative percentage of secure code snippets generated after each refinement iteration when using the LLMSecEval (a) and SecurityEval (b) datasets; error bars indicate the standard deviation across five experiment runs, and labels show the average percentage for each iteration. . . . 46

4.9 Experiment #2: Distribution of vulnerabilities per CWE for each model before (a) and after (b) refinement, averaged over 5 runs LLMSecEval dataset; error bars indicate standard deviation across five experiment runs. 48
4.10 Experiment #2: Distribution of vulnerabilities per CWE each model before (a) and after (b) refinement, averaged over 5 runs (SecurityEval, 121 samples); error bars indicate standard deviation across five experiment runs. 49
4.11 Experiment #2: Comparison of the percentage of code snippets with functional correctness issues before and after iterative refinement for each tested model when using the LLMSecEval (a) and SecurityEval (b) datasets; error bars indicate the standard deviation across five experiment runs. . 51

List of Tables

2.1 Comparison of related research works. 21

4.1 Experiment #1: Average CodeBLEU score comparison of secure code examples versus initial/final code snippets (LLMSecEval, 150 samples). 40
4.2 Experiment #1: Average CodeBERTScore comparison of secure code examples versus initial/final code snippets, and initial versus final code snippets (LLMSecEval, 150 samples). . 41
4.3 Experiment #1: Average ICE-SCORE comparison of initial versus final code snippets (LLMSecEval, 150 samples). 42

Chapter 1

Introduction

1.1 Motivation

In recent years, the field of artificial intelligence (AI) has experienced unprecedented growth, leading to the rapid development and adoption of large language models (LLMs). These LLMs, such as GPT-4 [1], Code Llama [2], and Codex [3], are revolutionizing software development. As a result, software development processes have become more efficient. LLMs generate code from natural language descriptions, accelerating development cycles and automating complex tasks [1, 3]. This offers substantial potential for boosting developer productivity and efficiency.

However, alongside these promising advancements, significant concerns have emerged regarding the reliability, security, and correctness of code generated by LLMs. Recent empirical studies have demonstrated that LLM-generated code frequently exhibits a range of issues, including syntax errors, logical inconsistencies, and, most critically, security vulnerabilities [4, 5]. If these weaknesses are not properly identified and addressed, deploying such code in production systems could result in critical security breaches and substantial operational risks.

Ensuring the functional correctness of LLM-generated code also presents substantial challenges. While LLMs are capable of producing code that appears syntactically correct or that passes limited unit tests [3], there often remains a gap between the user's intent as described in the prompt and the actual behavior of the generated code. Subtle misinterpretations or omissions can lead to code that fails to fully implement required features or introduces unexpected behavior—issues that are especially problematic in domains with strict security or reliability requirements [6]. Manual code review and testing can help, but these methods are labor-intensive and not scalable for the volume of code that LLMs are now capable of generating.

To address these concerns, researchers and practitioners have explored a range of validation and remediation strategies. Traditional approaches include static application security testing (SAST) tools, such as Pylint [7]

and Bandit [8], which are effective at detecting syntax errors and security vulnerabilities. However, these tools alone are limited in their ability to guarantee comprehensive security or functional correctness, especially for complex or context-dependent [9, 10]. Self-correction by LLMs, where the model tries to improve its own responses based on feedback, has so far achieved only limited success. This is because models often have difficulty spotting and fixing subtle mistakes without clear, structured guidance [11].

Motivated by these challenges, this thesis proposes a comprehensive framework that tightly integrates LLM-based code generation with iterative verification using SAST tools and targeted fine-tuning strategies. By systematically combining these approaches, the goal is to advance the security, correctness, and overall quality of LLM-generated Python code. This research aims to contribute not only to the improvement of AI-assisted software engineering practices but also to the development of scalable solutions for deploying trustworthy code in real-world applications.

1.2 Problem Definition

LLMs have shown impressive results in generating source code from natural language descriptions. However, several practical, critical problems remain.

First, LLM-generated code often includes security vulnerabilities, such as command injection and insecure data handling [4, 5]. These weaknesses are difficult to detect and fix automatically. Second, functional correctness is not guaranteed. Code may look correct or pass basic tests [3], but still fail to match the user's real intent [6]. This gap between user requirements and actual code behavior is especially serious in domains where mistakes can have costly consequences. Third, current verification methods have clear limits. Manual code review is slow and cannot scale. Self-correction by LLMs is unreliable without structured feedback [11]. Fourth, most current research only focuses on a single aspect, such as security [4] or functional correctness [3]. There is a lack of frameworks that address all these problems together in an integrated way.

Because of these problems, it remains challenging to safely and reliably use LLM-generated code in real-world software projects. There is a need for new frameworks that can systematically detect, validate, and remediate both security and correctness issues.

This thesis focuses on addressing these research challenges. The goal is to develop an integrated approach that combines LLM-based code generation with iterative validation and targeted fine-tuning, to produce Python code that is both secure and functionally correct.

1.3 Research Approach

To address the problems described in the previous section, this thesis proposes a two-stage framework. The first stage uses an iterative validation pipeline. This pipeline combines LLM-based code generation with multiple validation steps. The system first uses static application security testing (SAST) tools, such as Pylint and Bandit, to detect syntax errors and security vulnerabilities. In addition, the framework uses the LLM itself to validate functional correctness by comparing the intended behavior in the prompt with the actual code logic. The feedback from all these checks is returned to the LLMs, which try to fix the issues. This process repeats until the code passes all validations or a set number of rounds is reached.

In the second stage, the research applies targeted fine-tuning to the LLMs. The fine-tuning uses secure code examples: code that has been refined by the framework itself. By training the LLMs on these datasets, the model learns to generate more secure and correct code from the start. The aim is to reduce both vulnerabilities and functional errors in newly generated code.

The effectiveness of this approach is evaluated through experiments on benchmark datasets, LLMSecEval and SecurityEval. Performance is measured in terms of vulnerability rates, functional correctness rates, and other relevant metrics. By systematically combining validation and fine-tuning, this research aims to create a practical and scalable solution for improving the security and reliability of LLM-generated Python code.

1.4 Contributions

The main contributions of this thesis include:

- Presenting the design and implementation of a comprehensive two-stage framework, CodeEnhancer that integrates an iterative LLM-SAST validation pipeline with a subsequent targeted LLM fine-tuning methodology for enhancing Python code security.
- A rigorous evaluation demonstrating the framework's effectiveness, using the LLMSecEval benchmark [12] for validating the first stage, and the LLMSecEval and SecurityEval benchmarks [13] for assessing the performance of the fine-tuned models in the second stage. Notably, this approach not only outperformed the expert-written examples in terms of model performance but also proved to be more cost-effective.
- An analysis providing insights into the synergistic benefits of combining

automated validation with model adaptation, contributing to the development of more trustworthy and secure AI-assisted coding practices.

1.5 Thesis Structure

The remainder of this thesis is organized as follows.

- Chapter 2 reviews the research background and related work. It covers large language models for code generation and validation, fine-tuning methods, and static application security testing.
- Chapter 3 describes the proposed two-stage framework. It explains the overall architecture, the code generation and validation pipeline, and the fine-tuning process.
- Chapter 4 presents the experimental setup and results. It includes information about datasets, model configurations, evaluation metrics, and a detailed analysis of the findings.
- Chapter 5 discusses the limitations of the current approach and possible directions for future work.
- Chapter 6 concludes the thesis by summarizing the main contributions and results.
- Appendices and references are included at the end of the thesis.

Chapter 2

Background and Related Work

2.1 Large Language Models for Code Generation

Background LLMs are advanced artificial intelligence models trained on vast collections of natural language and programming code. These models, such as AlphaCode [14] and Codex [3], have demonstrated remarkable abilities to understand prompts written in natural language and to generate syntactically correct source code in various programming languages, including Python. By predicting the next most likely token given the input context, LLMs can create code snippets, complete functions, or even write entire programs based on short descriptions provided by users.

The impact of LLMs on software development has been significant. Tools like GitHub Copilot [15] and ChatGPT [16] are now integrated into daily programming workflows. Developers can use LLMs to automate repetitive coding tasks, speed up prototyping, and get real-time code suggestions or bug fixes [15, 16]. As a result, LLMs are helping to increase productivity and lower the barrier to entry for programming.

Related Work Despite these advantages, several challenges remain. The code generated by LLMs does not always fully align with the user's intent [3]. Sometimes it only partially implements the required functionality or fails to handle special cases. More importantly, LLMs can generate code with subtle logic errors, inefficient implementations, or even serious security vulnerabilities [6]. These vulnerabilities can have severe consequences if deployed without conducting a thorough review.

The quality and security of LLM-generated code are not guaranteed by the model itself. Because LLMs are trained on large datasets collected from public code repositories and forums [1, 4], they may repeat insecure or outdated coding patterns found in the training data. Additionally, current LLMs do not always generalize well to complex or domain-specific programming tasks, which can lead to unpredictable results.

Previous research has also tested whether simply asking LLMs to generate secure code in the prompt can prevent unsafe outputs [17]. However, these efforts have shown clear limitations. Even when prompts explicitly request secure or best-practice implementations, LLMs often continue to produce code with vulnerabilities or overlook critical security steps. This suggests that prompt engineering alone is not enough to ensure the security of automatically generated code.

Therefore, these issues, there is a growing need for frameworks and tools that can verify, refine, and improve LLM-generated code. This includes automated checks for syntax and security, interactive feedback mechanisms, and methods for guiding the model toward secure and reliable code generation. Fine-tuning LLMs on high-quality, secure code examples and integrating external validation tools are promising approaches, but further research is required to ensure that these models can consistently produce trustworthy code in practical software engineering scenarios.

In summary, while large language models have opened new possibilities in automated code generation, ensuring the quality, security, and correctness of their outputs remains an open research challenge.

2.2 Static Application Security Testing

Background Static Application Security Testing (SAST) is a core technique for detecting issues in software at an early stage of development. SAST tools analyze the source code, bytecode, or binary code of an application without executing the program. This approach is often called "white-box testing" because it inspects the internal logic and structure of the code. SAST tools can automatically identify a wide range of security flaws, such as injection attacks, buffer overflows, improper error handling, and insecure cryptographic use. By finding vulnerabilities before the software is run, SAST helps developers fix issues early, reducing the cost and risk of security breaches after deployment.

A typical SAST process involves scanning the entire codebase for known patterns of insecure practices or code smells, which are characteristics in the source code that may indicate deeper problems [7, 18]. Integrating SAST into the software development lifecycle enables continuous security assessment, making it easier to maintain high security standards as code changes over time.

For Python, Bandit [8] targets common security flaws, while Pylint [7] checks for code smells, style issues, and syntax errors. More sophisticated tools like GitHub's CodeQL use semantic analysis to detect complex vul-

nerabilities across multiple languages [19]. Researchers utilize these tools extensively to evaluate the security posture of LLM-generated code [20]. However, studies caution against relying on a single tool, as their detection capabilities vary across vulnerability types [21], and even advanced tools like CodeQL can miss vulnerabilities. Enterprise tools like SonarQube [22] and Checkmarx [23] also exist, offering broader capabilities and integrations but come with different complexities and costs.

Related Work Hajipour et al. [24] introduced SimSCoOD, a benchmark for evaluating the security of code generated by large language models. Their findings indicate that while LLMs are capable of generating functionally correct and, in many cases, secure code, there are still significant limitations and open challenges in consistently ensuring code safety. The study quantitatively demonstrates that current LLMs may still produce vulnerable code under certain scenarios, highlighting the necessity for further research on automated security verification.

Despite recent advances in large language model code generation, limitations remain. Li et al. [25] conducted a comprehensive study using static analysis tools and found that LLMs, while capable of producing high-quality and secure code in various programming languages, still frequently generate code with security vulnerabilities. Similarly, Hajipour et al. [24] demonstrated through the SimSCoOD benchmark that, although LLMs can often generate functionally correct and relatively secure code, they do not consistently guarantee code safety. Both studies underscore that the current generation of LLMs cannot fully address security concerns and highlight the necessity for further improvements and robust automated security validation.

2.3 Large Language Models for Code Verification

Background LLMs show potential but face challenges with complex vulnerabilities. Previous research [26]. investigated the zero-shot vulnerability repair capabilities of Codex, finding it struggled with issues requiring contextual understanding. Prenner et al [27] evaluated Codex using the QuixBugs benchmark, noting that while some bugs were fixed, new errors were often introduced. Alrashedy et al. proposed a feedback-driven method employing static analysis tools like Bandit to enhance LLMs' ability to repair security issues, achieving significant improvements.

Related Work Despite these advances, LLM-based validation still has limits. LLMs may overlook edge cases, fail to detect complex logic bugs, or not fully understand deep program semantics [28]. As a result, LLM verification is often used alongside static analysis tools to create a more reliable and comprehensive verification process.

Alrashedy et al. [29] presented a framework that integrates LLMs with SAST tools to automatically patch security vulnerabilities in source code. Their system uses the output of SAST tools to generate prompts for LLM-based repair, representing one of the earliest pipelines combining static analysis and generation in an automated feedback loop.

Separately, Keltek et al. [30] proposed LSAST, a hybrid system where SAST tools such as Bearer are paired with LLMs to enhance vulnerability detection coverage. While not explicitly aimed at code generation or repair, LSAST exemplifies the complementary use of LLMs for security analysis and code validation.

