Journal of Cybersecurity and Privacy

Tracked for Impact Factor

CITESCORE 9.1

Article

Leveraging Static Analysis for Feedback-Driven Security Patching in LLM-Generated Code

Kamel Alrashedy, Abdullah Aljasser, Pradyumna Tambwekar and Matthew Gombolay

MDPI

https://doi.org/10.3390/jcp5040110

Journal of *Cybersecurity and Privacy*

MDPI

Article

Leveraging Static Analysis for Feedback-Driven Security Patching in LLM-Generated Code

Kamel Alrashedy *, Abdullah Aljasser, Pradyumna Tambwekar and Matthew Gombolay

School of Interactive Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA; aaljasser3@gatech.edu (A.A.); ptambwekar3@gatech.edu (P.T.); matthew.gombolay@gatech.edu (M.G.)
* Correspondence: kalrashedy3@gatech.edu

Abstract

Large language models (LLMs) have shown remarkable potential for automatic code generation. Yet, these models share a weakness with their human counterparts: inadvertently generating code with security vulnerabilities that could allow unauthorized attackers to access sensitive data or systems. In this work, we propose Feedback-Driven Security Patching (FDSP), wherein LLMs automatically refine vulnerable generated code. The key to our approach is a unique framework that leverages automatic static code analysis to enable the LLM to create and implement potential solutions to code vulnerabilities. Further, we curate a novel benchmark, *PythonSecurityEval*, that can accelerate progress in the field of code generation by covering diverse, real-world applications, including databases, websites, and operating systems. Our proposed FDSP approach achieves the strongest improvements, reducing vulnerabilities by up to 33% when evaluated with Bandit and 12% with CodeQL and outperforming baseline refinement methods.

Keywords: large language models; secure AI code; security patching

check for updates

Academic Editor: Giorgio Giacinto

Received: 2 October 2025
Revised: 1 November 2025
Accepted: 13 November 2025
Published: 5 December 2025

Citation: Alrashedy, K.; Aljasser, A.; Tambwekar, P.; Gombolay, M. Leveraging Static Analysis for Feedback-Driven Security Patching in LLM-Generated Code. *J. Cybersecur. Priv.* **2025**, *5*, 110. https://doi.org/10.3390/jcp5040110

1. Introduction

While large language models (LLMs) such as GPT-4 [1,2] and CodeLlama [3,4] show strong performance in code generation tasks—including producing code from natural language [5], code translation [6], and code optimization [7]—they are prone to generating code with security vulnerabilities [8]. Even though LLMs have been shown to improve developer productivity in writing, explaining, and even refining buggy code [9], LLMs often lack the security-specific knowledge needed to avoid critical security vulnerabilities in code generation [10–12].

Prior work has not sufficiently addressed techniques to mitigate security vulnerabilities in LLM-generated code. For example, recent work has explored self-debugging techniques, in which LLMs generate, audit, and iteratively refine their own code [13]; however, these merely fix bugs rather than assessing the code's susceptibility to malign actors. Such attacks could include gaining unauthorized access to sensitive data or systems, e.g., through SQL injection attacks that manipulate database queries. This problem is especially critical when code interfaces with external services as LLMs struggle to detect and correct security issues due to limited understanding of those external services and best practices for secure coding [14,15]. Recent LLM-centric efforts like INDICT [16] and LLM-SecEval [17] attempt to benchmark or refine insecure outputs, but either focus on synthetic examples or require extensive multi-agent orchestration.

A seemingly straightforward approach to mitigating these security vulnerabilities is to train the models to simply recognize and refine such flaws. Yet, this naive approach would require a large, high-quality, costly and time-consuming dataset curation effort with annotations from human cybersecurity experts distinguishing secure from insecure code. Furthermore, relying on generative models alone may be insufficient. Prior work highlights the need for robust feedback mechanisms during and after training [18,19]. LLMs depend on accurate feedback to improve, and such feedback must come either from security experts or from automated static analysis tools.

In this paper, we introduce Feedback-Driven Security Patching (FDSP), closed-loop refinement process, and present a new benchmark, *PythonSecurityEval*, to address key limitations in prior work. FDSP addresses those limitations by integrating LLMs and static code analysis techniques to automatically detect vulnerabilities in LLM-generated code and provide actionable feedback to the LLM. Using the feedback, the LLM proposes a specific patch for each vulnerability. In the final step, FDSP integrates these patches into the original code to create a more secure version (see Figure 1). Unlike one-shot correction approaches, FDSP's closed-loop design repeatedly cycles between detection and refinement, achieving better results than either the static analyzer or the LLM could on its own.

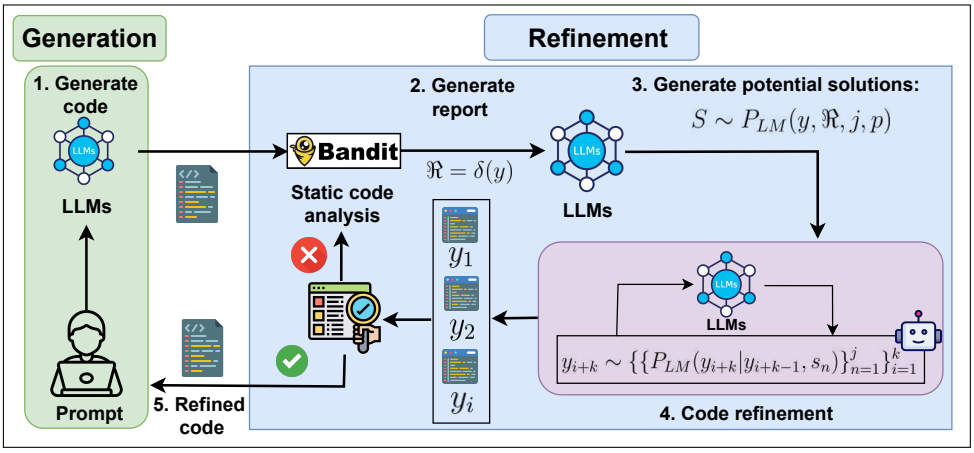

Figure 1. Overview of our closed-loop approach: Code generated by the LLM is analyzed with Bandit to detect vulnerabilities (see Figure 2). For each issue, the LLM generates multiple candidate fixes, each refined iteratively up to K times. The process repeats until the code passes static analysis or reaches the iteration limit, as illustrated by the red 'X' loop. The details are described in Section 4.

To further the science of secure code generation and to evaluate the contribution of *PythonSecurityEval*, we also introduce *PythonSecurityEval*, a novel benchmark of 470 natural language prompts derived from real-world Stack Overflow posts. Unlike datasets and benchmarks from prior work [17,20], this benchmark includes diverse and realistic application domains—such as databases (e.g., MySQL), operating systems, URLs, and web frameworks (e.g., Flask)—and targets common vulnerabilities including SQL injection, cross-site scripting (XSS), broken access control, and command injection.
Our contributions are as follows:

1. We propose FDSP, a method for improving LLM-generated code security by incorporating feedback from static analyzers such as Bandit.
2. We develop a novel benchmark, *PythonSecurityEval*, to evaluate how well language model-based approaches can produce secure code stratified by diverse and common types of security vulnerability.
3. Across three benchmarks, including *PythonSecurityEval*, using GPT-4, GPT-3.5, and CodeLlama, we find that FDSP improves patch success rates by up to 17.6% against baselines.

In the following sections, we discuss related work and present the design of FDSP. We then describe our benchmark and experimental setup, report results across multiple LLMs and static analysis tools, and discuss limitations and future directions.

2. Related Work

We review two primary areas of prior research: Language models for code and methods for refining LLM-generated code.

2.1. Language Models for Code

Code generation models have become a popular research area among Machine Learning (ML) and Software Engineering (SE) communities. The most common application of code generation models is the text-to-code generation task, wherein users prompt an LLM with natural language instructions to complete a coding task, and the LLM generates the corresponding code. Examples of the text-to-code generation include CodeLlama [3] and CodeGeeX [21]. All three achieve state-of-the-art performance on the Mostly Basic Programming Problems (MBPP) dataset [22]. The DocPrompting approach further demonstrates that prompting language models with code documentation improves code generation performance on models such as CodeT5, CodeX, and GPT-Neo on MBPP [23,24]. Beyond code generation, LLMs are also capable of code translation [6], code repair [25,26], code documentation [27], code testing [28,29] and defect prediction [30–33]. Our interest lies in exploring how these LLM capabilities can be applied to address security issues in LLM-generated code.

CyberSecEval [34] introduces a benchmark that utilizes static analysis to automatically extract insecure code examples from open-source repositories, subsequently generating prompts for evaluating large language models. In contrast, our proposed benchmark, *PythonSecurityEval*, is manually curated from Stack Overflow, ensuring that the prompts directly reflect real-world scenarios and practical challenges faced by developers. While INDICT [16] employs multi-agent systems in which LLMs autonomously interact through internal dialogue to address both security and general helpfulness, the FDSP framework adopts a different strategy. FDSP integrates deterministic static code analysis as an external feedback loop. Unlike INDICT's primarily LLM-driven feedback and broader focus, FDSP is specifically designed to address security, leveraging static analysis to guide the generation of multiple, diverse solutions for each security issue.

2.2. Refinement of LLMs

Recent studies have demonstrated that LLMs can refine their own output and adapt to feedback from external tools or human input. Self-Refine [35] prompts LLMs to generate remedial feedback to refine its output, across a variety of tasks including code-generation. A similar technique called Self-Debugging [13] enables code generation models to debug initially generated code using feedback from the same LLM, unit test results, or compiler error messages. Feedback from the LLM explains the code line by line, which is then used to refine the generated code. This approach has shown improvement in three different code generation applications. An alternate approach, called Self-Repair [36], seeks to produce feedback specifically focusing on why any faulty code snippet is incorrect. Another study [37] introduced CRITIC, which enables the model to engage with external tools such as a code interpreter, calculator, and search engine to receive feedback and improve the generated output. In our work, we build on these self-refinement methods towards enabling large language models to refine security issues in generated code.

Feedback can come from various sources, including human feedback, external tools, or the deployment environment. Human feedback is the most effective, accurate source

of feedback; however, it is also cost- and time-intensive [38,39]. Alternatively, feedback can be provided by external tools, e.g., compiler error messages [40] and Pylint, a static code analyzer, for improving Python coding standards [41]. Additionally, previous studies have proposed techniques to obtain feedback from LLMs, including the LLM-Augmenter system [42] and Recursive Reprompting and Revision framework [43–45]. Our approach combines the strength of these automated approaches by incorporating feedback from both external tools and LLMs.

3. Background

In this section, we discuss the role of LLMs in software engineering, refinement techniques, and code vulnerabilities. We summarize key concepts fundamental to our study.

3.1. LLMs in Software Engineering Applications

LLMs such as GPT are pre-trained on vast general-purpose datasets, with model sizes reaching millions or even billions of parameters. Training is typically performed using supervised learning. Some LLMs, such as CodeLlama [3], are trained exclusively on large code-specific datasets and are further fine-tuned for particular tasks, including code generation. These models are designed to comprehend user input and generate code that aligns with user requirements, aiming to meet the functional needs of developers.

LLMs generate code in a manner similar to human programmers, writing solutions to specified problems. However, just as human-written code can contain vulnerabilities and flaws, code produced by LLMs may also introduce security issues [13,46]. To enhance developer productivity and build trust in LLM-generated code, it is crucial to evaluate these models in order to find and refine bugs and security vulnerabilities. Today, LLMs have become essential tools for software engineers, supporting tasks such as code generation, automated debugging, and code documentation.

3.2. LLM Refinement

Similar to human developers, LLMs often do not produce correct or optimal code on their first attempt [35]. In practice, programmers iteratively write, test, and refine code to improve efficiency, refine bugs, or address security issues. This process frequently involves running unit tests or using static code analysis tools to generate feedback, which is then incorporated into subsequent code revisions.

LLMs also benefit from refinement processes, as their initial code outputs are not always of high quality. Empirical studies have shown that refinement—where the model revises its output based on feedback can significantly improve the quality of generated code compared to the first attempt. Refinement techniques can involve self-refinement, in which the LLM generates feedback and iteratively improves its own code, or external refinement, where feedback is provided by automated tools, the environment, or through reinforcement learning approaches. The core concept behind code refinement is to supply feedback to the LLM, enabling it to iteratively enhance its initial output.

3.3. Code Vulnerabilities

Code vulnerabilities are security weaknesses in software implementations that can be exploited by attackers to compromise the integrity, confidentiality, or availability of a system. These vulnerabilities often arise from a range of issues, including improper input validation, insecure API usage, insufficient authentication, or unsafe handling of sensitive data. The Common Weakness Enumeration (CWE) framework provides a standardized classification of software vulnerabilities, covering categories such as injection flaws (e.g., SQL injection, cross-site scripting), buffer overflows, and authentication bypasses.

Traditional vulnerability detection methods rely on static analysis tools that examine code without executing it, identifying potential security issues and generating reports for developers. However, these tools are limited in that they only detect vulnerabilities—they do not refine or automatically refine the code. Recent research has begun to explore the use of LLMs for vulnerability detection, but this area is still in its early stages and faces significant challenges, such as the availability and quality of training data. Moreover, even experienced developers find many types of vulnerabilities difficult to detect and address. In the context of LLM-based code generation, the risk of introducing security vulnerabilities remains a significant concern [13].

4. Our Approach

LLMs often generate code containing security vulnerabilities that static analysis tools can detect but not automatically fix. To address this challenge, we introduce FDSP a closed-loop, iterative refinement process designed to automatically refine vulnerable code using LLMs. The key insight is that, although LLMs may initially produce vulnerable code, LLMs can generate effective security solutions when given explicit feedback about vulnerabilities. Formally, FDSP operates in the following four phases:

1. Code generation : An LLM generates candidate code for a given task.
2. Code testing: The code is passed through a static analyzer to identify security issues and produce structured feedback reports.
3. Multi-solution generation: Using this feedback, the LLM generates multiple candidate patches aimed at refining the detected vulnerabilities.
4. Iterative refinement: Each candidate solution is fed back into the LLM along with the vulnerable code to further refine and resolve any remaining vulnerabilities.

Crucially, phases 2 through 4 form a closed feedback loop and this cycle continues until the vulnerabilities are resolved or a maximum number of iterations is reached. This loop is formalized in Algorithm 1 and illustrated with an example in Figure 2.

FDSP's multi-solution, multi-iteration design overcomes two core limitations of single-shot patching strategies:

- Solution diversity: Different security vulnerabilities may require different fix strategies. Generating multiple solutions increases coverage of potential remediation approaches.
- Implementation reliability: LLMs may not perfectly implement a solution on the first attempt. Multiple iterations allow for refinement and correction of partial implementations.

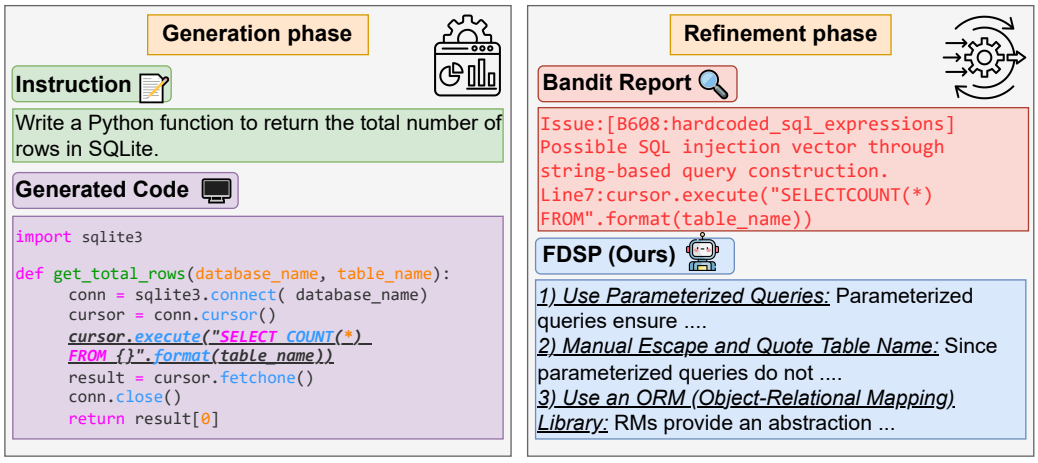

Figure 2. An example prompt from the *PythonSecurityEval* dataset where GPT-4 generates SQL injection-vulnerable code. Bandit, a static code analysis tool, then detects the vulnerability and generates a report. Finally, FDSP produces potential solutions.

Algorithm 1 Feedback-Driven Security Patching (FDSP) algorithm

Require: Input x, LLMs P_{LM}, number of potential solutions J, number of iterations K
Ensure: Refine vulnerable code y from the LLMs $P_{LM}(y_i|x)$

```
# Generate code (Equations (1) and (2))
Initialize output y_i from P_LM(x)
S ~ P_LM(y, ℜ, j, p)
# Generate potential solutions (Equation (4))
for s ∈ S do
    # Iteration for each potential solution (Equation (5))
    for k ← 1 to K do
        y_i ← P_LM(y, s)
        if δ(y_i) is secure then
            # Stop condition
            Return y_i
        end if
    end for
end for
Return y
```

4.1. Code Generation

Given a natural language description of a Python function denoted as x, an LLM generates a Python program, y, according to $P_{LM}(y|x)$ (Equation (1)). Next, the program, y, is executed. If there is a compiler error message, which we denote by $\{e_c\}$. We then send both the original program y and its compiler error message $\{e_c\}$ to the LLM so it can attempt to refine the error, as described in Equation (2).

$$y_i \sim P_{LM}(y_i|x) \tag{1}$$

$$y^c \sim P_{LM}(y^c|x, y_i, e_c) \tag{2}$$

In our work, we focus on a zero-shot setting as, in real-world use cases, users prompt LLMs to generate code directly without providing examples.

4.2. Code Testing

Static code analysis tools are utilized by software engineers to evaluate the quality of the code and identify any potential vulnerabilities. We use Bandit, https://github.com/PyCQA/bandit (10 November 2023), a static code analysis tool designed to detect common vulnerabilities in Python functions. Bandit constructs the Abstract Syntax Tree (AST) for a Python function and conducts analysis on the AST nodes. Subsequently, the Bandit, denoted by δ, generates a report $\Re$ about the code y (see Figure 2). We then pass the report $\Re$ from Bandit to the LLMs to generate potential solutions to refine the vulnerabilities. We can describe the Bandit report as shown in Equation (3).

$$\Re = \delta(y) \tag{3}$$

4.3. Multi-Solution Generation

A central innovation in FDSP is its ability to leverage structured vulnerability reports to generate multiple diverse candidate fixes instead of relying on a single correction attempt. Given a report, $\Re$, detailing the identified vulnerabilities, the LLM is prompted to produce, J, distinct remediation strategies, each offering a different approach to addressing the issues. This approach systematically explores a broader solution space, increasing the likelihood that at least one candidate will fully eliminate the vulnerability.

$$S \sim P_{LM}(y, \Re, j, p) \tag{4}$$

Equation (4) defines the generation step, where $S = \{s_1, s_2, \ldots, s_J\}$ denotes the set of solutions, j, indexes the strategy, and, p, is the instruction prompt guiding the LLM toward refinement. Each solution, s_i, typically consists of two components: (i) a description of the identified vulnerability, and (ii) a proposed fix method (e.g., input sanitization or parameterized queries).

For example, when Bandit detects SQL injection vulnerability, the LLM might generate solutions involving parameterized queries, input validation, and ORM usage (as shown in Figure 3). This diversity increases the likelihood that at least one solution will successfully address the vulnerability.

An example of generated solutions

(1) Use Parameterized Queries: Parameterized queries ensure that user input is treated as a literal value rather than executable code. Most database libraries provide a way to create these queries, also known as prepared statements.
(2) Manual Escape and Quote Table Names: Since parameterized queries do not support table or column names, you can manually ensure that table names are valid, using a whitelist approach where only approved table names are used.
(3) Use an ORM (Object-Relational Mapping) Library: ORMs provide an abstraction over SQL by allowing you to interact with the database using objects. Libraries such as SQLAlchemy for Python handle escaping and quoting internally in a secure manner.

Figure 3. An example of a solution generated for the security issues in Figure 2.

4.4. Iterative Refinement

We generate a set of diverse candidate solutions, $S = \{s_1, s_2, \ldots, s_n\}$. Each solution $s_i \in S$ is refined independently for up to K iterations, producing at most $n \times K$ refined candidates. This structure combines breadth (diverse initial solutions) with depth (iterative refinement of each solution).

Formally, the refinement process can be expressed as

$$y_{i+k} \sim \{\{P_{LM}(y_{i+k}|y_{i+k-1}, s_n)\}_{n=1}^{j}\}_{i=1}^{k} \tag{5}$$

where y_i denotes the candidate at iteration, i, and P_{LM} is the probability distribution induced by the language model. Equation (5) captures the iterative update applied to each candidate solution.

The refinement process terminates early for a candidate solution if Bandit analysis indicates that the code is secure, ($\delta(y_i)$, returns no vulnerabilities), or once the maximum iteration limit, K, is reached. This per-solution loop ensures that vulnerabilities unresolved in earlier attempts can still be addressed in later iterations. Table 1 summarizes the key FDSP algorithm parameters and their configuration values.

Table 1. FDSP algorithm parameters and their configuration values.

Parameter	Value	Description
J	3	Number of diverse solution strategies generated for each detected vulnerability
K	3	Maximum refinement iterations applied per solution (terminates early upon successful vulnerability remediation)

5. Experimental Settings

In this section, we discuss the experimental setup used to evaluate the effectiveness of our proposed approach, FDSP.

5.1. Benchmarks

Existing benchmarks, LLMSecEval and SecurityEval, are insufficient for large-scale evaluation due to their limited size and diversity (see Table 2). To address this limitation, we introduce *PythonSecurityEval*, comprising 470 natural language prompts for diverse real-world applications, collected from Stack Overflow. We utilize *PythonSecurityEval* to compare FDSP with existing strategies for refining security issues.