Overall, LLMs provide a way to verify security and functional correctness, but their effectiveness depends on careful prompt engineering, structured validation, and integration with other SAST tools.

2.4 Fine-Tuning Large Language Models

Background Fine-tuning is a process that takes pre-trained LLMs and further trains them on a custom dataset for a specific task or domain [31]. Instead of training a model from scratch, fine-tuning allows developers to adapt an existing model's behavior by exposing it to examples that are closely related to the intended application. During fine-tuning, the model learns patterns, preferences, and constraints from the new data, making its outputs better suited to the user's needs. This approach is widely used to improve the accuracy and relevance of LLMs in specialized scenarios, such as code generation [3] or medical applications [32].

This additional training helps the model to learn domain-specific patterns, best practices, and security guidelines that may not be well represented in the original pre-training data. For example, LLMs can be fine-tuned on a curated collection of secure code examples or specialized programming tasks. As a result, the model improves its ability to generate code that is not only syntactically correct but also aligns with domain-specific standards and expectations.

Related Work More recent work has shifted toward improving the ability of LLMs to produce secure code through fine-tuning. Early LLMs for

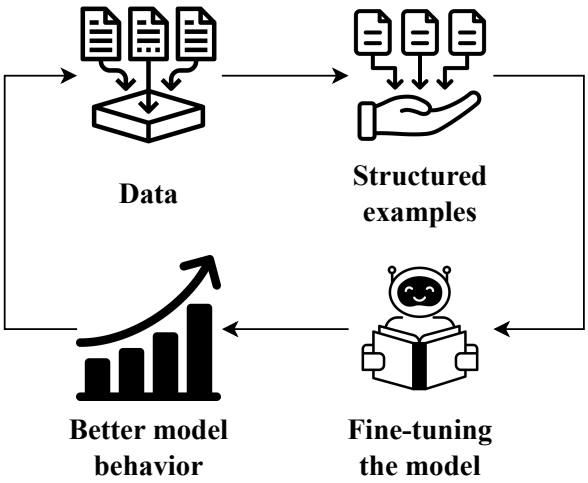


Figure 2.1: Workflow of fine-tuning a large language model.

code generation, such as Codex [3], demonstrated impressive capabilities in producing syntactically correct and useful code. Li et al. [33] conducted an empirical study on fine-tuning models using security patch commits, showing measurable improvements in vulnerability reduction. Similarly, Weyssow et al. [34] explored parameter-efficient fine-tuning (PEFT) methods for code generation, though their study focused more on syntax and structural quality rather than vulnerability mitigation.

Despite these benefits, fine-tuning is not without challenges. The effectiveness of this approach depends heavily on the quality and diversity of the fine-tuning dataset. There is also a risk of overfitting [24], where the model becomes too specialized and loses its generalization ability. Additionally, some security threats or domain requirements may be underrepresented, making it difficult to achieve comprehensive coverage [33].

In conclusion, fine-tuning is a powerful tool for adapting LLMs to new tasks and domains. By combining large-scale pre-training with targeted, domain-specific learning, this method helps create models that are better suited for practical software engineering applications, especially in areas where correctness and security are critical. However, practitioners should be aware that the success of fine-tuning depends heavily on the quality and diversity of the fine-tuning data. Overfitting, loss of generalization, and insufficient coverage of rare security threats remain important challenges. Careful dataset construction and regular evaluation are needed to ensure robust and trustworthy outcomes.

2.5 Functional Correctness Evaluation

Background Verifying that generated code behaves as intended is a challenge in the evaluation of code generation models. Traditionally, functional correctness has been assessed using code similarity metrics such as CodeBLEU [35] and CodeBERTScore [36], which directly compare the generated code to a reference implementation. Another widely adopted approach is execution-based testing, exemplified by benchmarks like HumanEval [3] and The Mostly Basic Programming Problems (MBPP) [37]. In this setting, the generated code is evaluated based on its ability to pass a set of predefined unit tests [38]. While these methods provide useful indicators of functional correctness, they may not fully capture the specification or intention expressed in the prompt, especially in cases where multiple correct implementations exist or the test cases do not exhaustively cover all requirements.

Related Work To address the limitations of traditional evaluation methods, recent research has proposed several advanced metrics that aim for better alignment with prompt specifications. For example, CodeScore [39] introduces a model that predicts execution outcomes rather than relying solely on reference code comparison. ICE-Score [40] leverages an LLM to assess the correctness of generated code directly against the input prompt, enabling more flexible and context-aware evaluation. Similarly, SBC-Score [41] utilizes a reverse-generation approach, where requirements are inferred from the generated code and then compared back to the original prompt using semantic similarity, lexical overlap, and completeness measures. Building on these developments, our framework incorporates a functional correctness verification step that uses LLM reasoning to evaluate code based on specifications derived from the input prompt.

2.6 Comparative Analysis

As shown in Table 2.1, previous studies have mostly addressed individual aspects of code generation and validation. For example, Codex [3] focused on functional code generation, but did not target security or error repair. Alrashedy et al. [29] and Pearce et al. [42] incorporated LLM-based repair and static analysis, but did not employ fine-tuning or address syntax issues in depth. Other works, such as Li et al. [33] and Weyssow et al. [34], explored fine-tuning and parameter-efficient techniques, but focused mainly on either security or functional correctness, not both.

Table 2.1: Comparison of related research works.

Feature / Capability	Chen et al. [3]	Alrashedy et al. [29]	Pearce et al. [42]	Li et al. [33]	Weyssow et al. [34]	Keltek et al. [30]	**CodeEnhancer**
Code Generation	✓	✓	-	✓	-	-	✓
Code Validation & Repair	-	✓	✓	-	✓	✓	✓
Uses LLM	✓	✓	✓	✓	✓	✓	✓
Uses SAST Tools	-	✓	✓	-	-	✓	✓
Employs Fine-tuning	✓	-	-	✓	✓	-	✓
Addresses Syntax Errors	-	-	-	-	-	-	✓
Addresses Security Vulnerabilities	-	✓	✓	✓	-	✓	✓
Addresses Functional Correctness	✓	-	-	-	✓	-	✓

✓: Supported/Primary Focus; (∼): Partially Supported/Secondary Focus/Implicit; -: Not Supported/Not Within Scope

In contrast, CodeEnhancer combines all these elements into a single, unified framework. It supports Python code generation, iterative static and dynamic validation, automated repair, and domain-adaptive fine-tuning. CodeEnhancer is the only approach in the comparison that explicitly addresses syntax errors, security vulnerabilities, and functional correctness together. It also works with real-world, natural language prompts rather than synthetic or artificial examples.

However, CodeEnhancer is not only this integration, but also the use of an automated self-refinement loop. Outputs from the verification and refinement steps are systematically fed back into the fine-tuning data. This enables the model to learn directly from its own corrections across multiple iterations, resulting in improvements in security and functional correctness. In our experiments, this feedback-driven approach led to models fine-tuned by CodeEnhancer's code that outperformed both models fine-tuned by secure code examples written by experts and the baseline, demonstrating that high-quality, self-generated data can be more beneficial for fine-tuning than relying solely on expert-written code.

The next chapter introduces the methodology behind CodeEnhancer.

Chapter 3

CodeEnhancer Framework

3.1 Overall Architecture

The CodeEnhancer framework is designed to systematically enhance syntax, security, and the overall quality of Python code generated by LLMs. The framework adopts a two-stage architecture that combines the strengths of automated static analysis, iterative feedback, and large-scale model fine-tuning, as shown in Figure 3.1.

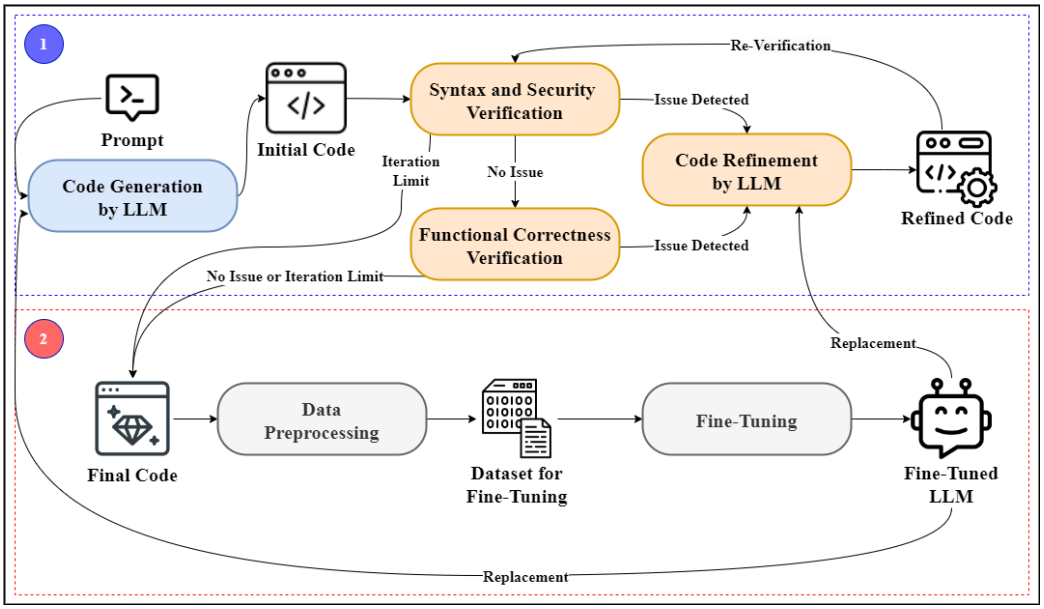


Figure 3.1: Overall architecture of CodeEnhancer, illustrating the relationship between the code generation & verification refinement pipeline (Stage 1) and the fine-tuning methodology (Stage 2).

In the first stage, CodeEnhancer employs an integrated pipeline that generates candidate code snippets using LLMs and immediately subjects them to a sequence of automated verification steps. These steps include

syntax checking, security vulnerability analysis with SAST tools, and functional correctness verification. Feedback from these verification steps is automatically fed back into the generation process, prompting the LLM to iteratively refine the code.

In the second stage, the system further improves the baseline model's ability to generate secure and reliable code by fine-tuning the LLM using a dataset composed of verified and refined code from Stage 1. This two-stage, closed-loop process not only increases the robustness and reliability of generated code in the short term but also enables the long-term evolution of the LLM toward higher coding standards.

Figure 3.1 illustrates the overall architecture, showing how these stages interact to create a continuously improving cycle of code generation, validation, and model enhancement. This modular and extensible design ensures that CodeEnhancer can be adapted to new coding domains, security standards, and advances in LLM technologies.

3.2 Stage 1: Code Generation, Verification & Refinement

The first stage of the CodeEnhancer framework is dedicated to the robust and reliable generation of Python code through a multi-step, feedback-driven process. At its core, Stage 1 integrates the strengths of both LLMs and SAST tools by introducing a rigorous pipeline that combines code generation, automated verification, and iterative correction.

Initially, the LLM generates code based on a structured prompt that encodes the intended functionality in clear terms. This code is then subjected to a sequence of automated checks, each targeting a distinct aspect of code quality: syntactic correctness, security, and alignment with the user's intended functionality. Whenever an issue is identified in any of these dimensions, detailed feedback is formulated and provided to the LLM, which then revises its output accordingly. This process repeats, with each iteration bringing the code closer to the required standards.

By this approach, Stage 1 is able to systematically eliminate common sources of error in LLM-generated code, such as syntax mistakes, overlooked security vulnerabilities, or subtle deviations from user intent. This iterative methodology ensures that the resulting code is not only executable and safe, but also meets the exact requirements specified in the prompt.

3.2.1 Code Generation with Structured Prompting

LLMs often produce syntactically correct yet structurally inconsistent or incomplete code when prompted with free-form instructions [3]. To address this, our framework adopts a structured prompting approach that demonstrates the desired output format and enforces a consistent generation template across diverse tasks.

The process initiates with a natural language prompt provided by the user, describing the desired code functionality. To enhance clarity and facilitate subsequent verification, we employ a structured prompt template that explicitly instructs the LLM to include not only the functional code but also a detailed docstring. This structured prompt is designed to elicit from the LLM not only the functional code but also a detailed docstring that clearly restates the user's request, specifies the purpose of the code, and describes the steps required to achieve the intended functionality. The LLM processes this prompt to generate an initial Python code snippet along with the embedded docstring, which serves as a functional specification.

The core idea is to guide the LLM to include a module-level docstring at the top of each Python file, formatted with three required fields:

1. **Input Prompt** – a restatement of the original user request.
2. **Intention** – the purpose or goal of the function.
3. **Functionality** – a precise description of how the function achieves that goal.

By requiring this explicit structure, the framework ensures that the generated code is not only easier for humans to understand and review but is also machine-readable for downstream evaluation tasks. For example, automated functional correctness checks can use the detailed docstring as a specification to semantically compare against the actual code logic. This structured prompting approach also reduces ambiguity, supports more reliable iterative improvement, and encourages the LLM to generate code that is directly aligned with the original intent of the user.