- LLMSecEval: This dataset contains natural language prompts to evaluate LLMs on generating secure source code [17]. LLMSecEval is comprised of 150 total prompts (natural language descriptions of code), covering the majority of the top 25 Common Weakness Enumeration (CWE).
- SecurityEval: This dataset can evaluate LLMs on their ability to generate secure Python 3 programs [20]. SecurityEval comprises 121 natural language prompts covering 75 types of vulnerabilities. Each prompt includes the header of a Python function along with comments describing each function.
- PythonSecurityEval (Ours): We collected a new benchmark from Stack Overflow to address the limitations of the existing datasets. Current datasets are limited in size and diversity and are therefore insufficient for evaluating the ability of LLMs to generate secure code adequately addressing security vulnerabilities. *PythonSecurityEval* includes natural language prompts intended to generate Python functions that cover diverse real-world applications. This dataset consisting of 470 prompts is three times larger than those used in LLMSecEval and SecurityEval.

PythonSecurityEval is a diverse and extensive benchmark, covering the majority of real-world applications that consider the primary sources of common vulnerabilities. For example, SQL injection occurs when Python code connects to, inserts into, and queries from a SQL database. There are several examples in our benchmark where the prompt involves writing Python code to insert a value into an SQL database. Another example is command injection, where Python code interacts directly with the operating system, allowing attackers to gain unauthorized access. We include examples that generate Python code with access to the operating system and diagnose how LLMs generate the code without any issues or the ability to refine the code. Lastly, cross-site scripting (XSS) is a common type of security vulnerability that occurs in web applications. We include prompts that generate Python code for Flask, which is a Python web framework for creating websites (see Table 2).

Table 2. Domain diversity statistics across benchmarks. Some functions appear in multiple domains. We determined the type of domain for the function by identifying calls to domain-specific libraries (see Table 3).

Domain	PythonSecurityEval (Ours)	LLMSecEval	SecurityEval
Computation	168 (35.7%)	44 (29.5%)	32 (26.4%)
System	313 (66.6%)	94 (63.1%)	68 (56.2%)
Network	147 (31.3%)	63 (42.3%)	29 (24.0%)
Cryptography	29 (6.2%)	8 (5.4%)	16 (13.2%)
General	414 (88.1%)	128 (85.9%)	118 (97.5%)
Database	114 (24.3%)	23 (15.4%)	6 (5.0%)
Web Frameworks	43 (9.1%)	46 (30.9%)	8 (6.6%)
Total	470	150	121

Table 3. To determine the domain category of each function in Table 2, we analyzed whether it invoked domain-specific libraries [47].

Domain	Library
Computation	os, pandas, numpy, sklearn, scipy, math, nltk, statistics, cv2, statsmodels, tensorflow, sympy, textblob, skimage
System	os, json, csv, shutil, glob, subprocess, pathlib, io, zipfile, sys, logging, pickle, struct, psutil
Network	requests, urllib, bs4, socket, django, flask, ipaddress, smtplib, http, flask_mail, cgi, ssl, email, mechanize, url
Cryptography	hashlib, base64, binascii, codecs, rsa, cryptography, hmac, blake3, secrets, Crypto
General	random, re, collections, itertools, string, operator, heapq, ast, functools, regex, bisect, inspect, unicodedata
Database	sqlite3, mysql, psycopg2, sqlalchemy, pymongo, sql
Web Frameworks	Django, Flask, FastAPI, Tornado, Pyramid, Bottle

5.2. Baselines

We consider four baseline refinement approaches, which are as follows:

I. Direct prompting: This approach involves sending generated code back to an LLM with the instruction: *Does the provided function have a security issue? If yes, please refine the issue.* If LLMs detect any security issues in the code, they will refine the issue and generate secure code.
II. Self-Debugging: The initial step in self-debugging is for LLMs to generate the code. Subsequently, the generated code is sent back to the same LLMs to generate feedback. Finally, both the generated code and the feedback are fed back to the LLM to correct any existing bugs.
III. Bandit feedback: We develop this baseline that utilizes Bandit to produce a report if there are any security issues in the code, as shown in Figure 2. We use this report as feedback to enable the LLM to refine the vulnerable code. This strategy is similar to prior approaches wherein external tools provide feedback to the LLM to refine its outputs [48–50]. Bandit feedback does not provide a solution to refine the issue; it simply highlights the problematic line and type of issue.
IV. Verbalization: We verbalize the feedback from Bandit, via an LLM, to produce intelligible and actionable feedback to resolve security issues. The verbalized feedback provides a detailed explanation in natural language of the specialized output from Bandit This expanded explanation offers deeper insights into the security issues and may suggest solutions to address the vulnerabilities.

5.3. Evaluation Metrics

This paper evaluates the ability of LLMs to generate vulnerable code and subsequently correct security issues identified in the code. The primary evaluation metric is the *Vulnerability Rate*, which measures the proportion of code samples identified as insecure by the static analysis tools, as shown in Equation (6).

$$\text{Vulnerability Rate} = \frac{N_{\text{vuln}}}{N_{\text{total}}} \tag{6}$$

In Equation (6), N_{total} represents the total number of generated code samples, and N_{vuln} corresponds to the subset of those samples that contain at least one vulnerability detected by static analysis tools. This metric provides a quantitative measure of

the baseline security of generated code prior to applying refinement techniques. To verify whether the generated code contains vulnerabilities, we utilize two static analysis tools:

I Bandit: Bandit is an open-source static analysis tool developed by the OpenStack Security Project to identify security vulnerabilities in Python source code. Its core mechanism involves parsing the Abstract Syntax Tree (AST) of Python programs and systematically inspecting it to detect known security anti-patterns and vulnerable code constructs. Bandit's rule set covers a broad range of common security issues in Python, including hardcoded credentials, weak cryptographic algorithms, and unsafe subprocess management. The tool automatically generates reports highlighting potential vulnerabilities, their severity, and their precise locations within the code. In this study, we leverage Bandit both as a vulnerability detection tool for providing external feedback to large language models (LLMs) during code refinement and as an evaluation metric for measuring the effectiveness of our approach.

II CodeQL: is an open-source static analysis framework developed by GitHub (Version 2.23.1) for detecting vulnerabilities and code patterns in source code. The CodeQL workflow begins by parsing the source code into a database representation that captures its syntax, structure, and semantics, including abstract syntax trees (ASTs), control flow graphs, and data flow information. Custom queries can then be executed against this database to identify specific issues, such as insecure API usage or potential SQL injection vulnerabilities. CodeQL supports multiple programming languages, including Python. In this study, we use CodeQL as an external evaluation metric to assess the security of generated code and to evaluate how effectively refinement techniques mitigate identified vulnerabilities [51].

We selected Bandit as both a feedback resource during refinement and as an evaluation tool, while CodeQL is employed for external evaluation. Both tools are widely adopted and highly regarded within the security community. The combination of Bandit and CodeQL enables a more comprehensive and rigorous assessment of code security. Bandit is particularly effective at detecting common Python-specific vulnerabilities, while CodeQL represents a state-of-the-art approach to code security analysis, ensuring the robustness and thoroughness of our evaluation.

5.4. Models

We conduct our experiments on code generation and refinement using three state-of-the-art large language models (LLMs) described below:

I GPT-4: GPT-4 is a Generative Pre-trained Transformer model developed by OpenAI. Trained on massive text corpora using unsupervised learning, GPT-4 leverages the Transformer architecture to excel in a wide range of language tasks, including code generation, summarization, translation, and bug refining. Notably, GPT-4 is a closed-source model.

II GPT-3.5: GPT-3.5 is also part of the GPT family developed by OpenAI. With 175 billion parameters, it was trained on a general-purpose dataset. Among the various GPT-3.5 versions, we utilize "gpt-3.5-turbo-instruct", which is specifically instruction-tuned to follow user prompts and generate responses aligned with user intent.

III CodeLlama: CodeLlama is an advanced, open-source LLM developed by Meta AI, trained primarily on code datasets. It is available in three model sizes—7B, 13B, and 34B parameters. In this study, we employ CodeLlama-Instruct-34B, an instruction-tuned variant optimized for understanding and following user instructions, making it well-suited for both code generation and refinement tasks.

These models were chosen for their strong performance on a variety of code-related benchmarks and their complementary characteristics—including both closed-source and open-source paradigms, general-purpose and code-focused training, and a range of model sizes. In our experiments, each model was evaluated in a zero-shot setting, reflecting real-world scenarios where developers prompt LLMs directly without demonstration examples. This diversity enables a comprehensive evaluation of LLM capabilities in automated code security refinement.

5.5. Research Questions

This paper explores four research questions, regarding the capacity of LLMs in detecting and refining vulnerable code.

RQ1. What is the fundamental capability of LLMs in refining security vulnerabilities? This question aims to determine how effectively LLMs can inherently correct insecure code and highlight their limitations without incorporating external feedback.

RQ2. How does Bandit feedback affect the ability of LLMs to refine code vulnerabilities? This question examines how effectively the LLMs incorporate feedback provided by provided by Bandit, a static code analysis tool.

RQ3. How does FDSP improve LLM performance in refining code vulnerabilities? This question aims to assess how well the LLMs generate multiple potential solutions and iterate over each one to refine vulnerabilities.

RQ4. How important are the multiple generated solutions and iterations of FDSP? We conduct ablation studies to isolate these factors by restricting FDSP to a single solution or iteration. This analysis reveals whether the diversity of generated solutions and iterative refinement contribute to FDSP effectiveness.

6. Experimental Results

In this section, we present empirical results addressing each research question, with detailed findings summarized in Table 4. We also include an ablation study to assess the specific contributions of our approach, along with a qualitative analysis to further contextualize the results.

6.1. RQ1: LLMs Are Somewhat Effective at Refining Vulnerable Code on Their Own

Across the three datasets, Bandit detects between 28% and 46% of the generated code as vulnerable, while CodeQL identifies between 9.1% and 17%. In Bandit's evaluation, direct prompting and self-debugging result in modest vulnerability reductions, with improvements of less than 10% for GPT-3.5 and CodeLlama, and around 15% for GPT-4. Direct prompting and self-debugging enable LLMs to refine their generated code without feedback from external tools. Our results indicate that LLMs can intrinsically generate feedback to refine their vulnerable code, though the improvement is limited. CodeQL detects fewer vulnerabilities than Bandit, but results indicate that direct prompting and self-debugging refine approximately 50% of the vulnerabilities.

Finding #1: LLMs can modestly reduce code vulnerabilities through direct prompting and self-debugging, but their improvements are limited without external feedback.

6.2. RQ2: Bandit-Based Feedback Is Beneficial Towards Correcting Security Vulnerabilities in Generated Code

Integrating Bandit feedback into LLMs enhances their ability to address security vulnerabilities, as evidenced by notable improvements in Bandit evaluations and modest gains

in CodeQL evaluations. In contrast, approaches that exclude Bandit's feedback are less effective. While simple strategies like direct prompting and self-debugging can address basic security issues, they are generally insufficient for more complex vulnerabilities. As shown in Table 4, methods utilizing Bandit feedback consistently outperform simpler techniques, improving accuracy across all models and datasets. Specifically, LLMs incorporating Bandit feedback provides approximately a 30% improvement for GPT-4 and up to a 24% improvement for GPT-3.5 and CodeLlama based on Bandit evaluations. Additionally, verbalizing Bandit's feedback yields a slight increase of 1% to 2% in both evaluation metrics.

Table 4. The table presents our results utilizing Bandit and CodeQL across various datasets and approaches. The percentage of vulnerable code is reported for both the generation and refinement phases. The Generated code row indicates the percentage of vulnerable code out of the entire dataset. We report the percentage of remaining vulnerable code for each refinement approach relative to the vulnerabilities present in the initially generated code. The arrows ($\downarrow x$) indicate the absolute reduction in percentage points compared to the Generated code baseline.

Dataset	Models	GPT 4		GPT 3.5		CodeLlama	
	Evaluation Metrics	**Bandit**	**CodeQL**	**Bandit**	**CodeQL**	**Bandit**	**CodeQL**
LLM Sec Eval	Generated code	38.2%	10.1%	34.2%	18.1%	28.6%	20.7%
	Direct prompting	35.3% (↓ 2.6)	**4.7**% (↓ 5.4)	28.0% (↓ 6.0)	7.4% (↓ 10.7)	24.0% (↓ 4.6)	11.6% (↓ 9.1)
	Self-debugging	24.0% (↓ 14.0)	7.4% (↓ 2.7)	28.0% (↓ 6.0)	8.7% (↓ 9.4)	24.6% (↓ 4.0)	15.7% (↓ 5.0)
	Bandit feedback	8.0% (↓ 30.0)	5.4% (↓ 4.7)	18.6% (↓ 15.3)	8.7% (↓ 9.4)	18.0% (↓ 10.6)	13.2% (↓ 7.5)
	Verbalization	7.3% (↓ 30.6)	5.4% (↓ 4.7)	18.0% (↓ 16.0)	**6.7%** (↓ 11.4)	16.6% (↓ 12.0)	10.7% (↓ 10.0)
	FDSP (Ours)	**6.0%** (↓ 32.0)	6.7% (↓ 3.4)	**12.6%** (↓ 21.3)	8.1% (↓ 10)	**14.6%** (↓ 14.0)	**9.1%** (↓ 11.6)
Security Eval	Generated code	34.7%	12.4%	38.0%	9.1%	46.2%	32.2%
	Direct prompting	21.4% (↓ 13.2)	**5.8%** (↓ 6.6)	25.6% (↓ 12.4)	8.3% (↓ 0.8)	38.0% (↓ 8.2)	14.1% (↓ 18.1)
	Self-debugging	16.5% (↓ 18.1)	9.1% (↓ 3.3)	27.2% (↓ 10.7)	9.1% (↓ 0.0)	38.8% (↓ 7.4)	17.4% (↓ 14.8)
	Bandit feedback	**4.1%** (↓ 30.5)	6.6% (↓ 5.8)	13.2% (↓ 24.7)	5.8% (↓ 3.3)	21.4% (↓ 24.7)	13.4% (↓ 18.8)
	Verbalization	4.9% (↓ 29.7)	6.6% (↓ 5.8)	13.2% (↓ 24.7)	5.8% (↓ 3.3)	17.3% (↓ 28.92)	13.4% (↓ 18.8)
	FDSP (Ours)	**4.1%** (↓ 30.5)	8.3% (↓ 4.1)	**5.7%** (↓ 32.2)	**2.5%** (↓ 6.6)	**8.2%** (↓ 38.0)	**12.1** (↓ 20.1)
Python Security Eval	Generated code	40.2%	17.9%	48.5%	13.2%	42.3%	13.2%
	Direct prompting	25.1% (↓ 15.1)	9.6% (↓ 8.3)	42.5% (↓ 5.9)	8.5% (↓ 7.2)	31.0% (↓ 11.3)	6.6% (↓ 6.6)
	Self-debugging	24.8% (↓ 15.3)	8.7% (↓ 9.2)	43.4% (↓ 5.1)	8.9% (↓ 7.0)	33.1% (↓ 9.2)	7.9% (↓ 5.3)
	Bandit feedback	9.3% (↓ 30.8)	9.1% (↓ 8.8)	26.3% (↓ 22.1)	6.4% (↓ 11.5)	20.0% (↓ 22.3)	6.2% (↓ 7.0)
	Verbalization	8.7% (↓ 31.4)	8.5% (↓ 9.4)	23.6% (↓ 24.8)	7.4% (↓ 10.2)	19.5% (↓ 22.8)	6.0% (↓ 7.2)
	FDSP (Ours)	**7.4%** (↓ 32.7)	**7.7%** (↓ 10.2)	**15.7%** (↓ 32.7)	**5.7%** (↓ 11.7)	**8.7%** (↓ 33.6)	**5.7%** (↓ 7.5)

Finding #2: Bandit feedback significantly boosts LLMs' ability to refine security vulnerabilities, leading to much higher improvements than simple prompting or self-debugging alone.

6.3. RQ3: FDSP Shows Consistent Improvement over the Baseline

FDSP boosts the LLM ability to generate potential solutions based on feedback provided by Bandit. Our FDSP approach enhances the performance of GPT-3.5 and CodaLlama, exceeding the results achieved by either directly incorporating Bandit's feedback or verbalizing it. For *PythonSecurityEval*, FDSP shows consistent improvement over the verbalization approach, with improvements for GPT-4 (from 8.7% to 7.4%), GPT-3.5 (from 23.6% to 15.7%), and CodeLlama (from 21% to 13.6%) for Bandit evaluation. These results support that LLMs can generate potential solutions that effectively address security issues when they are supplied with feedback from static code analysis, and outperforming self-refinement or merely passing the feedback from static code analysis directly.

We evaluate the effectiveness of each method in addressing the most common security issues in CodeLlama (see Figure 4). These results suggest that neither self-refinement

nor directly passing the feedback from Bandit proves useful for CodeLlama; however, verbalization and FDSP perform well for CodeLlama.

In Code Snippet 1, we present an example of code generated by GPT-4 that contains an SQL injection vulnerability, one of the most common types of security flaws produced by LLMs. The vulnerability appears on line 7, where the `username` variable is directly interpolated into the SQL statement. This allows an attacker to craft input that could manipulate the query, leading to unauthorized data access or potential damage to the database. The FDSP approach addresses this vulnerability, as shown on line 24.

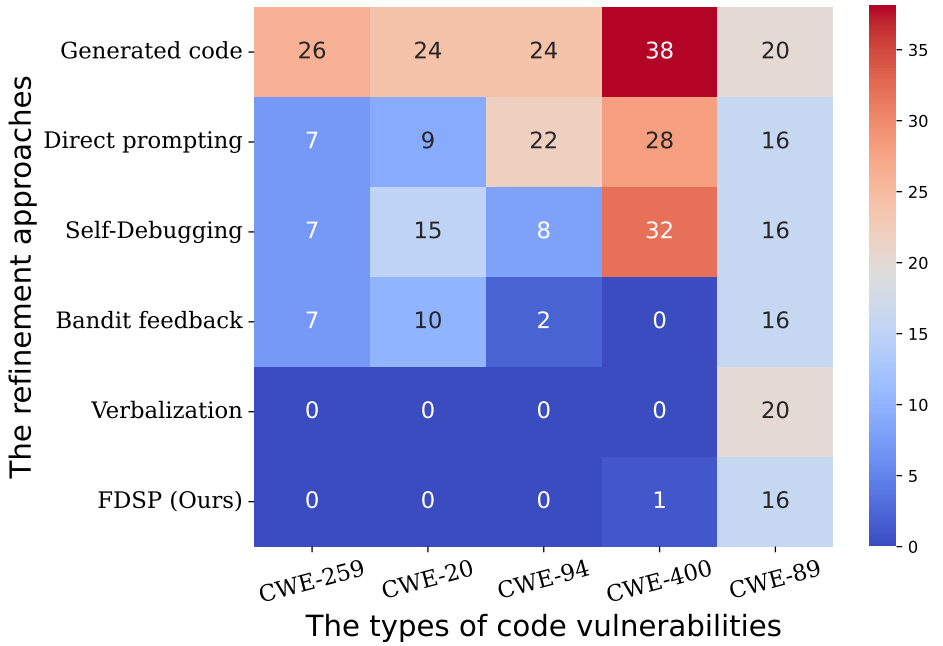

Figure 4. Vulnerability count of the five most frequent issues across approaches using CodeLlama in *PythonSecurityEval*.

Table 4 demonstrates that FDSP outperforms all baselines across all datasets and LLMs in refining vulnerable code. We attribute this improvement to two key factors: (1) FDSP's ability to generate multiple, diverse solutions for each vulnerability, and (2) its iterative feedback loop, which allows the LLMs to refine the code over several iterations. In contrast, basic approaches such as direct prompting and self-debugging rely on a single round of feedback and generate only one solution, which often fails to fully address the vulnerability. While the Bandit feedback approach highlights the vulnerable line and the type of issue, it does not provide actionable solutions to the LLM.

We evaluated only the originally vulnerable cases using Wilson confidence intervals. Direct prompting (62.4%, CI: 55.3–69.0) and self-debugging (60.8%, CI: 53.7–67.5) showed overlapping intervals, indicating no significant difference. Bandit feedback (21.7%, CI: 16.4–28.1) and verbalization (21.2%, CI: 15.9–27.5) achieved statistically significant reductions relative to these baselines. FDSP obtained the lowest residual vulnerability rate at 16.9% (CI: 12.3–22.9), with non-overlapping intervals that establish its improvements as both substantial and statistically validated.

Finding #3: FDSP consistently outperforms baseline methods by leveraging Bandit feedback, enabling LLMs to generate better security refinements than self-refinement or direct feedback alone.

```python
# Generated code containing a vulnerability.
def show_user(username):
    connection = get_connection()
    cursor = connection.cursor()

    # building and executing SQL select query
    select_query = f"SELECT * FROM users WHERE username = '{username}'"
    cursor.execute(select_query)

    # getting all rows from the executed query
    user_info = cursor.fetchall()

    # close the cursor and connection
    cursor.close()
    connection.close()
    return~user_info

# Refined the generated code using FDSP.
def show_user(username):
    connection = get_connection()
    cursor = connection.cursor()

    # building and executing SQL select query using parametrization
    select_query = "SELECT * FROM users WHERE username=%s"
    cursor.execute(select_query, (username,))

    # getting all rows from the executed SQL query
    user_info = cursor.fetchall()

    # close the cursor and connection
    cursor.close()
    connection.close()
    return~user_info
```

Code Snippet 1: An example of LLM-generated code containing a vulnerability (**top**), and the corresponding code refined and fixed by FDSP (**bottom**).

6.4. RQ4: Ablation Study

The generation of multiple solutions and repeated iterations play a critical role in FDSP (see Section 4).