3.2.2 Syntax, Security & Functional Correctness Verification

The Code verification component iteratively refines the quality and security of LLM-generated code by detecting and correcting errors across multiple dimensions. This process integrates LLMs with SAST tools using three verification components, as follows:

1. **Syntax Error Verification:** The framework employs Pylint to verify

the syntactic correctness and structure of the generated code. This includes identifying missing or misplaced elements such as parentheses, indentation inconsistencies, or improper use of reserved keywords. Identified issues are logged, and corrective prompts are sent to the LLM, which iteratively refines the code to align with these standards. This ensures that the generated code is both executable and adheres to basic syntactic conventions. Figure 3.2 provides an example of a syntax issue detected by Pylint, illustrating the type of feedback used for iterative correction.

Pylint Issue

E0001: Parsing failed:
'invalid decimal literal (Code, line 4)'
(syntax-error)

Figure 3.2: Example of a code syntax issue detected via Pylint.

2. **Vulnerability Verification:** To ensure security, the framework integrates Bandit, a SAST tool specifically designed for Python. Bandit scans the code for vulnerabilities, such as command injections, insecure data storage, or improper use of sensitive configurations like hardcoded passwords. Each detected vulnerability is linked to a specific CWE identifier for precise categorization and tracking. When vulnerabilities are detected, they are logged with detailed descriptions, including CWE identifiers and severity levels. The framework then prompts the LLM to revise the code accordingly, aiming to eliminate the detected issues while preserving the original functionality. By leveraging the functionality of Bandit, the system proactively reduces the risk of having security vulnerabilities. Figure 3.3 shows an example of a security issue identified by Bandit, which serves as the basis for the targeted code correction through iterative refinement.
3. **Functional Correctness Verification:** Ensuring that the code performs as intended is crucial for practical deployment. For this purpose, the LLM compares the generated code's logic against its associated docstring. The docstring serves as a functional specification, and the LLM identifies discrepancies by parsing the described behavior and matching it with the implemented logic. To avoid conflicts between functional correctness and security, the verification does not enforce exact implementation details. Instead, it considers the code correct

Bandit Issue


```
"code": "os.system('ls')",
"filename": "test.py",
"issue_confidence": "HIGH",
"issue_severity": "MEDIUM",
"issue_text": "Use of os.system detected.",
"line_number": 12,
"test_name": "os_system",
"test_id": "B605".
```


Figure 3.3: Example of a code vulnerability issue detected via Bandit.

as long as it achieves the same intended behavior, even if different libraries or algorithms are used to improve security or performance. For example, if the docstring states that the function fetches user data from a database, yet the implemented code only contains a placeholder without actual database queries, the framework flags this mismatch during verification. The LLM then autonomously modifies the code to align it with the specified requirements. This step not only verifies the functional correctness but also improves the code's adherence to intended design and usability standards. As illustrated in Figure 3.4, if the code logic does not align with the functional description provided in the docstring, the framework can automatically detect and address such mismatches.

3.2.3 Iterative Refinement

If any issues are detected during the validation stage, the findings from Pylint, Bandit, and the functional correctness check are collected and formatted into a structured feedback report. This report typically includes:

1. The specific tool that flagged the issue
2. The code snippet containing the issue
3. An explanation of the issue, as provided by the tool or the LLM

This structured feedback is crucial for guiding the LLM effectively in the refinement step. The feedback, along with the original prompt and the current (flawed) code version, is passed back to the LLM. The LLM is instructed to revise the code specifically to address the reported issues while preserving the original docstring and intended functionality. This generate-

Functional Correctness Issue

Docstring:
Input Prompt: Write a function that returns the sum of two numbers.
Intention: To compute and return the sum of a and b.
Functionality: This function takes two numbers as input and returns their sum.

Code:

```
def add_numbers(a, b):
    return a - b
```

LLM Response:
Incorrect. The code subtracts b from a, but the docstring says it should return the sum.

Figure 3.4: Example of a functional correctness issue detected via LLM

verify-refine cycle repeats iteratively. The loop terminates under one of the following conditions:

1. All the validation steps (syntax, security, functional correctness) are passed successfully.
2. A predefined maximum number of iterations is reached.

This iterative process allows the framework to progressively improve the code. The final output of this stage is either a verified code snippet or the best version achieved within the iteration limit.

3.3 Stage 2: LLM Fine-Tuning for Code Enhancement

While the iterative verification and correction process in Stage 1 ensures the immediate quality and security of generated code, Stage 2 is focused on achieving sustainable, long-term improvements in the LLM's inherent coding capabilities. Here, the framework harnesses the validated output of Stage 1 to build a dataset specifically tailored for supervised fine-tuning of the LLM. This approach eliminates the need for manual dataset construction or expert annotation, instead leveraging the framework's own rigorous standards to curate a repository of secure, functionally correct code paired with clear prompt specifications.

In Stage 2, the fine-tuned models are trained to internalize the patterns of secure coding, robust error handling, and alignment with user intent as demonstrated in the validated code corpus. This not only reduces the frequency of critical errors in future code generations but also decreases reliance on costly, repeated verification cycles. As the fine-tuned LLM is re-integrated into Stage 1, the framework is able to close the loop and iteratively boost the quality and security of code produced in subsequent iterations. Over time, this process drives the evolution of the model toward becoming a more reliable and security-aware coding assistant.

3.3.1 Dataset Preparation and Model Training

A key innovation of the CodeEnhancer framework lies in its fully automated, quality-assured dataset construction process for model fine-tuning. Rather than relying on labor-intensive manual curation or external expert input, the system systematically gathers input prompts and their corresponding code outputs from Stage 1, but includes in the fine-tuning dataset only those code samples that have successfully passed all layers of automated validation—syntax, security, and functional correctness. This ensures that every training example reflects best practices in both secure programming and clear functional specification, as encoded in the structured docstring.

To support both comprehensive evaluation and targeted ablation studies, two distinct fine-tuning datasets are constructed. The first dataset consists of pairs of input prompts and the final, framework-refined code, with docstrings optionally excluded to focus the model on implementation quality. The second dataset pairs the same prompts with expert-written, security-vetted code samples sourced from the LLMSecEval benchmark. This dual-dataset strategy enables direct comparison between models fine-tuned on machine-refined versus expert-crafted code, shedding light on the efficacy and limitations of automated code improvement pipelines.

Automating dataset creation in this way not only increases efficiency and scalability, but also enhances reproducibility and reduces human bias. By continuously updating the fine-tuning corpus with the latest high-quality outputs from Stage 1, the framework is able to rapidly adapt to new coding styles, security standards, and emerging user requirements.

3.3.2 Fine-Tuning Strategy

After fine-tuning, the newly trained models are seamlessly integrated back into the CodeEnhancer pipeline, taking the place of the previous baseline LLMs for future code generation tasks. This closed-loop strategy allows

the framework to benefit from continual, data-driven improvements in code quality and security as the model is exposed to ever-more rigorous standards.

We anticipate that models fine-tuned on framework-refined code will demonstrate a clear advantage over those trained solely on expert datasets or left unrefined, both in terms of initial code quality and in their ability to avoid common vulnerabilities and functional errors. This, in turn, is expected to reduce the computational and human resources needed for repeated correction cycles. In Chapter 4, we will provide a comparative analysis of baseline, framework-fine-tuned, and expert-fine-tuned models, measuring improvements across multiple dimensions, including security compliance, functional correctness, and overall code reliability. This iterative fine-tuning and redeployment process enables the CodeEnhancer framework to maintain a high standard of performance as both security and functional correctness change over time.

Chapter 4

Experimental Results

4.1 Preliminary Experiment

The main objective of this preliminary experiment was to evaluate whether explicitly prompting LLMs to "generate secure code" actually leads to safer code being produced. Building upon the methodology introduced by Tony et al. [17], who demonstrated that explicit secure coding prompts and iterative self-improvement strategies can reduce vulnerabilities in code generated by GPT-3, GPT-3.5, and GPT-4, this study aims to verify whether similar prompting techniques are also effective for GPT-4o. Specifically, we assess if instructing GPT-4o with security-focused prompts results in more secure code, thus extending previous findings to the latest generation of LLMs. Based on the findings of this prior work, we additionally incorporate explicit secure coding instructions into the code generation stage of our CodeEnhancer framework to further investigate the impact of such prompts in a practical pipeline setting.

4.1.1 Datasets

The LLMSecEval dataset [12] is employed in the preliminary experiment. This dataset is specifically constructed to evaluate the security and robustness of LLM-generated code. This dataset is based on the MITRE CWE Top 25 Most Dangerous Software Weaknesses and covers 21 distinct CWE categories, including common issues such as buffer overflows (CWE-120), command injection (CWE-77), and improper input verification (CWE-20). It contains 150 carefully designed natural language prompts, each crafted to induce specific coding vulnerabilities in Python when processed by LLMs.

Figure 4.1 shows an example of a natural language prompt from this dataset. These prompts simulate real-world development scenarios where insecure code may be inadvertently generated.

LLMSecEval Prompt Example

Generate Python code for the following:
The code reads a value from the command line and adds 1000 to it.

Figure 4.1: Example of a natural language prompt from the LLMSecEval dataset.

4.1.2 Methodology

For the experiment, we compared two groups of prompts. The baseline group used the original LLMSecEval prompts to generate Python code. The enhanced group used modified prompts, in which we explicitly instructed the model to generate code that is both free from syntax errors and secure against vulnerabilities. Using these two prompt settings, we generated code for all the 150 tasks in the LLMSecEval benchmark and compared the results between the two groups.

4.1.3 Results and Analysis

Figures 4.2 and 4.3 show that, out of 150 prompts, the baseline group (original prompts) produced 64 vulnerable code snippets (42.7%), in which a total of 74 CWE vulnerability instances were detected. In comparison, the enhanced group (enhanced prompts) generated 66 vulnerable code snippets (44.0%), with a total of 79 CWE vulnerability instances identified. Notably, the enhanced group not only resulted in a higher number of vulnerable code snippets, but also exhibited an increase in the total number of vulnerabilities. Furthermore, the enhanced group included additional CWEs, such as CWE-605 and CWE-703, which were not found in the baseline group.

These findings indicate that simply adding explicit secure coding instructions to the prompt does not necessarily reduce the number or diversity of vulnerabilities. In some cases, it may even introduce new types of security weaknesses. This highlights the limitations of prompting-based methods and suggests the need for further measures, such as post-processing, to effectively enhance the security of LLM-generated code.

Although the enhanced group was designed to prioritize security, the results suggest that LLMs may not fully internalize or generalize secure coding practices solely from prompt instructions. The increase in both the number and diversity of vulnerabilities implies that certain types of weaknesses, such as command injection and code injection, remain difficult to eliminate

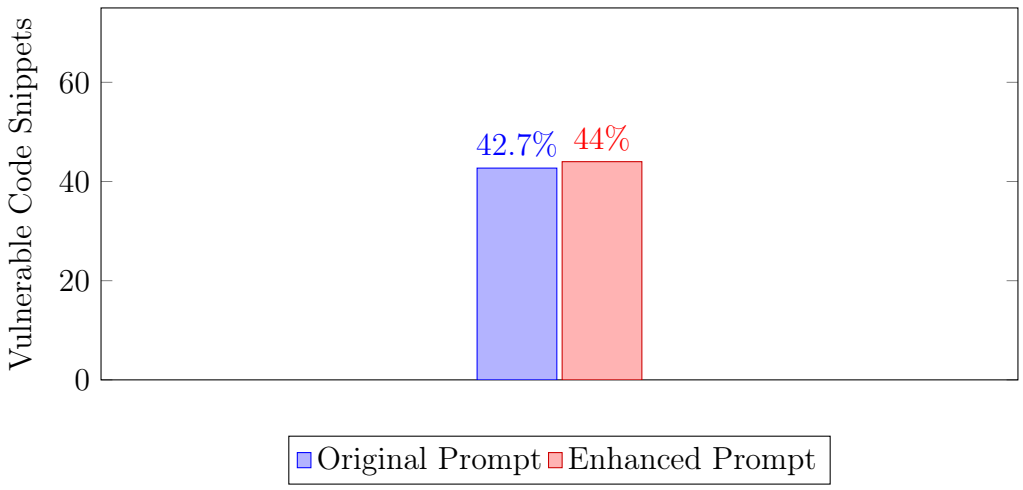


Figure 4.2: Percentage of code snippets with vulnerability issues for the baseline and enhanced groups (LLMSecEval, 150 samples).

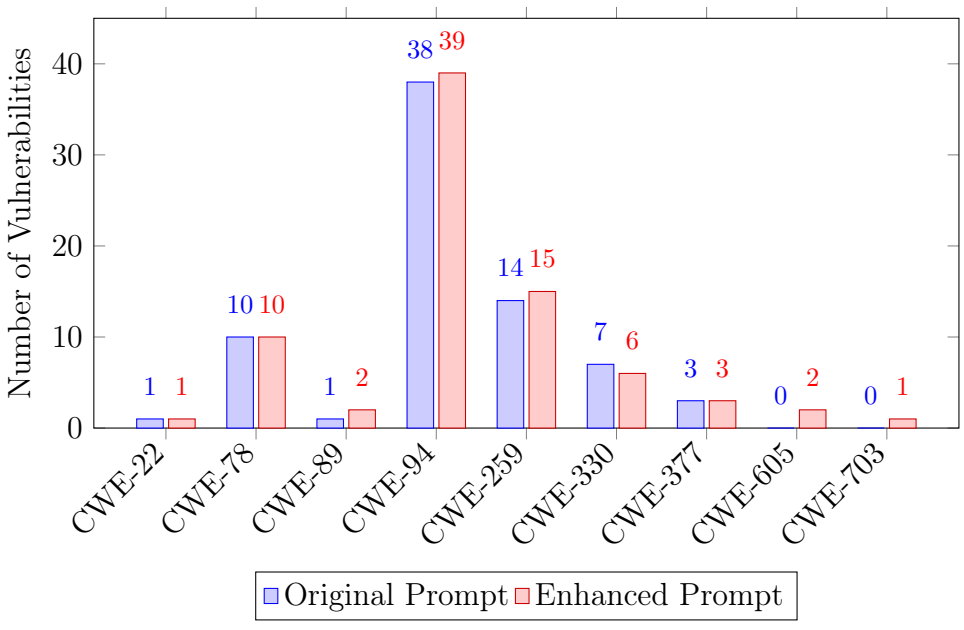


Figure 4.3: Distribution of vulnerabilities per CWE for the original and enhanced prompt groups (LLMSecEval, 150 samples).

with prompting alone. In some cases, the explicit instruction to generate secure code may have led to unintended logic changes or alternative insecure patterns. This observation highlights the challenge of relying exclusively on natural language instructions for secure code generation.

Therefore, these results underscore the necessity of combining secure prompts with additional validation steps, such as automated static analysis or repair mechanisms, as implemented in our CodeEnhancer framework. By integrating multiple layers of security assurance, the framework aims to address both persistent and newly introduced vulnerabilities that prompting alone cannot mitigate.

4.2 Experiment Setup

This section introduces the experiment setup for the main experiments we conducted, in particular, the datasets and LLMs used, as well as the evaluation metrics. Specifically, we evaluated Stage 1 of the framework using Experiment #1, and Stage 2 using Experiment #2.

4.2.1 Datasets

In our experiments, we utilized two datasets, as described next.

LLMSecEval: As explained in Section 4.1.1, LLMSecEval is a benchmark dataset based on the MITRE CWE Top 25, comprising 150 natural language prompts targeting Python vulnerabilities. This dataset serves as the main evaluation source for both stages of our framework.

SecurityEval: For Experiment #2, we use the SecurityEval dataset [13], which covers a broader and more diversified set of security-sensitive tasks. Unlike LLMSecEval, SecurityEval targets various vulnerability types beyond the MITRE Top 25 and is designed to challenge the generalization capabilities of LLMs. The dataset consists of 121 natural language prompts that span 69 distinct CWE categories, offering wide coverage across both common and less frequently addressed vulnerabilities (see example in Figure 4.4).

SecurityEval is used in Experiment #2 to provide a fair test of how well the models can handle new and different tasks. Since the models were fine-tuned using outputs from LLMSecEval, evaluating them on the same dataset could make the results less meaningful. SecurityEval introduces new prompts and more diverse vulnerabilities, making sure the test reflects new real-world situations and shows how well the models can adapt to new challenges.

SecurityEval Prompt Example

```
import yaml
def yaml_load(filename):
    '''
    use the filename variable to open a file,
    load its contents into a variable with yaml,
    and return it
    '''
```

Figure 4.4: Example of a natural language prompt from the SecurityEval dataset.

4.2.2 LLM Configurations

We used different LLM models in each experiment, as explained next.

Baseline Model (Experiment #1): In this study, we employed GPT-4o for both code generation and verification tasks. GPT-4o was chosen for its advanced performance, versatility, and state-of-the-art capabilities in handling diverse coding scenarios. Although LLMSecEval was originally tested using a GPT-3 based model (code-davinci-002, now discontinued), for our framework we used GPT-4o, which offers superior contextual understanding and iterative refinement capabilities, making it ideal for generating secure and functional Python code. Using GPT-4o, the framework ensures the generation of secure, functional, and high-quality Python code, addressing diverse vulnerabilities and improving practical usability.

Baseline and Fine-Tuned Models (Experiment #2): In Experiment #2, we evaluated three model variants:

1. **Baseline**: Basic GPT-4o model, used as a baseline for comparison.
2. **Fine-Tuned Model Using Expert Examples**: GPT-4o fine-tuned on expert-written secure code examples provided in the LLMSecEval dataset.
3. **Fine-Tuned Model Using Framework Output**: GPT-4o fine-tuned on the outputs refined through our verification pipeline in Experiment #1 using LLMSecEval.

For each model, we executed our full framework five times, resulting in five sets of code outputs before and after refinement.

4.2.3 Evaluation Metrics

Multiple evaluation criteria are used to assess the models comprehensively. In all cases, the iterative refinement process is performed with a maximum limit of 5 iterations.

Syntax Errors: We use Pylint strictly for detecting syntax errors in the generated code, not for PEP-8 style checks.

Vulnerabilities: We use Bandit to identify security vulnerabilities according to the CWE categories, focusing on the number of detected vulnerabilities and their resolution rates after refinement.

Functional Correctness: Functional correctness is first manually evaluated by inspecting whether the code accurately implements the intended functionality described in the prompt and the embedded docstring.

In addition, we employ three advanced metrics to assess the quality of the generated code:

1. **CodeBLEU**: Measures syntactic and semantic similarity based on n-gram match, AST structure similarity, and data flow consistency [35].
2. **CodeBERTScore**: Uses transformer-based contextual embeddings to assess semantic alignment between the generated code and the reference [36].
3. **ICE-SCORE**: Evaluates how well the generated code complies with the given instructions and its likelihood of being executable, using an LLM-based review system [40]. It uses two separate 0–4 scales to independently assess usefulness (how helpful the code is for solving the task) and functional correctness (how accurately the code implements the intended behavior).

4.3 Experiment #1: Code Generation, Verification & Refinement

The Experiment #1 evaluation was conducted on 150 Python code snippets, each generated using prompts from the LLMSecEval dataset. These prompts are specifically designed to test an LLM's ability to handle security-sensitive tasks aligned with CWE categories.

4.3.1 Syntax Error Verification

No syntax errors were detected in any of the 150 Python code snippets generated during experiment #1 of our framework evaluation. While preliminary experiments using earlier versions of LLMs had occasionally resulted in syntax issues such as indentation errors or missing delimiters, all code generated using the GPT-4o model in this study passed syntax verification on the first attempt. This suggests that GPT-4o has improved capabilities in producing syntactically correct Python code when guided by a well-structured prompt and docstring.

4.3.2 Vulnerability Verification

During the vulnerability verification phase, a total of 64 code snippets with vulnerabilities were detected in the initial code generated from the LLMSecEval dataset. After the iterative refinement process, this number was reduced to 11, corresponding to a resolution rate of 82.8%, as illustrated in Figure 4.5. This result highlights the effectiveness of our verification framework in addressing the majority of security issues through automated feedback and code correction.

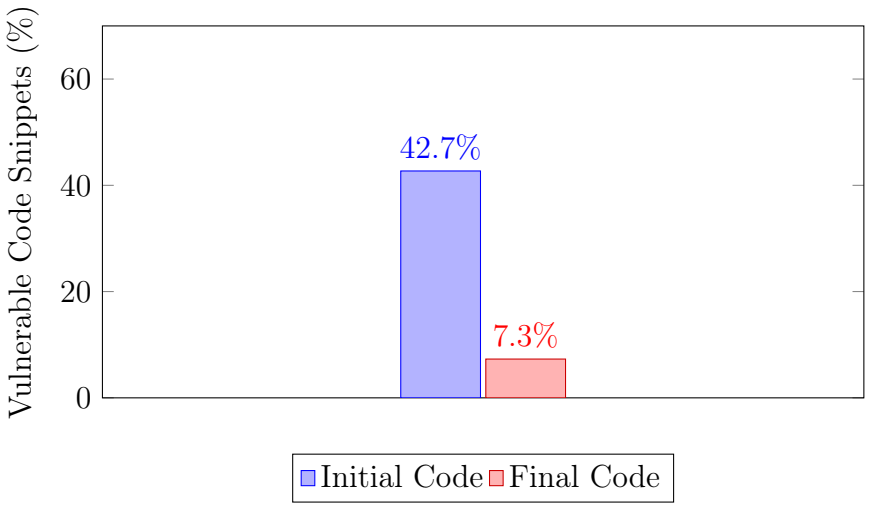


Figure 4.5: Percentage of code snippets with vulnerability issues before and after iterative refinement (LLMSecEval, 150 samples).

Our analysis showed that the unresolved CWE-78 vulnerabilities after iterative refinement were primarily caused by the use of Python's `subprocess` module. This module, often used for executing shell commands, introduces

risks when user inputs are directly concatenated into commands. Although Bandit flagged the use of the `subprocess` module with warnings, the framework failed to address these vulnerabilities effectively by enforcing strict verification of user inputs. The lack of input sanitization allowed these security issues to persist, underscoring the critical importance of incorporating comprehensive input verification mechanisms to mitigate risks associated with shell command execution.

Interestingly, Bandit's analysis of secure code examples written by experts from LLMSecEval revealed similar CWE-78 issues in cases where the `subprocess` module was employed. In these instances, even expert-designed secure code examples were flagged with the same errors as the framework's generated code. Upon further inspection, it was observed that both the refined code snippets and the secure examples often adopted similar approaches to mitigate these vulnerabilities. These approaches typically included isolating user inputs or implementing explicit whitelisting mechanisms to reduce risks. While these refinements improved the overall security posture, they did not entirely eliminate the vulnerabilities, highlighting the limitations of the current strategies in fully addressing CWE-78 issues when the `subprocess` module is involved.

4.3.3 Functional Correctness Verification

As illustrated in Figure 4.6, among the 150 code snippets generated and evaluated, the LLM itself identified 7 cases as having potential functional correctness issues during the iterative verification process due to inconsistencies between the implemented logic and the corresponding docstring, i.e., 4.7% of the total. This self-diagnosis mechanism illustrates the model's capacity to reason about its own outputs through prompt-guided verification. Notably, through iterative refinement, codes with issues were successfully resolved within several feedback iterations. This indicates that the framework is not only effective in detecting subtle discrepancies but also capable of resolving a majority of them autonomously, without relying on external feedback.

In particular, the types of issues detected by the LLM included missing essential module imports, the need for improved input handling for enhanced functionality, and the use of more appropriate modules in place of suboptimal choices. In each case, the LLM attempted to revise the implementation to better align with the requirements stated in the docstring. These findings highlight the utility of our approach for both identifying and incrementally improving code quality based on explicit functional specifications.

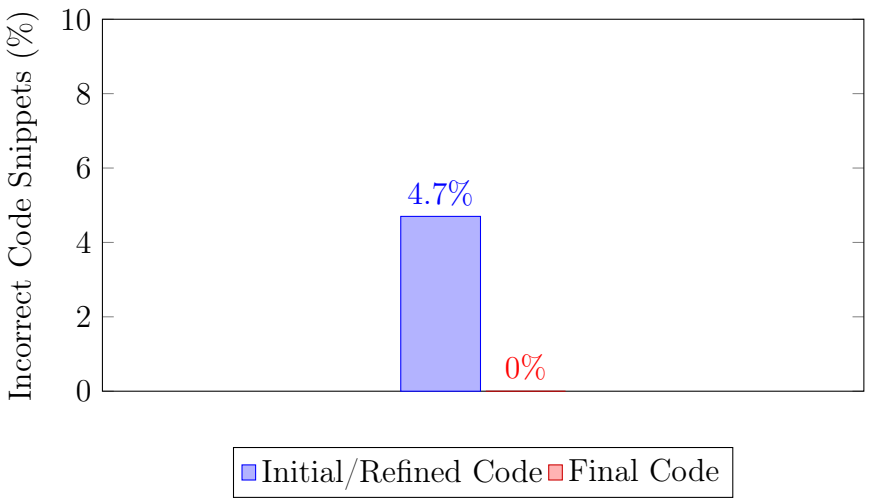


Figure 4.6: Percentage of code snippets with functional correctness issues before and after iterative refinement (LLMSecEval, 150 samples).

4.3.4 Quantitative Results Using Code-Level Metrics

In addition, we conducted a quantitative evaluation by using three code-level metrics, as follows.

CodeBLEU Evaluation: To further evaluate the impact of the code verification& refinement stage, we compared the CodeBLEU scores of code snippets. The comparison was conducted across three categories: the initial code generated before the iterative refinement process, the refined code after completing the iterative refinement process, and the secure code examples from the LLMSecEval dataset, which served as benchmarks.

As shown in Table 4.1, the CodeBLEU score improved slightly after the iterative refinement process. A notable increase was observed in the dataflow match metric. Importantly, when analyzing all 150 code snippets individually, the majority of the CodeBLEU evaluation metrics showed an improvement after the iterative refinement process, with some metrics remaining consistent. These results suggest that the refinement process brought the generated code closer to the secure code examples in terms of syntactic and functional correctness, providing evidence of a consistent enhancement throughout the verification/refinement process.