To quantify the impact of these two factors, we evaluate FDSP with two ablation studies: (i) FDSP with a single solution, wherein the LLM generates only one solution instead of multiple, and (ii) FDSP with a single iteration, wherein the LLM attempts to address the vulnerable code with a single generated solutions rather than making multiple attempts. We conducted these ablation studies using GPT-4 on the *PythonSecurityEval* dataset (see Table 5). The percentage of unsolved vulnerabilities increased from 7.4% to 9.5% for single iteration and 10.0% for single solution. These results demonstrate that while multiple iterations offer some improvement, generating multiple solutions plays a more significant role in improving the generation of secure code.

Table 5. This table provides a performance comparison of FDSP and its ablated variants on *PythonSecurityEval*, using GPT-4, across multiple solutions and iterations.

Ablation Experiments	**Evaluation Metrics**	
	Bandit	**CodeQL**
Generated code	40.2%	17.9%
FDSP with single solution	10.0% (+2.6%)	8.7% (+1.0%)
FDSP with single iteration	9.5% (+2.1%)	7.9% (+0.2%)
FDSP	**7.4%**	**7.7%**

Finding #4: Generating multiple solutions and performing repeated iterations are both important for FDSP, but generating multiple solutions has a greater impact on reducing vulnerabilities.

6.5. Qualitative Analysis

We qualitatively analyze the solutions generated by FDSP and its iterations for GPT-4. In particular, we evaluate 30 randomly selected examples of vulnerable code from *PythonSecurityEval*, comprising 23 refined and 7 unrefined cases. Our findings show that the solutions generated by FDSP provide at least one actionable refine to address security issues, with 74% of the solutions offering at least two actionable recommendations. Although FDSP consistently generates correct solutions, there are instances where the LLMs fail to refine the vulnerable code. Only 26% of the generated solutions include one general security measure, such as error handling (e.g., exceptions) or input validation.

In the seven cases where vulnerabilities were not refined, FDSP still produced valid solutions, but the LLMs did not incorporate the feedback to refine the code. Three of these failures involved SQL injection vulnerabilities, where FDSP produced valid solutions, but the LLMs failed to incorporate the feedback and refine the code. The other four failures involved high-risk library calls (e.g., subprocess, paramiko), which pose significant security risks if not used properly, potentially leading to shell injection vulnerabilities.

Finding #5: FDSP frequently generates actionable and effective refinement, but struggles with complex vulnerabilities like SQL injection and unsafe library use, where LLMs often fail to fully incorporate feedback.

6.6. What Are the Most Frequent, Unresolved Coding Vulnerabilities Produced by LLMs?

Figure 5 illustrates the most common types of code vulnerabilities generated by three LLMs on the *PythonSecurityEval* dataset, with the top two being CWE-259 (use of Hard-coded Password) and CWE-400 (uncontrolled resource consumption). Fortunately, the LLMs refine most of these types of vulnerabilities (Figure 6). Next, Figure 6 depicts the most frequent unresolved security issues, where the top two are related to injection: CWE-78 (OS Command Injection) and CWE-89 (SQL Injection) at 61.1% and 80.0%, respectively, for GPT-4. Additionally, these injection vulnerabilities are among the most frequent vulnerabilities generated by LLMs.

Finding #6: The most frequent unresolved vulnerabilities produced by LLMs are injection-related issues, especially OS Command Injection (CWE-78) and SQL Injection (CWE-89), which remain difficult for LLMs to refine.

6.7. Evaluating Functional Correctness in Code Refinement

To assess whether the refinement process preserves functional correctness in addition to improving security, we conducted a systematic evaluation over the full *PythonSecurityEval* dataset. Among the 470 GPT-4 generated programs, 189 were identified by static analysis tools as containing security issues and thus refined by FDSP. The remaining 281 programs did not trigger security warnings and were therefore left unchanged.

We evaluate functional correctness for *both* groups to ensure two properties: (1) FDSP does not break functionality when applying patches, and (2) code that appears "secure" via static analysis may still contain functional defects. For each program, we implemented unit

tests using Python's *unittest* framework. Where programs interacted with external systems (e.g., operating systems, databases, web APIs), we used mocking to isolate program logic and ensure deterministic behavior consistent with best practices.

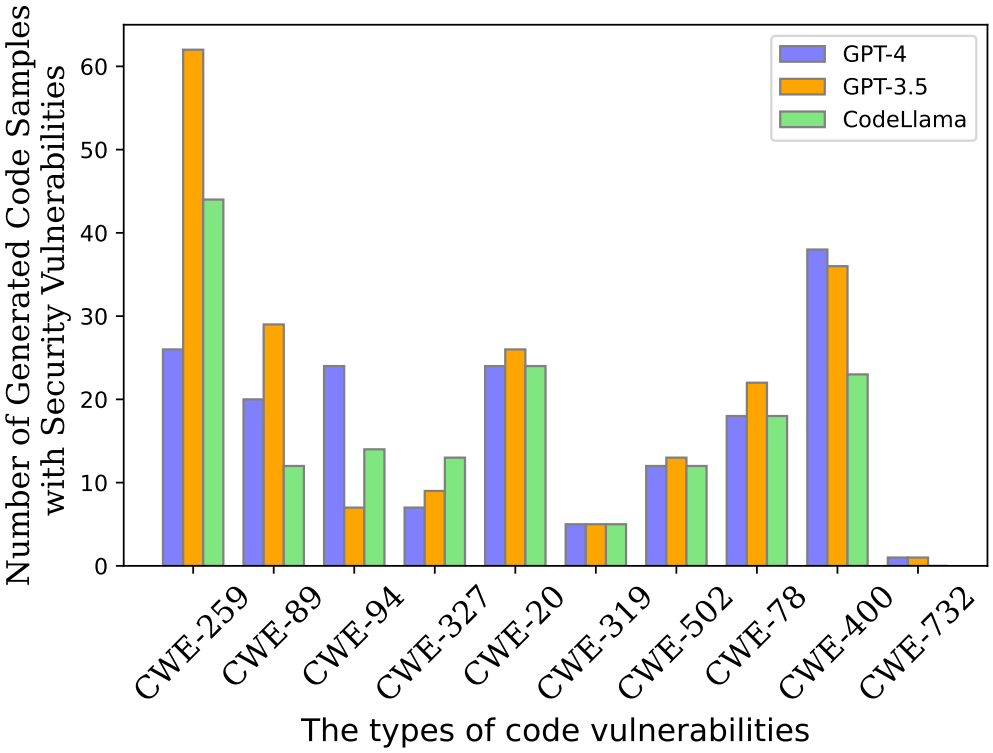

Figure 5. The total count of the most common security issues in the code generated for the *PythonSecurityEval* dataset (Top 10).

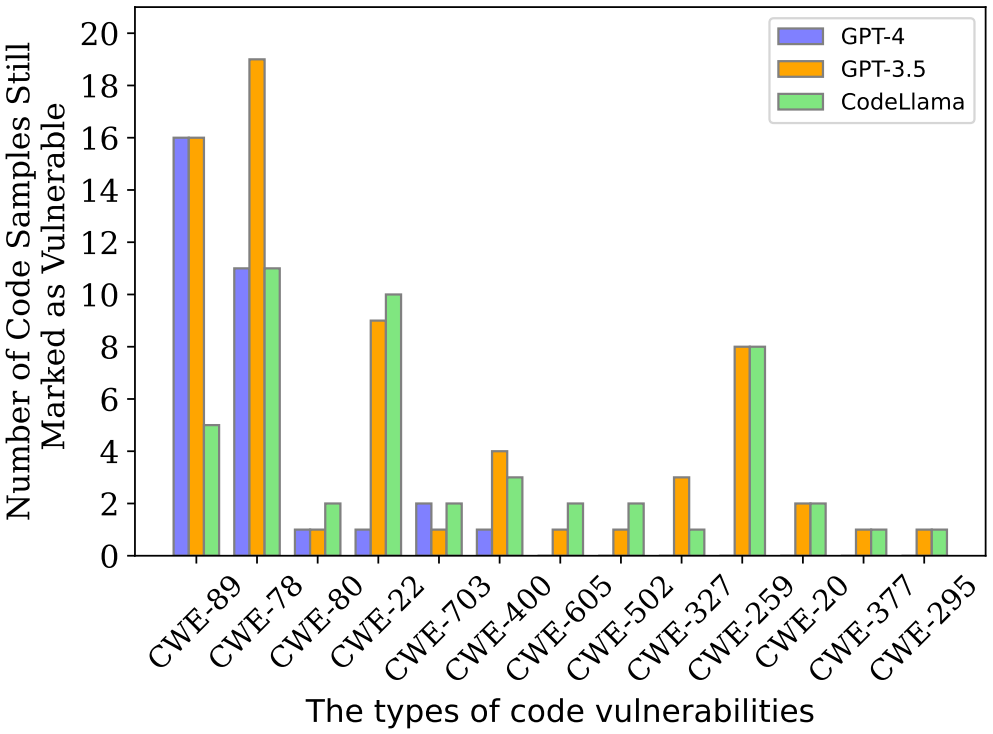

Figure 6. The total number of unresolved vulnerable code instances identified by three LLMs on the *PythonSecurityEval* dataset.

Across the 189 patched examples, 182 successfully passed all unit tests, yielding a functional correctness rate of 96.2%. The seven failing cases stemmed from edge-case logic inconsistencies introduced during patching and were resolvable with minor adjustments. For the 281 unmodified examples, 266 passed all tests, resulting in a 94.7% pass rate. This indicates that even code deemed "secure" by static analyzers may still contain functional issues—a complementary dimension to security quality.

Finding #7: FDSP preserves functional correctness in the vast majority of cases, achieving a 96% pass rate in unit tests across 50 randomly sampled GPT-4 refinements from the *PythonSecurityEval* dataset. The two failures were due to minor compiler-related issues, demonstrating that FDSP-enhanced code remains realistic, secure, and functionally reliable.

6.8. Cross-Tool Evaluation of FDSP with Semgrep

Our additional Semgrep evaluation further underscores the effectiveness of FDSP compared to baseline approaches. On our benchmark with GPT-4, the raw generated code exhibited a relatively high vulnerability rate of 12.1%. Direct prompting and self-debugging provided moderate reductions, lowering vulnerabilities to 8.0% and 5.3%, respectively, while incorporating Bandit feedback reduced this further to 2.3%. Verbalizing Bandit's feedback yielded a slightly stronger outcome at 1.9%. Most notably, FDSP achieved the lowest vulnerability rate of only 0.6%, demonstrating its clear advantage in leveraging static analysis feedback through iterative and diverse solution generation. These results align with our earlier findings and highlight FDSP's robustness in systematically refining insecure code beyond what simpler refinement strategies can achieve.

Finding #8: Evaluated with Semgrep, FDSP achieves the lowest vulnerability rate (0.63%), outperforming all baselines. This confirms that FDSP remains the most effective approach for refining insecure code.