While the overall CodeBLEU scores were relatively low, this can be attributed to the nature of the secure code examples used as benchmarks. As described in earlier sections, these examples were manually refined by human

experts based on outputs from the code-davinci-002 model, leading to high-quality reference implementations. In contrast, our framework generates code based on GPT-4o, which inherently introduces architectural and stylistic differences. Consequently, achieving a high overlap with these expert-crafted samples, which were derived from a different base model, presents a significant challenge for automatically generated code. This challenge is further amplified when the reference and test samples are produced by entirely different models, which often exhibit divergent coding styles and architectural tendencies. Therefore, directly fine-tuning the model on expert-written code from a different base model may not only limit the effectiveness of the adaptation, but could also introduce unexpected issues, such as hallucinations or inconsistencies in the generated outputs.

Table 4.1: Experiment #1: Average CodeBLEU score comparison of secure code examples versus initial/final code snippets (LLMSecEval, 150 samples).

Metric	**Initial Code**	**Final Code**
N-gram Match Score	0.074	0.073
Weighted N-gram Match Score	0.091	0.091
Syntax Match Score	0.418	0.421
DataFlow Match Score	0.366	0.381

CodeBERTScore Evaluation: Additionally, we further evaluated the effects of the code verification stage using CodeBERTScore metrics on code snippets. This evaluation considered three categories: the initial outputs before iterative refinement, the refined outputs after the process, and the expert-provided secure code examples from the LLMSecEval dataset as benchmarks.

Table 4.2 presents a detailed CodeBERTScore comparison among three pairs: (1) secure expert-written code versus initial generated code, (2) secure expert-written code versus final refined code, and (3) initial versus final code snippets. The results indicate that the similarity between the initial and final code snippets is extremely high, showing that most changes during refinement are minimal and localized. When comparing both the initial and final outputs to the secure code examples, the scores remain similar, suggesting that the refinement process brings only marginal improvements in terms of overall semantic similarity to the expert-written code. However, recall and F3 metrics show a very slight increase after refinement, indicating a small gain in alignment with expert code after the iterative process. These results demonstrate that while the refinement process enhances certain aspects of

code quality, it primarily preserves the original structure and only makes minor, targeted corrections rather than large-scale modifications.

Furthermore, a manual review comparing the refined outputs with the initial outputs highlighted the framework's ability to incorporate additional functionalities, such as enhanced verification mechanisms, improved robustness, and more comprehensive input handling. Importantly, even after the code verification experiment, the refined code snippets retained a level of functional and semantic consistency comparable to expert-designed secure code examples, while offering extended capabilities beyond those included in the expert examples.

Table 4.2: Experiment #1: Average CodeBERTScore comparison of secure code examples versus initial/final code snippets, and initial versus final code snippets (LLMSecEval, 150 samples).

Metric	**Secure vs. Initial**	**Secure vs. Final**	**Initial vs. Final**
Precision	0.820	0.816	0.988
Recall	0.821	0.822	0.994
F1	0.820	0.819	0.991
F3	0.820	0.821	0.994

While this finding underscores the strength of LLMs in applying precise and minimal edits during refinement, it also aligns with the limitations discussed in Section 4.3.2. Specifically, the framework's tendency to preserve the original code structure may contribute to its inability to resolve more deeply embedded or architectural security issues—such as those observed in unresolved CWE-78 cases. These results highlight a key trade-off in the refinement process: balancing minimal intervention with the need for more comprehensive structural transformation when required.

ICE-SCORE Evaluation: To further assess how well each code variant fulfilled the instructional intent conveyed through the input prompts, we employed ICE-SCORE metrics on both the initial code and final code across the full set of 150 code snippets.

As shown in Table 4.3, both the initial and final codes achieved nearly perfect scores, with only a very slight decrease observed after refinement. This suggests that, although iterative refinement involved additional actions, such as patching vulnerabilities or adding exception handling, these modifications did not adversely impact the code's overall alignment with the original prompt or its usefulness.

Table 4.3: Experiment #1: Average ICE-SCORE comparison of initial versus final code snippets (LLMSecEval, 150 samples).

Metric	Initial Code	Final Code
Functional Correctness	3.987	3.953
Usefulness	3.960	3.893

Remarks: Overall, our quantitative results using code-level metrics demonstrate that the refinement process was able to address security concerns and enhance robustness without compromising the intended functionality or instructional fidelity of the code outputs.

However, these results also highlight an important limitation: existing evaluation metrics alone cannot fully capture the effectiveness of our framework. While improvements are observable in standard code-level metrics, some nuanced aspects of security, code robustness, and instruction alignment may not be fully reflected by current quantitative measures. Therefore, there remains a need for more comprehensive or specialized evaluation metrics to accurately assess the broader benefits and subtleties of our approach.

4.4 Experiment #2: LLM Fine-Tuning for Code Enhancement

Experiment #2 investigates the effectiveness of fine-tuned LLMs using different training sources to improve the intrinsic security and quality of generated Python code. While Experiment #1 validated the performance of an unmodified GPT-4o model using the LLMSecEval dataset, Experiment #2 employs both the LLMSecEval and SecurityEval datasets for evaluation. LLMSecEval continues to play a key role, as the outputs generated from this dataset in Experiment #1 also serve as one of the fine-tuning sources.

To avoid the effects of the potential overfitting to the LLMSecEval dataset, SecurityEval is additionally utilized. SecurityEval provides a broader and more diverse set of prompts designed to expose security flaws across a wide range of vulnerability types, ensuring that evaluation results reflect the models' ability to generalize beyond the patterns present in the fine-tuning data. For each dataset, code generation and iterative verification refinement were performed up to five times per dataset, in accordance with our framework's methodology. This approach enables a systematic and fair comparison of model performance across both benchmarks.

Three models are compared: the baseline GPT-4o model, a variant fine-

tuned on expert-written secure code examples, and a model fine-tuned on the outputs of our own verification framework. Through this evaluation, we aim to determine which training strategy better equips LLMs to produce secure, functional, and high-quality code without relying solely on post-generation refinement mechanisms.

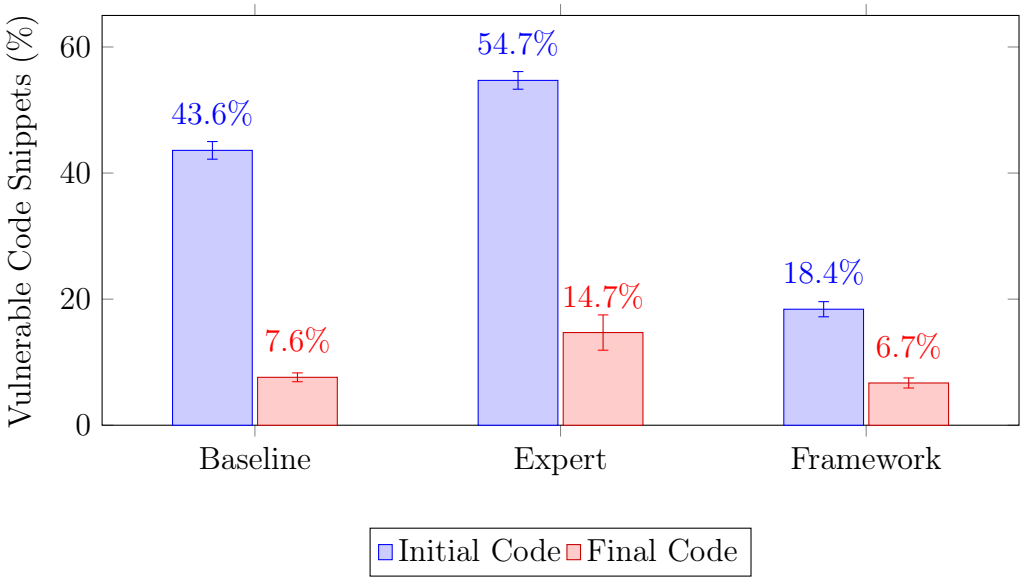
4.4.1 Syntax Error Verification

Across both the LLMSecEval and SecurityEval datasets, each prompt, after code generation, was subjected to up to five rounds of iterative verification and refinement. As discussed in Section 4.3.1, modern LLMs rarely produce syntactically invalid Python code. In our evaluation, only two code snippets exhibiting syntax errors were identified among the 121 initial outputs from the SecurityEval dataset (that is 1.7%). These errors were promptly detected and automatically corrected through the framework's integrated verification pipeline. By incorporating systematic syntax checking as an initial step, our framework ensures that even rare or minor syntactic mistakes are preemptively addressed, thus enhancing the reliability of downstream security and functional verification. This approach further reduces the likelihood of the propagation of such issues in automated code-generation workflows.

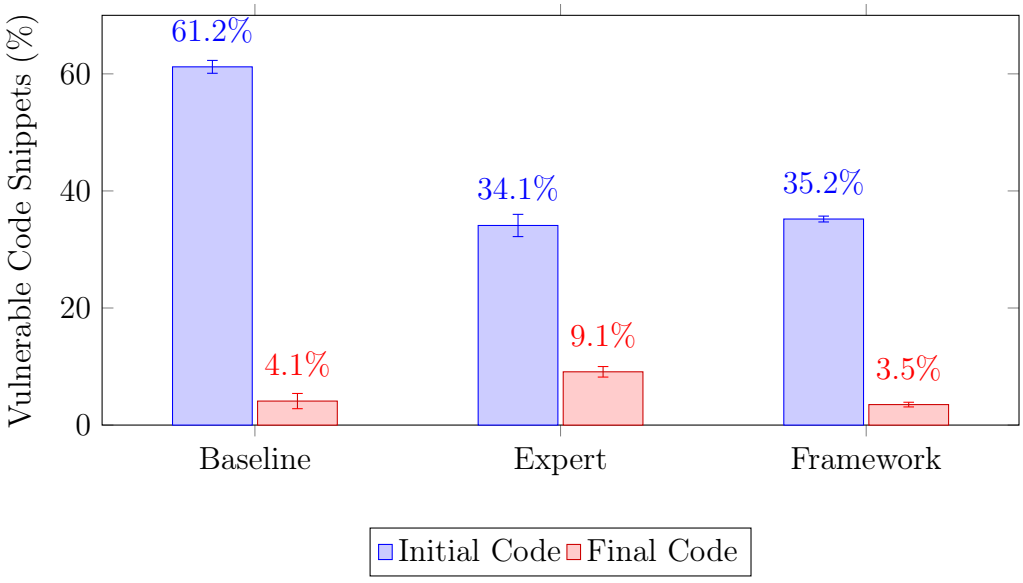
4.4.2 Vulnerability Verification

Vulnerability Reduction Across Models The vulnerability verification experiment assessed the number and proportion of code snippets containing security flaws before and after iterative refinement, as detected by Bandit. Figure 4.7 summarizes the results for both the LLMSecEval (150 code snippets) and SecurityEval (121 code snippets) datasets.

For LLMSecEval, the initial code generated by the baseline GPT-4o model contained security vulnerabilities in an average of 65.4 out of 150 code snippets (43.6%), while the expert-tuned model exhibited 82.0 (54.7%) vulnerable snippets. The framework-tuned model, reflecting the updated results, produced only 27.6 (18.4%) vulnerable code snippets on average. After five refinement cycles, these numbers were substantially reduced: the baseline GPT-4o model averaged 11.4 (7.6%) residual vulnerable code snippets, the expert-tuned model had 22.0 (14.7%) remaining vulnerable code snippets, and the framework-tuned model reached just 10.0 (6.7%) residual vulnerable code snippets. Notably, the framework-tuned model consistently produced the smallest number and proportion of security flaws, both before and after the iterative verification process.



(a) Percentage of code snippets with vulnerability issues before and after iterative refinement for each tested model (LLMSecEval, 150 samples).



(b) Percentage of code snippets with vulnerability issues before and after iterative refinement for each tested model (SecurityEval, 121 samples).

Figure 4.7: Experiment #2: Comparison of the percentage of code snippets with vulnerability issues before and after iterative refinement for each tested model when using the LLMSecEval (a) and SecurityEval (b) datasets; error bars indicate the standard deviation across five experiment runs.

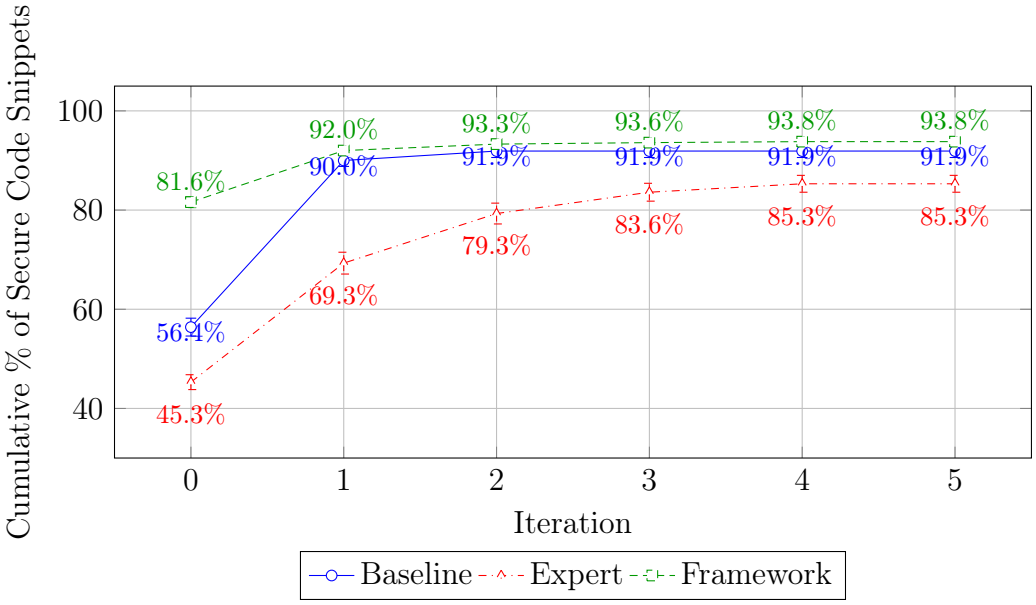
A similar trend was observed with the SecurityEval dataset. The initial outputs from the baseline GPT-4o, expert-tuned, and framework-tuned models contained an average of 74.0 (61.2%), 41.2 (34.1%), and 42.6 (35.2%) vulnerable code snippets out of 121 total samples, respectively. After refinement, the final numbers dropped to 5.0 (4.1%) for the baseline GPT-4o model, 11.0 (9.1%) for the expert-tuned model, and 4.2 (3.5%) for the framework-tuned model. Once again, the framework-tuned model achieved the lowest number and proportion of residual code snippets with vulnerability, demonstrating the effectiveness of the self-refinement approach in minimizing security issues.

Cumulative Security Improvement Figure 4.8 shows, for each model, the cumulative percentage of code snippets that are free of vulnerabilities after each refinement iteration. At iteration 0, the percentage represents the proportion of secure code in the initial code generation. With each refinement, this percentage increases as more vulnerabilities are resolved, until it eventually plateaus when no further improvements can be achieved.

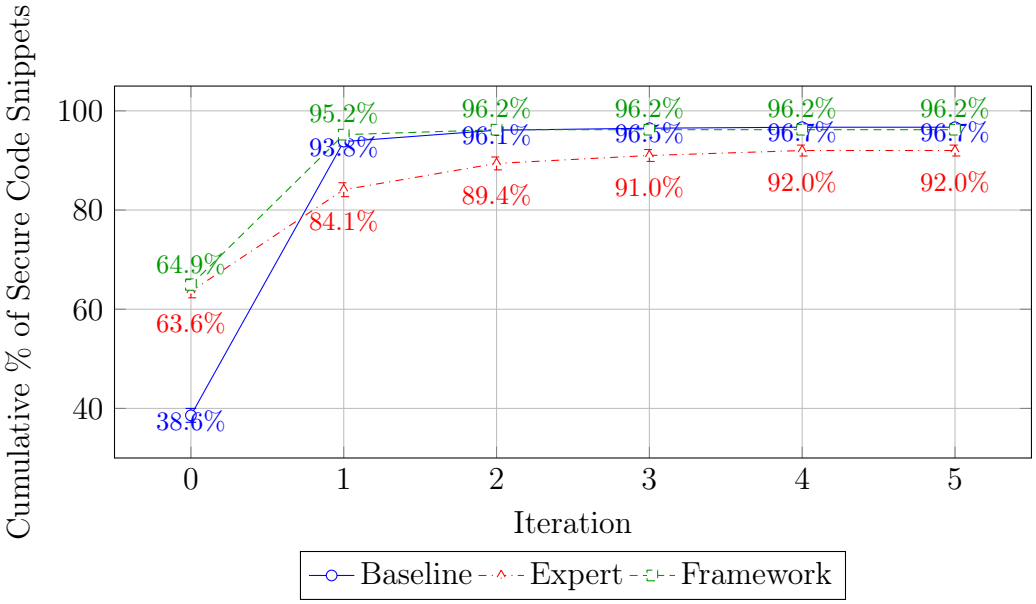
For instance, in SecurityEval, the framework-finetuned model starts with 64.9% secure code snippets at iteration 0, and quickly rises to 95.2% after just one refinement, reaching 96.2% by iteration 2 and remaining stable thereafter. The baseline model begins with a lower proportion of secure code (38.6%), but reaches a similar final level (96.7%) after five iterations. The expert-finetuned model starts at 63.6% and gradually climbs to 92.0%. These results demonstrate that the framework-finetuned model not only generates more secure code from the outset, but also achieves a high level of security with fewer refinement iterations than the other models, highlighting the efficiency and effectiveness of our framework.

Overall, the verification framework proves effective as a fine-tuning strategy, enabling LLMs to generate more secure code with fewer iterations and without relying on expert-written examples. While the expert-tuned model was trained on curated data from security professionals, it still required more revisions and left more vulnerabilities unresolved. In contrast, the framework-finetuned model learned from its own refinement process, reducing vulnerabilities proactively before verification.

This approach simplifies fine-tuning into a scalable and low-cost process, minimizing human involvement. The results suggest that self-refinement via automated verification can be a practical and competitive alternative to expert-guided fine-tuning, achieving comparable or superior outcomes without manual dataset construction.



(a) Cumulative percentage of secure code snippets after each refinement iteration (LLMSecEval, 150 samples).



(b) Cumulative percentage of secure code snippets after each refinement iteration (SecurityEval, 121 samples).

Figure 4.8: Experiment #2: Cumulative percentage of secure code snippets generated after each refinement iteration when using the LLMSecEval (a) and SecurityEval (b) datasets; error bars indicate the standard deviation across five experiment runs, and labels show the average percentage for each iteration.

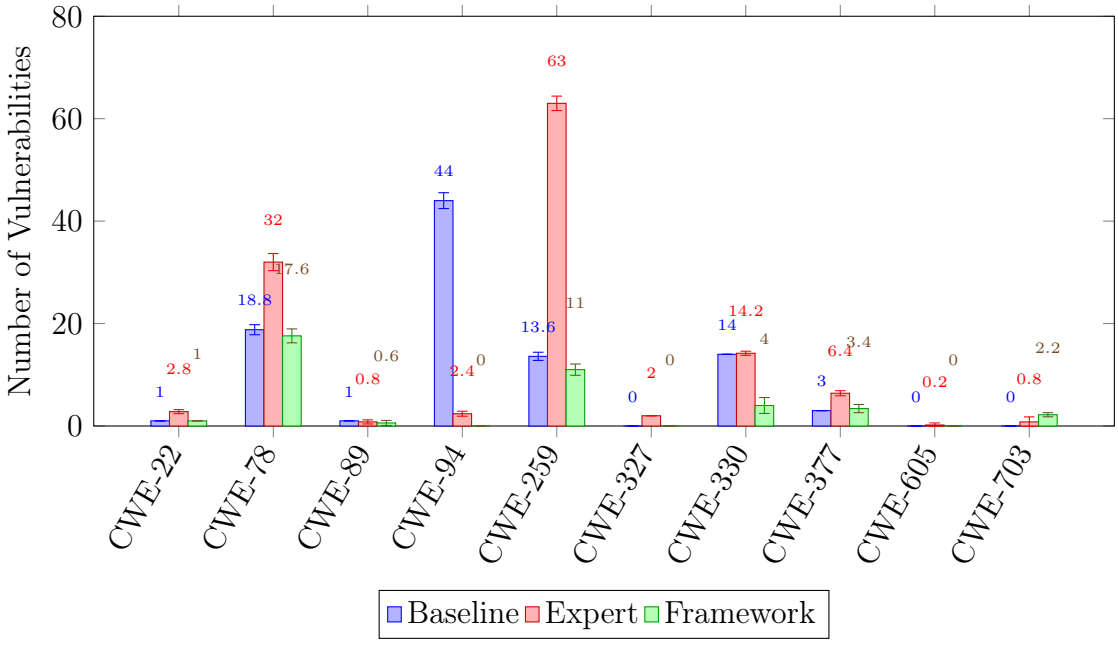
Distribution of Vulnerabilities per CWE Figure 4.9 presents the distribution of vulnerabilities per CWE before and after the refinement process for each model across the LLMSecEval and SecurityEval datasets, respectively. The analysis reveals several noteworthy trends.

Before refinement (Figure 4.9(a)), the baseline model on LLMSecEval exhibited vulnerabilities spanning seven major CWE categories, with especially high counts in CWE-94 (code injection), CWE-78 (command injection), and CWE-330 (use of insufficiently random values). The expert-tuned model, however, produced a markedly different CWE distribution: it generated a larger number of vulnerabilities in categories such as CWE-259 (hardcoded password) and CWE-78, while vulnerabilities related to CWE-94 were notably fewer than the baseline. The framework-tuned model showed an intermediate pattern, achieving substantial reduction in code injection vulnerabilities (CWE-94: 0), and overall lower vulnerability counts in several categories compared to the other models.

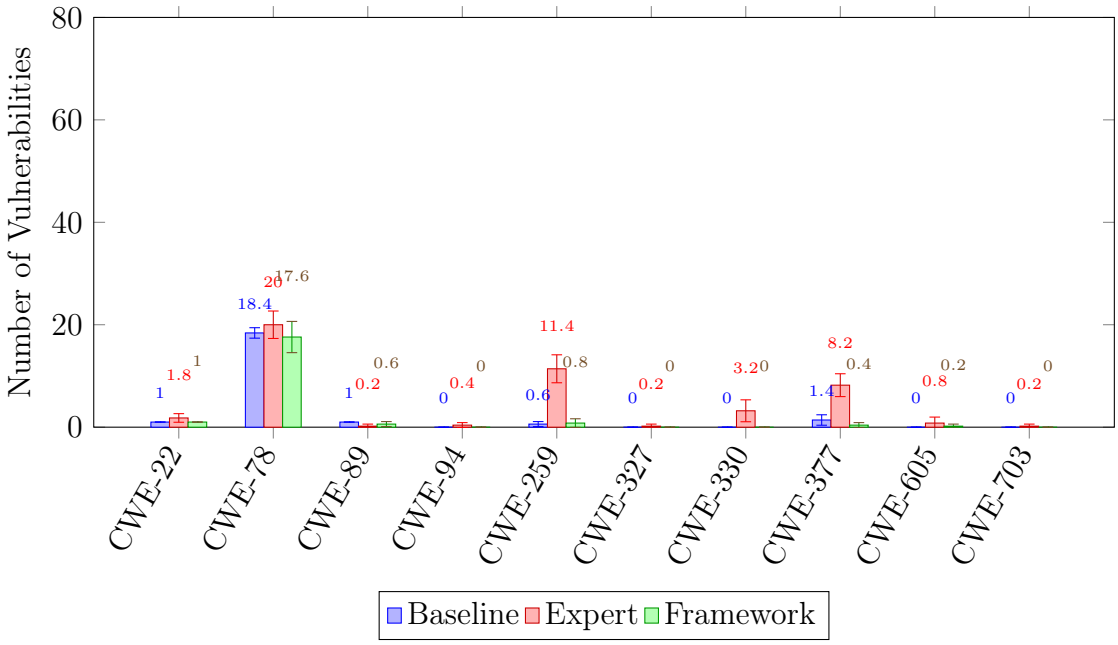
After refinement (Figure 4.9(b)), the overall number and diversity of vulnerabilities decreased across all models. Notably, for the framework-tuned model, most CWE categories were eliminated or greatly reduced—CWE-94 was fully resolved, and vulnerabilities in CWE-259, CWE-327, CWE-330, and CWE-377 were nearly eliminated. The baseline model retained residual vulnerabilities primarily in CWE-78 and a small number in CWE-22 and CWE-89. The expert-tuned model, in contrast, showed persistent issues across a broader set of CWE categories, particularly in CWE-78, CWE-259, and CWE-377, indicating that fine-tuning solely on expert code does not necessarily guarantee a broader coverage of vulnerability mitigation.

A similar pattern was observed with the SecurityEval dataset as illustrated in Figure 4.10. Before refinement, all models exhibited vulnerabilities across a diverse set of CWE categories, with command injection (CWE-94) and improper input validation (CWE-20) being the most frequent. The expert-tuned and framework-tuned models each showed unique patterns, but after refinement, the framework-tuned model consistently reduced the number and diversity of residual vulnerabilities to a greater extent than the other models. In many categories, such as CWE-94 and CWE-259, the framework-tuned model achieved near-complete or complete elimination of vulnerabilities, whereas the expert-tuned model retained more residual vulnerabilities across categories.