6.9. Statistical Analysis of Refinement Effectiveness

To further evaluate the reliability of vulnerability remediation, we conducted a Wilson confidence interval analysis on *PythonSecurityEval* results generated using GPT-4. Our analysis focused exclusively on originally vulnerable cases across five remediation strategies. Direct prompting and self-debugging left 62.4% (CI: 55.3%–69.0%) and 60.8% (CI: 53.7%–67.5%) of the code samples vulnerable, respectively—both exhibiting overlapping confidence intervals that suggest no statistically significant difference in effectiveness, as shown in Figure 7. In contrast, Bandit feedback (21.7%, CI: 16.4%–28.1%) and verbalization (21.2%, CI: 15.9%–27.5%) showed lower residual vulnerability rates with non-overlapping confidence intervals compared to the weaker baselines, confirming statistically meaningful improvements. Most notably, our FDSP (Feedback-Driven Self-Patching) approach yielded the lowest residual rate at 16.9% (CI: 12.3%–22.9%). This non-overlapping interval establishes FDSP as not only significantly more effective than direct prompting and self-debugging, but also marginally but meaningfully better than Bandit feedback and verbalization. These findings underscore the statistical rigor and robust reliability of FDSP in reducing vulnerabilities in LLM-generated code on *PythonSecurityEval*.

To statistically confirm these differences across all methods, we performed a one-way ANOVA test, which revealed a significant overall effect of remediation strategy on residual vulnerability rate ($F = 30.33$, $p = 9.09 \times 10^{-24}$). This result reinforces the conclusion that

the observed performance differences are unlikely due to chance, providing further support for the robustness of FDSP.

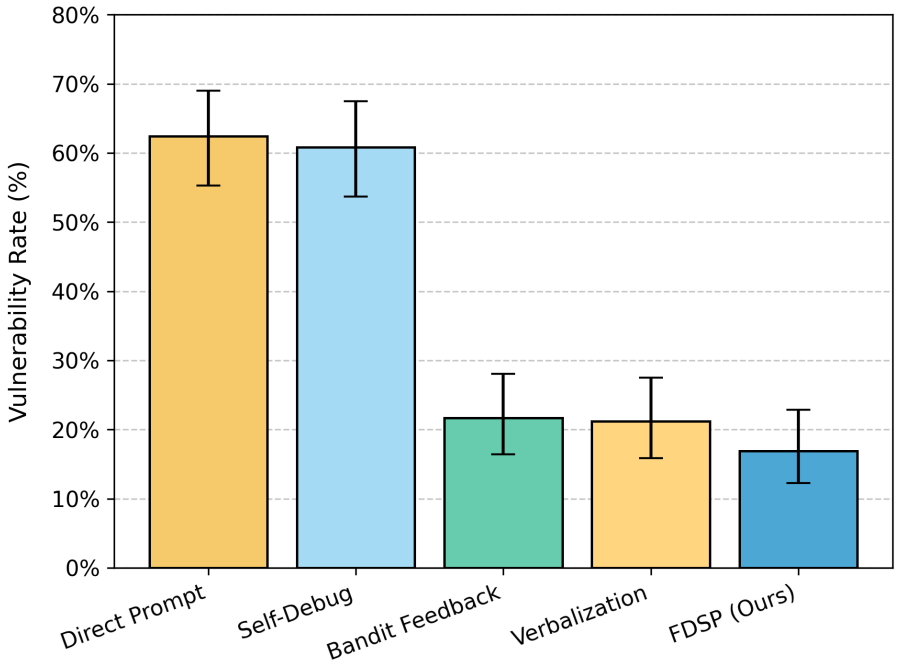

Figure 7. Residual vulnerability rates with 95% Wilson confidence intervals for five refinement methods on *PythonSecurityEval* using GPT-4.

6.10. Computational Cost

To assess the practical feasibility of FDSP, we compared the computational cost across all evaluated methods in terms of execution time and API expenditure (USD). The results, summarized in Table 6, reveal that FDSP incurs the highest computational overhead averaging 41.64 ± 25.06 s and costing USD 25.32 per full run primarily due to its multi-round feedback and multi-candidate generation loops. In contrast, Direct Prompting is the most efficient (11.27 ± 4.0 s, USD 1.05), followed by Direct Bandit Feedback (8.54 ± 3.35 s, USD 5.32). Self-Debugging and Verbalization occupy the middle ground, with moderate increases in both time and cost. While FDSP demands roughly 2–3× more resources than the next-best alternative, its substantially higher vulnerability-reduction rate justifies this expense in high-assurance settings. These results highlight an explicit trade-off between computational cost and security gain, underscoring FDSP's practicality for contexts where reliability outweighs runtime constraints.

Table 6. Computational cost comparison across methods using GPT-4. Mean and SD show average runtime per example (seconds); API Cost indicates total processing cost.

Method	Mean and SD (Second)	API Cost (USD)
Direct Prompting	11.27 (±4.00)	USD 1.05
Self-Debugging	22.29 (±6.00)	USD 12.43
Direct Bandit Feedback	8.54 (±3.35)	USD 5.32
Verbalization	13.73 (±4.57)	USD 8.06
FDSP (ours)	41.64 (±25.06)	**USD 25.32**

6.11. Comparative Performance Analysis: FDSP vs. INDICT in Multi-Round Vulnerability Reduction

The results in Table 7 illustrate a clear performance gap between FDSP and INDICT's multi-agent framework across all evaluation rounds using GPT-4. While INDICT pro-

gressively reduces vulnerability rates through iterative agent collaboration from 27.9% to 22.8% on Bandit and from 14.0% to 14.6% on CodeQL, its improvement plateaus after the third round, indicating limited gains from additional refinement cycles. In contrast, FDSP achieves substantially lower vulnerability rates of 7.4% on Bandit and 7.7% on CodeQL, representing more than a 2× improvement in overall code security. This demonstrates that FDSP's closed-loop feedback mechanism which leverages deterministic static analysis to guide iterative patching offers a more direct and reliable refinement signal than INDICT's dialogue-based multi-agent reasoning. The consistency of FDSP's results across both Bandit and CodeQL evaluations further confirms its robustness and generalizability as a framework for secure LLM-generated code refinement.

Table 7. Comparison of FDSP with INDICT's multi-agent framework across rounds using GPT-4. Lower values indicate fewer vulnerabilities, demonstrating FDSP's superior security performance.

Dataset	Round	Bandit	CodeQL
INDICT	Round 1	27.9%	14.0%
	Round 2	23.4%	11.7%
	Round 3	19.4%	12.1%
	Round 4	20.6%	15.3%
	Round 5	22.8%	14.6%
FDSP (Ours)	——	7.4%	7.7%

6.12. Static Analysis Limitations

While static analysis offers scalable and automated vulnerability detection, it cannot capture all classes of software flaws. In particular, data-dependent and runtime-specific vulnerabilities such as race conditions, logic flaws dependent on dynamic input, or memory errors triggered only at execution often evade static inspection. These gaps stem from the fact that static analyzers reason over abstract syntax trees without executing code, which limits visibility into runtime behavior. Future research could integrate hybrid static–dynamic feedback into FDSP, allowing the framework to combine compile-time signals (from tools such as Bandit and CodeQL) with runtime evidence from fuzzing, unit testing, or dynamic taint analysis. Such integration would extend FDSP's coverage to vulnerabilities that manifest only during execution and strengthen its ability to verify both security and functional correctness. Table 8 summarizes the major CWE categories in our benchmark and indicates which classes are currently detectable by static analysis and which require hybrid or dynamic approaches.

Table 8. Coverage of vulnerability types by static, dynamic, and hybrid detection methods.

CWE Category	Example CWE	Detection Method	
		Static Analysis	Dynamic/Hybrid Analysis
Input Validation	CWE-20, CWE-79	✓	–
SQL Injection	CWE-89	✓	–
Command Injection	CWE-78	✓	–
Hard-coded Secrets	CWE-259, CWE-798	✓	–
Path Traversal	CWE-22	✓	–
Denial of Service	CWE-400	–	✓
Insecure Deserialization	CWE-502	✓	–
Improper Authentication	CWE-287, CWE-306	–	✓
Race Condition/Resource Contention	CWE-362	–	✓
Cryptographic Weaknesses	CWE-327	✓	✓
Cross-Site Request Forgery (CSRF)	CWE-352	–	✓
Memory Errors/Buffer Overflow	CWE-119, CWE-125	✓	✓

6.13. Beyond Function-Level Repair: Multi-Function Evaluation

To extend the evaluation beyond the function-level scope, we conducted an additional analysis on a curated subset of 30 examples involving multi-function interactions, where vulnerabilities arise from interdependent logic or shared state across multiple functions. Within this subset, 15 examples exhibited cross-function security issues requiring contextual reasoning beyond isolated code blocks.

On these 15 cases, FDSP successfully resolved 12 vulnerabilities (80%). While the sample size here is deliberately small and exploratory, this result provides encouraging early evidence that FDSP can handle vulnerabilities spanning multiple functions. We also evaluated baselines on the same examples and observed: Direct Prompting resolved 6/15 cases (40%), Self-Debug resolved 7/15 cases (46.7%), Bandit-only feedback resolved 6/15 cases (40%), and Verbalization resolved 6/15 cases (40%).

These results suggest that FDSP's feedback-driven refinement mechanism shows promising potential in capturing broader program semantics compared to existing approaches. We emphasize that this is a preliminary analysis, and scaling FDSP to robust project-level and program-wide settings remains an important avenue for future work.

6.14. Controlled Vulnerability Injection and Qualitative Evaluation

To further assess FDSP's robustness and real-world applicability, we conducted an additional experiment using 50 intentionally injected vulnerabilities spanning common CWE categories, including SQL injection (CWE-89), OS command injection (CWE-78), and improper input validation (CWE-20). A summary of the vulnerability types is presented in Table 9. Each vulnerable function was analyzed using Bandit, and FDSP was applied to detect and remediate the issues. FDSP successfully identified and patched 46 out of 50 injected vulnerabilities (**92%**), demonstrating strong detection and repair performance under controlled conditions.

Table 9. Summary of vulnerability types in the 50-example dataset.

CWE Type (Name)	CWE ID	Total	Percentage (%)
OS Command Injection	CWE-78	22	44.0
SQL Injection	CWE-89	9	18.0
Hard-coded Password	CWE-259	5	10.0
Insecure Deserialization	CWE-502	4	8.0
Race Condition/Improper Synchronization	CWE-362	1	2.0
Multiple Binds to Same Port	CWE-605	2	4.0
Weak Random Number Generation	CWE-330	1	2.0
Improper Input Validation	CWE-20	1	2.0
Insecure Temporary File Creation	CWE-377	1	2.0
Open Redirect	CWE-601	1	2.0
Path Traversal	CWE-22	1	2.0
Total	–	50	100.0

To contextualize these results, we compared FDSP against several baselines. Direct prompting fixed 11 vulnerabilities (22%), self-debugging resolved 9 (18%), Bandit-only feedback resolved 7 (14%), verbalization fixed 5 (10%), and the strategies baseline fixed 2 (4%). FDSP thus delivers a substantial improvement in remediation accuracy over alternative approaches in controlled settings.