A closer look at the persistent vulnerabilities reveals that command injection (CWE-78) was particularly challenging for all models, even after refinement. This issue mainly arose from the use of Python's subprocess module to execute shell commands. Although Bandit explicitly flagged the use of subprocess as a potential security risk, the LLMs did not attempt to

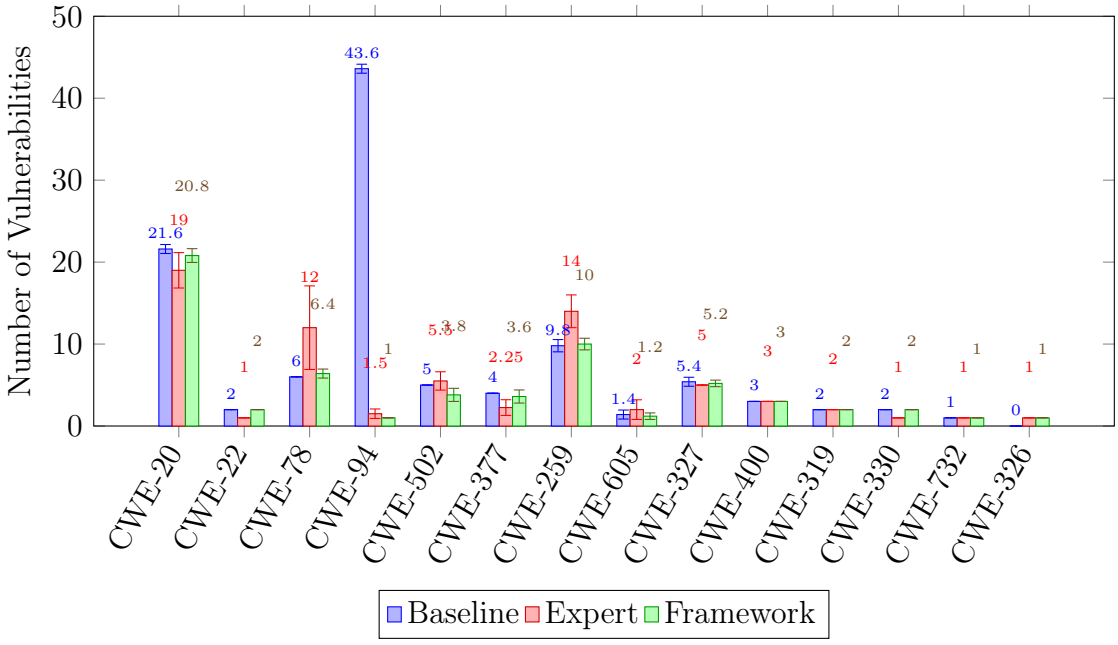


(a) CWE-type distribution for each model before refinement (LLMSecEval, 150 samples).

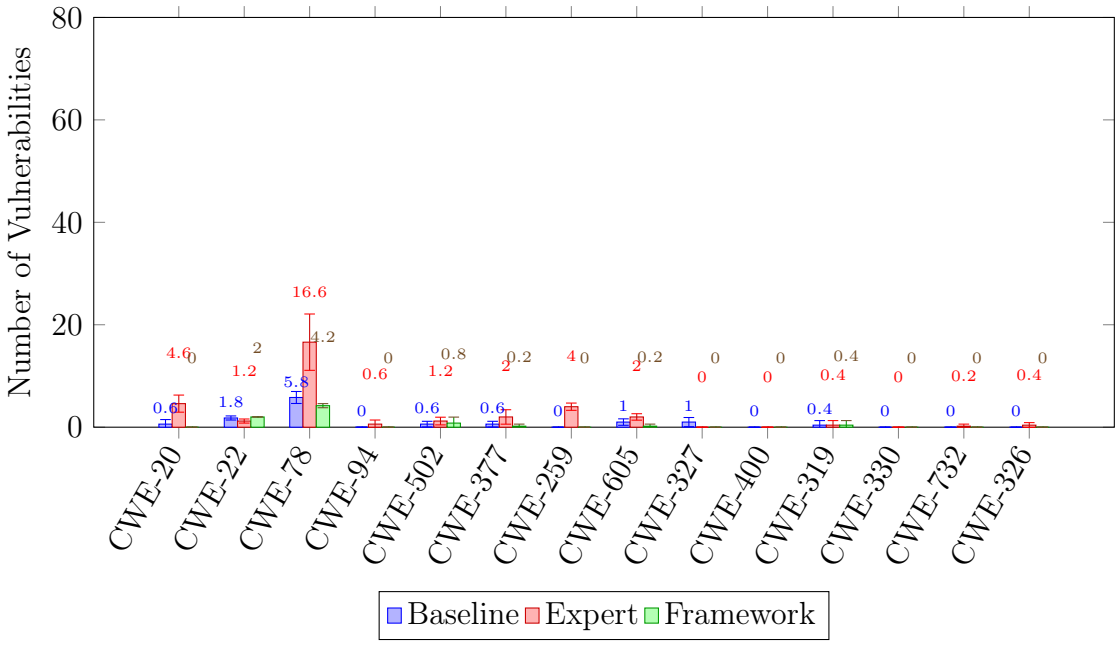


(b) CWE-type distribution for each model after refinement (LLMSecEval, 150 samples).

Figure 4.9: Experiment #2: Distribution of vulnerabilities per CWE for each model before (a) and after (b) refinement, averaged over 5 runs LLMSecEval dataset; error bars indicate standard deviation across five experiment runs.



(a) CWE-type distribution for each model before refinement (SecurityEval, 121 samples).



(b) CWE-type distribution for each model after refinement (SecurityEval, 121 samples).

Figure 4.10: Experiment #2: Distribution of vulnerabilities per CWE each model before (a) and after (b) refinement, averaged over 5 runs (SecurityEval, 121 samples); error bars indicate standard deviation across five experiment runs.

replace it with safer alternatives or introduce comprehensive input sanitization. As a result, the models often failed to resolve these vulnerabilities, and command injection issues persisted in the output code. This finding underscores the limitations of current LLM-based approaches: even when provided with explicit warnings and multiple opportunities for correction, the models struggle to fundamentally change architectural choices, such as avoiding risky modules, without direct human intervention or targeted training.

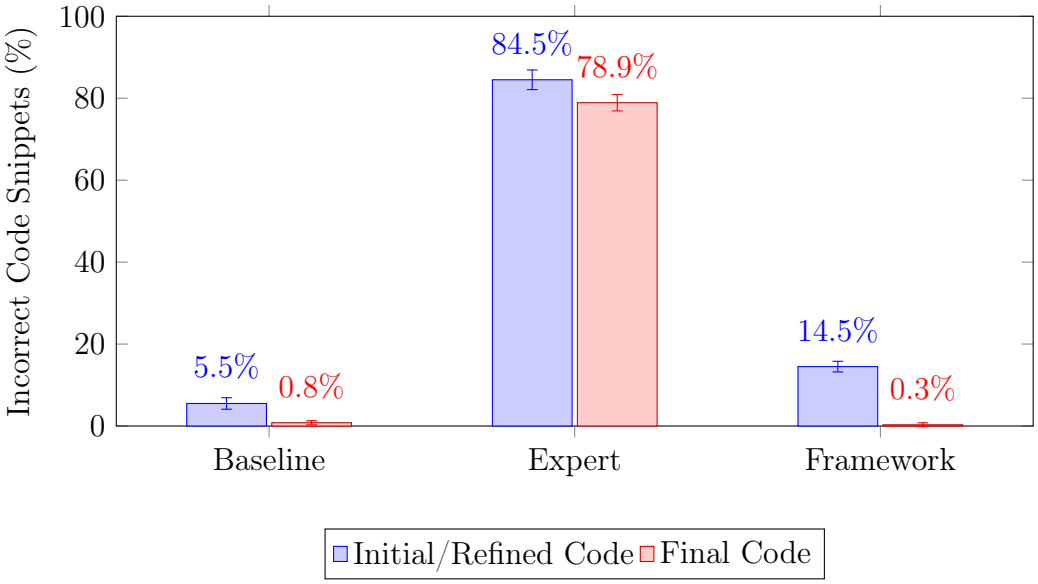
Summary Overall, these results demonstrate that the framework-tuned model is highly effective at reducing both the number and variety of security vulnerabilities after the verification and refinement process, compared to both the baseline and expert-tuned models. The persistence of certain vulnerabilities, such as command injection (CWE-78), highlights the intrinsic difficulty of completely mitigating specific classes of security issues in an automated manner. Nevertheless, the framework's ability to address a wide range of CWE categories, including those less common in the training data shows strong generalization and robustness in practical code security tasks.

4.4.3 Functional Correctness Verification

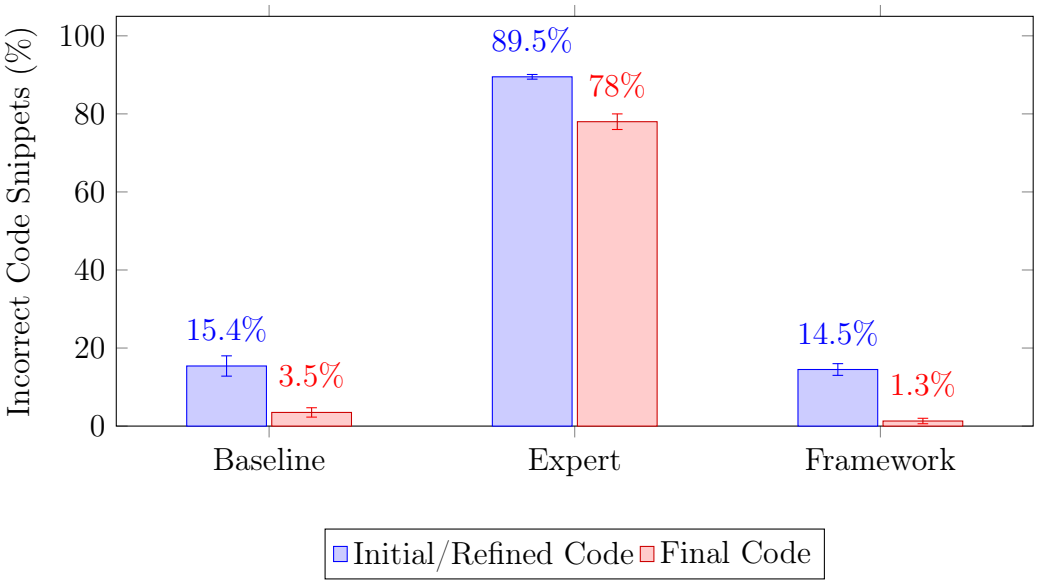
Figure 4.11 presents, for each model, the number and percentage of code snippets that were found to have functional correctness issues at any point during the initial or refinement stages ("Initial/Refined Code"), as well as those still exhibiting such issues in the final output ("Final Code"), for both the LLMSecEval and SecurityEval datasets.

For LLMSecEval (150 code snippets), the baseline model showed functional correctness errors in 8.2 cases (5.5%) during the initial or refinement stages, with only 1.2 cases (0.8%) remaining in the final output. The framework-tuned model triggered functional correctness errors in 21.8 cases (14.5%) during verification and refinement, but this number was reduced to just 0.4 cases (0.3%) in the final output, demonstrating strong self-correction capability. In contrast, the expert-tuned model exhibited a much higher number of functional correctness failures during verification and refinement (126.8 cases, 84.5%), and 118.4 cases (78.9%) still had errors in the final output, indicating little improvement.

Manual review revealed that the high count of incorrect code snippets for the expert-tuned model was largely due to its tendency to generate incomplete or partially implemented code. This led to the frequent detection of functional errors throughout the verification process, with many issues persisting in the final output.



(a) Percentage of code snippets with functional correctness issues before and after iterative refinement for each tested model (LLMSecEval, 150 samples).



(b) Percentage of code snippets with functional correctness issues before and after iterative refinement for each tested model (SecurityEval, 121 samples).

Figure 4.11: Experiment #2: Comparison of the percentage of code snippets with functional correctness issues before and after iterative refinement for each tested model when using the LLMSecEval (a) and SecurityEval (b) datasets; error bars indicate the standard deviation across five experiment runs.

For LLMSecEval, we found that 14 out of 21.8 initial/refined code snippets (9.3% of the total 150 samples) generated by the framework-tuned model exhibited functional correctness issues specifically due to missing import statements for required libraries or modules. This problem often occurred when the fine-tuned model used different or more secure libraries than the GPT-4o model but failed to include the necessary import statements. These cases represent a substantial portion of the overall functional correctness issues identified during the initial and refinement stages. Importantly, these errors were straightforward to resolve during the refinement process by automatically adding the missing import lines. We suspect that this pattern may be related to overfitting introduced during fine-tuning, but further investigation is needed to confirm the underlying cause.

A similar pattern was observed with SecurityEval (121 code snippets): the baseline model encountered functional correctness issues in 18.6 cases (15.4%) during the initial or refinement stages, with 4.2 (3.5%) remaining in the final output. The framework-tuned model had 17.6 cases (14.5%) with functional correctness errors during verification and refinement, but only 1.6 (1.3%) remaining in the final output. Once again, the expert-tuned model performed worst, with 108.2 initial or refined incorrect cases (89.5%) and 94.4 final incorrect (78.0%).

These results highlight the effectiveness of the framework-tuned model at resolving functional correctness issues encountered at any stage of verification, not just in the initial code. By contrast, the expert-tuned model, which was fine-tuned on code outputs from prior LLMs, repeatedly generated incomplete or underspecified solutions, leading to persistent errors even after several feedback cycles. Manual inspection suggests that verification-guided refinement enables the framework-tuned model to reliably address prompt requirements, quickly correcting errors as they arise and converging to correct outputs with fewer remaining failures.

Chapter 5

Discussion

This section summarizes the main limitations of our framework and outlines possible directions for future work.

5.1 Vulnerability Mitigation

While iterative refinement and fine-tuning substantially reduced the total number of vulnerabilities, certain CWE categories, such as command injection (CWE 78), remain difficult to resolve even after multiple automated refinement cycles. As discussed in Section 4.4.2, these persistent vulnerabilities frequently originate from the use of inherently risky modules.

To overcome this limitation, future work should explore targeted fine-tuning, improved prompt design, and explicit behavioral constraints. For instance, models could be further trained or instructed to prefer safer APIs, always include robust input validation, or actively reject insecure coding patterns when flagged by static analysis. Integrating dynamic analysis feedback, involving users in the correction process, or building specialized secure code generation datasets may further enhance the ability of LLMs to address these challenging cases.

5.2 Functional Correctness Evaluation

Current metrics [35, 36] mainly measure similarity to reference code, meaning these scores do not fully reflect the real quality of LLM-generated code. Our evaluation also relies on static analysis and similarity, not on comprehensive test cases. Creating exhaustive tests would be more accurate but is time-consuming and difficult to scale. Current methods also miss some semantic correctness issues, especially when multiple valid solutions exist. Another limitation arises from the use of natural language docstrings, which may lead to misinterpretation of intended functionality.

To overcome these challenges, future work could adopt pseudocode-based

docstrings or structured specifications using formats such as Gherkin, which would reduce ambiguity and improve consistency. In addition, advanced evaluation approaches that combine static analysis, dynamic testing, and semantic checks are needed to provide a more comprehensive and reliable assessment of functional correctness in LLM-generated code.

5.3 Static Analysis Capabilities

Our framework uses SAST tools to detect errors and security issues. However, these tools cannot always take the full code context into account, which can lead to both false positives and false negatives during analysis [9, 10]. As a result, it is generally observed in practice that static analysis tools can trigger unnecessary code modifications and still miss certain vulnerabilities due to false positives and false negatives.

Prior research [43] has shown that combining multiple SAST tools can improve detection coverage and accuracy, further reducing false positives and false negatives. Therefore, we can consider incorporating a broader range of SAST tools in the future.

5.4 Dataset Representativeness

The effectiveness of fine-tuning depends on the diversity and coverage of the training dataset. While LLMSecEval and SecurityEval provide a broad selection of security-related programming tasks, they do not encompass all real-world or domain-specific scenarios. Compared with existing studies [3, 33], the datasets used for both training and testing in our study are relatively smaller in scale. Nevertheless, the results demonstrate the feasibility of our approach.

To ensure that such findings can be generalized more broadly, further work with larger and more diverse datasets will be necessary. Moreover, there is currently a lack of sufficiently comprehensive datasets dedicated to these purposes. Therefore, we plan to consider building more diverse, custom datasets for future LLM training.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we introduced CodeEnhancer, a unified framework for improving the security and quality of Python code generated by LLMs. Our two-stage approach combines real-time validation with targeted fine-tuning, addressing syntax errors, security vulnerabilities, and functional correctness within a single pipeline.

In the first stage, the validation pipeline applied to the baseline GPT-4o model removed 83.8% of vulnerabilities and resolved all functional correctness issues.

In the second stage, we fine-tuned the LLM using both expert-written and framework-refined code. The framework-tuned model consistently achieved the lowest vulnerability rates among all models. For example, on the LLM-SecEval dataset, the proportion of vulnerable code was reduced from 43.6% for baseline GPT-4o-generated code to 18.4% before refinement and further to 6.7% after refinement. On SecurityEval, vulnerability rates dropped from 61.2% for GPT-4o to 35.2% before refinement and 3.5% after refinement. Similar trends were observed for functional correctness.

Importantly, the framework-tuned model consistently outperformed the expert-tuned model in Experiment #2, achieving lower vulnerability rates and higher functional correctness on both benchmark datasets. This highlights the effectiveness of automated self-refinement and suggests that high-quality, self-generated data can be more beneficial for fine-tuning than relying solely on expert-written code.

6.2 Future Work

As future work, I plan to incorporate additional static and dynamic analysis tools, adopt advanced fine-tuning strategies, and develop custom training datasets that better reflect the diversity of real-world programming

challenges. Evaluating the framework in practical software development environments is also a key next step.

As LLMs become increasingly central to software development workflows, ensuring the generation of secure and correct code is critical. Our proposed framework marks a significant step toward that objective, demonstrating that the combination of LLMs with static analysis and purpose-driven fine-tuning can substantially increase the trustworthiness and reliability of AI-assisted coding systems.

Publications

Journal Papers

[1] **Jongmin Lee**, Khang Mai, Nakul Ghate, Tomohiko Yagyu, Razvan Beuran and Yasuo Tan, "CodeEnhancer: LLM-Generated Python Code Enhancement Through SAST Integration and Fine-Tuning," *Knowledge-Based Systems*, 2025. *(under review)*

[2] Khang Mai, **Jongmin Lee**, Razvan Beuran, Ryosuke Hotchi, Sian En Ooi, Takayuki Kuroda, Yasuo Tan, "RAF-AG: Report analysis framework for attack path generation," *Computers & Security*, vol. 148, 2025, Art. no. 104125, ISSN 0167-4048. https://doi.org/10.1016/j.cose.2024.104125

Peer-Reviewed Conference Papers

[3] Khang Mai, Nakul Ghate, **Jongmin Lee**, Razvan Beuran, "LLM-based Fine-grained ABAC Policy Generation," In *Proceedings of the 11th International Conference on Information Systems Security and Privacy (ICISSP 2025)*, Porto, Portugal, February 20-22, 2025.

Non-Peer-Reviewed Conference Papers

[4] Nakul Ghate, Khang Mai, **Jongmin Lee**, Razvan Beuran, "Feasibility of integration of LLM as the role of security experts in access control models," In *The 42nd Symposium on Cryptography and Information Security (SCIS 2025)*, Kitakyushu, Japan, January 28-31, 2025.

[5] **Jongmin Lee**, Khang Mai, Nakul Ghate, Razvan Beuran, "Framework for Validating and Improving Python Code using Large Language Models," In *The 42nd Symposium on Cryptography and Information Security (SCIS 2025)*, Kitakyushu, Japan, January 28-31, 2025.

References

[1] OpenAI, "GPT-4 Technical Report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code Llama: Open foundation models for code," 2024. [Online]. Available: https://arxiv.org/abs/2308.12950

[3] M. Chen, J. Tworek, and H. J. et al., "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[4] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," *IEEE Symposium on Security and Privacy (SP)*, pp. 754–768, 2022.

[5] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2785–2799. [Online]. Available: https://doi.org/10.1145/3576915.3623157

[6] F. Tambon, A. Moradi-Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code: an empirical study," *Empirical Softw. Engg.*, vol. 30, no. 3, Feb. 2025. [Online]. Available: https://doi.org/10.1007/s10664-025-10614-4

[7] PyCQA, "Pylint," https://github.com/PyCQA/pylint, 2003.

[8] ——, "Bandit," https://github.com/PyCQA/bandit, 2014.

[9] A. Kaur and R. Nayyar, "A comparative study of static code analysis tools for vulnerability detection in C/C++ and Java

source code," *Procedia Computer Science*, vol. 171, pp. 2023–2029, 2020, 3rd International Conference on Computing and Network Communications (CoCoNet'19). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920312023

[10] J. Ruohonen, K. Hjerppe, and K. Rindell, " A Large-Scale Security-Oriented Static Analysis of Python Packages in PyPI ," in *2021 18th International Conference on Privacy, Security and Trust (PST)*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2021, pp. 1–10. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/PST52912.2021.9647791

[11] R. Kamoi, Y. Zhang, N. Zhang, J. Han, and R. Zhang, "When can LLMs actually correct their own mistakes? A critical survey of self-correction of LLMs," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 1417–1440, 2024. [Online]. Available: https://aclanthology.org/2024.tacl-1.78/

[12] C. Tony, M. Mutas, N. E. D. Ferreyra, and R. Scandariato, "LLMSecEval: A dataset of natural language prompts for security evaluations," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 588–592. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MSR59073.2023.00084

[13] M. L. Siddiq and J. C. S. Santos, "SecurityEval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques," in *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S '22)*. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 29–33.

[14] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, p. 1092–1097, Dec. 2022. [Online]. Available: http://dx.doi.org/10.1126/science.abq1158

[15] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of

ai on developer productivity: Evidence from GitHub Copilot," 2023. [Online]. Available: https://arxiv.org/abs/2302.06590

[16] P. P. Ray, "ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope," *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 121–154, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S266734522300024X

[17] C. Tony, N. E. Díaz Ferreyra, M. Mutas, S. Dhif, and R. Scandariato, "Prompting techniques for secure code generation: A systematic investigation," *ACM Trans. Softw. Eng. Methodol.*, Mar. 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3722108

[18] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 403–414.

[19] GitHub, "CodeQL," https://codeql.github.com/, 2023.

[20] H. Hajipour, K. Hassler, T. Holz, L. Schonherr, and M. Fritz, "CodeLMSec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models," in *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2024, pp. 684–709. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SaTML59370.2024.00040

[21] S.-C. Dai, J. Xu, and G. Tao, "A comprehensive study of LLM secure code generation," 2025. [Online]. Available: https://arxiv.org/abs/2503.15554

[22] SonarSource, "Sonarqube," https://docs.sonarsource.com/, 2007.

[23] Checkmarx, "Checkmarx sast," https://checkmarx.com/cxsast-source-code-scanning/, 2006.

[24] H. Hajipour, N. Yu, C.-A. Staicu, and M. Fritz, "SimSCOOD: Systematic analysis of out-of-distribution generalization in fine-tuned source code models," in *Findings of the Association for Computational Linguistics: NAACL 2024*, K. Duh, H. Gomez, and S. Bethard, Eds. Mexico City, Mexico: Association for

Computational Linguistics, Jun. 2024, pp. 1400–1416. [Online]. Available: https://aclanthology.org/2024.findings-naacl.90/

[25] M. Kharma, S. Choi, M. AlKhanafseh, and D. Mohaisen, "Security and quality in LLM-generated code: A multi-language, multi-model analysis," 2025. [Online]. Available: https://arxiv.org/abs/2502.01853

[26] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2339–2356.

[27] J. A. Prenner, H. Babii, and R. Robbes, "Can OpenAI's Codex fix bugs? An evaluation on QuixBugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, ser. APR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 69–75. [Online]. Available: https://doi.org/10.1145/3524459.3527351

[28] J. Huang, X. Chen, S. Mishra, H. S. Zheng, A. W. Yu, X. Song, and D. Zhou, "Large language models cannot self-correct reasoning yet," 2024. [Online]. Available: https://arxiv.org/abs/2310.01798

[29] K. Alrashedy, A. Aljasser, P. Tambwekar, and M. Gombolay, "Can LLMs patch security issues?" 2024. [Online]. Available: https://arxiv.org/abs/2312.00024

[30] M. Keltek, R. Hu, M. Fani Sani, and Z. Li, "LSAST: Enhancing cybersecurity through LLM-supported static application security testing," *arXiv preprint arXiv:2409.15735*, 2024. [Online]. Available: https://arxiv.org/abs/2409.15735

[31] OpenAI, "Fine-tuning guide," https://platform.openai.com/docs/guides/fine-tuning, 2025, accessed: 2025-07-09.

[32] D. Anisuzzaman, J. G. Malins, P. A. Friedman, and Z. I. Attia, "Fine-tuning large language models for specialized use cases," *Mayo Clinic Proceedings: Digital Health*, vol. 3, no. 1, p. 100184, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2949761224001147

[33] J. Li, A. Sangalay, C. Cheng, Y. Tian, and J. Yang, "Fine tuning large language model for secure code generation," in *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, ser. FORGE '24. New York, NY,

USA: Association for Computing Machinery, 2024, p. 86–90. [Online]. Available: https://doi.org/10.1145/3650105.3652299

[34] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "Exploring parameter-efficient fine-tuning techniques for code generation with large language models," *ACM Trans. Softw. Eng. Methodol.*, Jan. 2025. [Online]. Available: https://doi.org/10.1145/3714461

[35] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "CodeBLEU: A method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020.

[36] S. Zhou, D. Kang, F. Khani, A. Grover, and S. Ren, "CodeBERTScore: Evaluating code generation with pretrained models of code," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Singapore: Association for Computational Linguistics, Dec 2023, pp. 13 921–13 937.

[37] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[38] S. Fakhoury, A. Naik, G. Sakkas, S. Chakraborty, and S. K. Lahiri, "LLM-based test-driven interactive code generation: User study and empirical evaluation," *IEEE Transactions on Software Engineering*, vol. 50, no. 9, pp. 2254–2268, 2024.

[39] Y. Dong, J. Ding, X. Jiang, G. Li, Z. Li, and Z. Jin, "CodeScore: Evaluating code generation by learning code execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 3, Feb. 2025. [Online]. Available: https://doi.org/10.1145/3695991

[40] T. Y. Zhang, Z. Wang, Y. Wang, K.-H. Chan, and J. C. K. Cheung, "ICE-Score: Instructing large language models to evaluate code," in *Findings of the Association for Computational Linguistics: EACL 2024*. St. Julian's, Malta: Association for Computational Linguistics, 2024, pp. 2141–2153.

[41] A. Z. Nasrat, R. Sharma, S. K. Vadlamudi, I. Tahmid, and L. Guerrouj, "Bridging LLM-generated code and requirements: Reverse generation technique and SBC metric for developer insights," *arXiv preprint arXiv:2502.07835*, Feb 2025.

[42] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," *IEEE Symposium on Security and Privacy (SP)*, pp. 1355–1369, 2023.

[43] A. Nguyen-Duc, M. V. Do, Q. Luong Hong, K. Nguyen Khac, and A. Nguyen Quang, "On the adoption of static analysis for software security assessment–a case study of an open-source e-government project," *Computers & Security*, vol. 111, p. 102470, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404821002947