We further conducted a qualitative analysis to determine whether FDSP's patches followed secure coding best practices or merely suppressed static analyzer warnings. Manual inspection of 30 refined samples showed that over 90% of FDSP patches implemented principled mitigations such as parameterized queries, whitelisting, proper input sanitization,

and secure API usage rather than superficial modifications. Only a small subset produced syntactic changes without eliminating the core vulnerability. These findings confirm that FDSP not only achieves higher remediation rates but also generates fixes aligned with established secure software engineering practices.

7. Threats to Validity

Although our evaluation demonstrates clear improvements with FDSP across multiple datasets and models, several threats to validity should be considered.

7.1. Internal Validity

A potential internal validity concern stems from the reuse of static analysis tools across both feedback and evaluation. Specifically, Bandit provides diagnostic feedback during FDSP refinement and is also used as one of the evaluation metrics. To mitigate bias, we employ additional static analyzers (CodeQL and Semgrep) as independent evaluators. The consistent ranking of FDSP across all three tools suggests that performance gains are not an artifact of tool reuse. Furthermore, during refinement, Bandit feedback was restricted to line-level alerts (issue location and brief description) without exposing CWE identifiers or labels, preventing information leakage from evaluation to training signals. We also randomized prompt order and ran all models under identical conditions to reduce experimenter bias. To mitigate potential circularity from Bandit being used for both feedback and evaluation, Bandit feedback was sanitized to ensure only vulnerability descriptions—not the final evaluation signals—were passed to the model. Moreover, we incorporate CodeQL and Semgrep as independent evaluators to reduce tool-specific bias.

7.2. Construct Validity

Another possible threat concerns how we operationalize "security improvement." We measure improvement through the reduction in static-analyzer-detected vulnerabilities, which may not capture all dimensions of software security. To strengthen this operationalization, we evaluate each model with multiple analyzers (Bandit, CodeQL, Semgrep) and verify functional correctness using automated unit tests. On a random subset of 100 refined samples, FDSP maintained a 96% functional-pass rate, confirming that vulnerability reduction was not achieved by trivial deletions or non-functional code simplifications. This combination of quantitative and qualitative validation provides a robust construct for measuring genuine security improvement.

7.3. External Validity

Our findings are based on Python code drawn primarily from Stack Overflow and may not generalize to other programming languages or security paradigms. Nevertheless, the diversity of domains within *PythonSecurityEval* including system, web, database, and network code supports broader applicability within high-level programming contexts. Future work will extend FDSP to additional languages and frameworks to assess cross-language robustness.

8. Reproducibility Statement

To support reproducibility, we publicly release all code, dataset, prompts, and evaluation pipelines used in this work. This section summarizes the key components required to replicate our results.

8.1. Model Configuration

All models are evaluated in zero-shot settings. GPT-4, GPT-3.5, and CodeLlama are accessed via API. We use a temperature of 0.0 for initial code generation and 0.9 for iterative

refinements, maximum token limit of 2048, and set the FDSP parameters to $J = 3$ candidate solutions and $K = 3$ iterative refinement cycles. All prompt templates are provided in Figures A2–A4.

8.2. Dataset Collection and Preprocessing

We construct *PythonSecurityEval* by sampling Python-related questions from Stack Overflow (2015–2024). We filter posts containing database, web framework, operating system, cryptography, networking, filesystem, or API interaction tasks. We manually remove incomplete, ambiguous, or context-dependent tasks.

8.3. Security Labeling and Verification

Vulnerabilities are identified using Bandit and CodeQL. We apply both tools to each generated and refined program and record their results.

8.4. Static Analysis Configuration

We use Bandit v1.7.8 (default rule set) and CodeQL v2.15.2 with the Python security-extended query pack. Default query parameters are used, with execution timeouts of 30 s for Bandit and 60 s for CodeQL. We additionally report Semgrep v1.50.0 results for cross-tool comparison.

9. Conclusions and Future Works

In this paper, we systematically evaluated the capacity of LLMs to generate and refine code with respect to security vulnerabilities. Through comprehensive empirical studies on multiple datasets, we identified the prevalence of security issues in LLM-generated code and highlighted the limitations of existing approaches and benchmarks. To address these gaps, we introduced a new benchmark, *PythonSecurityEval*, which covers a broad range of real-world scenarios, and proposed the FDSP framework. FDSP leverages static code analysis to generate targeted feedback, enabling LLMs to iteratively produce and refine potential solutions for vulnerable code. Our results show that FDSP consistently outperforms established baselines across three state-of-the-art LLMs, achieving improvements of up to 33% with Bandit and 12% with CodeQL on the *PythonSecurityEval* dataset.

Looking ahead, several directions remain for future work. First, further research is needed to improve LLMs' ability to address complex or context-dependent vulnerabilities, such as those involving multi-function interactions or external system dependencies. Integrating dynamic analysis and runtime feedback could complement static analysis tools and enable more thorough security assessments. Additionally, future efforts may explore semi-automated or human-in-the-loop refinement processes, combining expert knowledge with LLM capabilities to maximize security and correctness.

Author Contributions: Conceptualization, K.A. and P.T.; methodology, K.A. and P.T.; software, K.A.; validation, K.A.; formal analysis, K.A., A.A., P.T., and M.G.; data curation K.A. and A.A.; writing—original draft preparation; K.A., P.T., and M.G.; visualization, K.A. and A.A.; writing—review and editing; K.A., P.T., and M.G.; supervision; M.G.; project administration, K.A. and M.G.; funding acquisition M.G. All authors have read and agreed to the published version of the manuscript..

Funding: This research was funded by the Saudi Arabian Cultural Mission and supported by a gift from Konica Minolta to Georgia Tech.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The dataset and source code are publicly available at https://github.com/Kamel773/LLM-code-refine (1 December 2025).

Conflicts of Interest: The authors declare no conflicts of interest..

Appendix A

Appendix A.1. Comparative Analysis of CoT and FDSP Performance

Table A1 shows that the FDSP approach consistently reduces the percentage of vulnerable code across all datasets compared to Chain-of-Thought (CoT). For both GPT-4 and CodeLlama, FDSP yields lower vulnerability rates under both Bandit and CodeQL analysis, demonstrating its stronger security-aware refinement capabilities. The improvements are particularly pronounced on LLMSecEval and SecurityEval, where FDSP lowers vulnerabilities by more than half, highlighting its robustness and generalization across diverse benchmarks.

Table A1. Comparison of vulnerable code percentages across datasets using Bandit and CodeQL for Chain-of-Thought (CoT) and FDSP approaches. Lower values indicate fewer vulnerabilities.

Dataset	Approach	GPT-4		CodeLlama	
		Bandit	CodeQL	Bandit	CodeQL
LLMSecEval	CoT	26.8%	7.38%	24.8%	12.0%
	FDSP (Ours)	6.0%	6.7%	14.6%	9.1%
SecurityEval	CoT	18.0%	5.8%	38.5%	17.5%
	FDSP (Ours)	4.1%	8.3%	8.2%	12.1%
Python SecurityEval	CoT	22.7%	8.4%	36.4%	9.14%
	FDSP (Ours)	7.4%	7.7%	8.7%	5.7%

Appendix A.2. Parameter Sensitivity Analysis

To evaluate the effect of FDSP's two key parameters—number of generated solutions (J) and number of refinement iterations (K)—we performed a sensitivity analysis summarized in Table A2. Results show that increasing the number of iterations (K) from 1 to 3 reduces vulnerability rate from 9.5% to 7.4%, after which further increases yield minimal improvement (7.2% at K = 5) while doubling runtime. Similarly, increasing the number of solutions (J) from 1 to 3 substantially lowers vulnerabilities (10.0 $\rightarrow$ 7.4%), but further increases to J = 5 produce negligible gains at high computational cost. Overall, the configuration $J = 3$, $K = 3$ achieves the best balance between security improvement and efficiency, confirming its suitability as a default setting for FDSP in future studies.

Table A2. Sensitivity analysis of FDSP parameters (J = number of generated solutions, K = number of refinement iterations) on *PythonSecurityEval* using GPT-4. Vulnerability rates are measured with Bandit, and runtime represents mean wall-clock time (Second) across samples.

Configuration	Vulnerability Rate	Mean Time (SD)	Observation
$K = 1, J = 3$	9.5%	7.40 (±2.5)	Fewer iterations, higher vulnerability
$K = 2, J = 3$	7.4%	22.8 (±11.4)	Performance improves with iteration depth
$K = 3, J = 3$	7.4%	29.9 (±21.3)	Balanced trade-off (default)
$K = 5, J = 3$	7.2%	59.0 (±72.0)	Marginal gain, high cost
$K = 3, J = 1$	10.0%	8.3 (±5.8)	Limited diversity, weaker coverage
$K = 3, J = 2$	8.7%	15.7 (±7.4)	Moderate improvement
$K = 3, J = 3$	7.4%	29.9 (±21.3)	Optimal balance of diversity and cost
$K = 3, J = 5$	7.4%	57.1 (±46.0)	Diminishing returns beyond $J = 3$

An example of verbalization.

The feedback from the static code analysis regarding the issue tagged as "B608: hardcoded sql expressions" indicates a security concern with the way SQL queries are being constructed in the code. Specifically, the issue points out that the code might be vulnerable to SQL injection attacks due to the use of string-based query construction. This vulnerability arises when input data is concatenated directly into SQL queries without proper sanitization or parameterization, allowing an attacker to manipulate the queries to execute arbitrary SQL commands.
The problematic code is found at line 7, where a cursor's execute method is called to perform a SQL query. The query attempts to count the rows in a table by dynamically inserting the table name into the SQL statement. However, the method of constructing this query by formatting a string is unsafe if the table name comes from an untrusted source or is not properly validated, as it can be exploited to inject malicious SQL.
To mitigate this risk, it is recommended to use parameterized queries or prepared statements provided by the database interface, which ensure that input data is treated as parameters rather than part of the SQL command. This approach helps prevent SQL injection by separating the data from the code that queries the database.

Figure A1. An example of how GPT-4 verbalizes the feedback from the Bandit shown in the Figure 2.

Chain-of-Thought.

Analyze the provided Python function for potential security vulnerabilities.

Step-by-step reasoning (Chain-of-Thought)
1. **Understand the function**—summarize what the function does and how it handles input, output, and external interactions.
2. **Identify possible vulnerabilities**—check for issues such as:
- Input validation and sanitization
- Insecure file handling or system calls
- Hardcoded credentials or secrets
- Missing authentication or authorization
- Use of unsafe functions or libraries
3. **Assess severity and impact**—explain why each issue is dangerous and what attacker actions it might enable.
4. **Propose a secure fix**—rewrite or patch the function to mitigate the vulnerabilities.
5. **Validate the fix**—confirm that the corrected version is secure and maintains functionality.

Instruction:
Does the provided function have any security issues?
If **yes**, follow the reasoning steps above and provide the **secure, fixed version** of the code.

Python code:
{code}

Figure A2. Chain-of-Thought Prompting.

Bandit feedback Prompting.

Static Analysis Feedback:
{Feedback}
#Task:
The static code analysis tool has identified potential security issues in the following Python code:
{code}
Please review the feedback and apply the necessary security fixes to the code.
When rewriting the code:
Address all issues highlighted in the static analysis feedback.
Preserve the intended functionality of the original program.
Follow secure coding best practices, including:
- Proper input validation and sanitization
- Safe file and system handling
- Avoidance of insecure library functions
- Secure handling of secrets and credentials

Appropriate authentication and authorization checks (if applicable)

Figure A3. Bandit feedback Prompting.

Table A3. Common weakness enumeration (CWE) IDs and descriptions.

CWE ID	Description
CWE-20	Improper Input Validation
CWE-22	Improper Limitation of a Pathname to a Restricted Directory (‘Path Traversal’)
CWE-78	Improper Neutralization of Special Elements used in an OS Command (‘OS Command Injection’)
CWE-79	Improper Neutralization of Input During Web Page Generation (‘Cross-site Scripting’)
CWE-89	Improper Neutralization of Special Elements used in an SQL Command (‘SQL Injection’)
CWE-94	Improper Control of Generation of Code (‘Code Injection’)
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer
CWE-200	Exposure of Sensitive Information to an Unauthorized Actor
CWE-284	Improper Access Control
CWE-287	Improper Authentication
CWE-306	Missing Authentication for Critical Function
CWE-352	Cross-Site Request Forgery (CSRF)
CWE-400	Uncontrolled Resource Consumption
CWE-502	Deserialization of Untrusted Data
CWE-611	Improper Restriction of XML External Entity Reference (XXE)
CWE-703	Improper Handling of Exceptional Conditions
CWE-798	Use of Hard-coded Credentials
CWE-120	Buffer Copy without Checking Size of Input (‘Classic Buffer Overflow’)
CWE-125	Out-of-bounds Read
CWE-190	Integer Overflow or Wraparound
CWE-295	Improper Certificate Validation
CWE-416	Use After Free
CWE-434	Unrestricted Upload of File with Dangerous Type
CWE-476	NULL Pointer Dereference
CWE-732	Incorrect Permission Assignment for Critical Resource
CWE-787	Out-of-bounds Write
CWE-862	Missing Authorization
CWE-918	Server-Side Request Forgery (SSRF)
CWE-1021	Improper Restriction of Rendered UI Layers or Frames
CWE-1295	Debug Features Enabled in Production

Table A4. Examples of corrected potential solutions generated by FDSP for SQL injection vulnerabilities, where GPT-4 was unable to refine the vulnerable code.

Examples	Generated Potential Solutions
Example 1	(1) Use Parameterized Queries: The most effective way to prevent SQL injection is to use parameterized queries, also known as prepared statements. This strategy doesn't form SQL commands by using string operations. Instead, it designates placeholders for each parameter, and the database engine will fill these placeholders securely. However, SQLite doesn't allow placeholders for table names, so it might not be possible to bind table names to placeholders in this case. (2) Input Validation: Another strategy is to validate the user inputs or in this case, the table_name variable. We can ensure that it only contains valid table name characters and does not contain dangerous SQL syntax. This can be done using a whitelist of allowed characters or a regular expression (regex) to match valid table names. (3) Devise a Whitelisting Mechanism: This would involve creating a list of tables that are allowed to be accessed via the "total_rows" function. By checking if the table name provided to the function is present in the whitelisted table names, the code can ensure that only legitimate queries are executed, proving effective against SQL injection.
Example 2	(1) Parameterized Query: The main solution for preventing SQL injection is to use parameterized queries. This entails using placeholders for data in your SQL statements and then providing the data to the SQL interpreter separately so it's correctly and safely quoted. Unfortunately, SQLite in Python doesn't directly support parameterization of table or column names. So, the programmer needs to ensure they sanitize their table or column name inputs. (2) Use of Sanitizing function: To handle not directly supported parameterization, we can also create a sanitizing function to ensure the validity of the table name provided to prevent SQL injection though still possible to think about the limitation of sanitizing process. (3) Use of Database API: Instead of allowing the program user to provide arbitrary table or column names, consider using a Database API to generate these names. This will provide an interface for the user to view and interact with the data, but not change the structure of the SQL queries themselves. This significantly reduces the risk of SQL injection attacks because the user may only affect the value in a field, not the structure of the SQL queries since the table names are not user inputs.
Example 3	(1) Use Parameterized Queries: This involves using placeholders instead of directly inserting values into the statements. Parameters are used to help differentiate between the SQL code and the data passed through. When executed, the SQL statement compiles together with the parameters as one command which prevents attackers from interfering with the structure of SQL queries. (2) Use Object-Relational Mapping (ORM) Libraries: Technologies such as SQLAlchemy, Django ORM, can be used to handle operations with databases. These libraries usually come with secure features that protect from SQL injection attacks. (3) Use Database Views: Business logic could be implemented in the database side to decrease the complexity of SQL built in the application, and to encapsulate the database structure. Views could be used to give simplified perspective of the database and to restrict access to sensitive parts of the database schema. Reducing database control and visibility minimizes the potential damage of SQL Injection attacks.

FDSP Prompting.

```
# Static Analysis Feedback:
{Feedback}
#Task:
The static analysis tool has reported security concerns in the Python function below.
Please propose three distinct remediation strategies, each addressing the identified
issues from a different secure-engineering perspective.

For each strategy:

- Provide a clear explanation of the approach.
- Describe how it mitigates the reported vulnerabilities.
- Include updated code reflecting the strategy.
- Ensure the fix maintains the intended functionality.

# Python code:
{code}
```

Figure A4. FDSP prompting.

References

1. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. In Proceedings of the Advances in Neural Information Processing Systems 33, Online, 6–12 December 2020; pp. 877–1901.
2. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention is All you Need. In *Advances in Neural Information Processing Systems*; Curran Associates, Inc.: Red Hook, NY, USA, 2017; Volume 30.
3. Rozière, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X.E.; Adi, Y.; Liu, J.; Remez, T.; Rapin, J.; et al. Code llama: Open foundation models for code. *arXiv* **2023**, arXiv:2308.12950.
4. Touvron, H.; Martin, L.; Stone, K.; Albert, P.; Almahairi, A.; Babaei, Y.; Bashlykov, N.; Batra, S.; Bhargava, P.; Bhosale, S.; et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv* **2023**, arXiv:2307.09288. [CrossRef]
5. Yu, T.; Zhang, R.; Yang, K.; Yasunaga, M.; Wang, D.; Li, Z.; Ma, J.; Li, I.; Yao, Q.; Roman, S.; et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv* **2018**, arXiv:1809.08887.
6. Lachaux, M.A.; Roziere, B.; Chanussot, L.; Lample, G. Unsupervised translation of programming languages. *arXiv* **2020**, arXiv:2006.03511. [CrossRef]
7. Shypula, A.; Madaan, A.; Zeng, Y.; Alon, U.; Gardner, J.; Hashemi, M.; Neubig, G.; Ranganathan, P.; Bastani, O.; Yazdanbakhsh, A. Learning performance-improving code edits. *arXiv* **2023**, arXiv:2302.07867.
8. Pearce, H.; Tan, B.; Ahmad, B.; Karri, R.; Dolan-Gavitt, B. Examining Zero-Shot Vulnerability Repair with Large Language Models. In Proceedings of the IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–25 May 2023.
9. Wong, M.F.; Guo, S.; Hang, C.N.; Ho, S.W.; Tan, C.W. Natural language generation and understanding of big code for ai-assisted programming: A review. *Entropy* **2023**, *25*, 888. [CrossRef]
10. Hermann, K.; Peldszus, S.; Steghöfer, J.P.; Berger, T. An Exploratory Study on the Engineering of Security Features. In Proceedings of the International Conference on Software Engineering (ICSE), Ottawa, ON, Canada, 27 April–3 May 2025.
11. Spiess, C.; Gros, D.; Pai, K.S.; Pradel, M.; Rabin, M.R.I.; Alipour, A.; Jha, S.; Devanbu, P.; Ahmed, T. Calibration and correctness of language models for code. In Proceedings of the International Conference on Software Engineering (ICSE), Lisbon, Portugal, 14–20 April 2024.
12. Zhang, T.; Yu, Y.; Mao, X.; Wang, S.; Yang, K.; Lu, Y.; Zhang, Z.; Zhao, Y. Instruct or Interact? Exploring and Eliciting LLMs' Capability in Code Snippet Adaptation Through Prompt Engineering. In Proceedings of the International Conference on Software Engineering (ICSE), Lisbon, Portugal, 14–20 April 2024.
13. Chen, X.; Lin, M.; Schärli, N.; Zhou, D. Teaching large language models to self-debug. *arXiv* **2023**, arXiv:2304.05128. [CrossRef]
14. Athiwaratkun, B.; Gouda, S.K.; Wang, Z. Multi-lingual Evaluation of Code Generation Models. In Proceedings of the International Conference on Learning Representations (ICLR), Kigali, Rwanda, 1–5 May 2023.
15. Siddiq, M.L.; Casey, B.; Santos, J.C.S. A Lightweight Framework for High-Quality Code Generation. *arXiv* **2023**, arXiv:2307.08220. [CrossRef]
16. Le, H.; Sahoo, D.; Zhou, Y.; Xiong, C.; Savarese, S. INDICT: Code Generation with Internal Dialogues of Critiques for Both Security and Helpfulness. In Proceedings of the Thirty-Eighth Annual Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 9–15 December 2024.
17. Tony, C.; Mutas, M.; Díaz Ferreyra, N.; Scandariato, R. LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. In Proceedings of the 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR), Melbourne, Australia, 15–16 May 2023.
18. Lightman, H.; Kosaraju, V.; Burda, Y.; Edwards, H.; Baker, B.; Lee, T.; Leike, J.; Schulman, J.; Sutskever, I.; Cobbe, K. Let's verify step by step. In Proceedings of the Twelfth International Conference on Learning Representations, Kigali, Rwanda, 1–5 May 2023.
19. Huang, J.; Chen, X.; Mishra, S.; Zheng, H.S.; Yu, A.W.; Song, X.; Zhou, D. Large Language Models Cannot Self-Correct Reasoning Yet. In Proceedings of the Twelfth International Conference on Learning Representations, Kigali, Rwanda, 1–5 May 2023.
20. Siddiq, M.; Santos, J. 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 S22), Virtually, 18 November 2022.
21. Zheng, Q.; Xia, X.; Zou, X.; Dong, Y.; Wang, S.; Xue, Y.; Wang, Z.; Shen, L.; Wang, A.; Li, Y.; et al. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv* **2023**, arXiv:2303.17568.
22. Austin, J.; Odena, A.; Nye, M.; Bosma, M.; Michalewski, H.; Dohan, D.; Jiang, E.; Cai, C.; Terry, M.; Le, Q.; et al. Program synthesis with large language models. *arXiv* **2021**, arXiv:2108.07732. [CrossRef]
23. Zhou, S.; Alon, U.; Xu, F.F.; Wang, Z.; Jiang, Z.; Neubig, G. DocPrompting: Generating Code by Retrieving the Docs. In Proceedings of the International Conference on Learning Representations (ICLR), Kigali, Rwanda, 1–5 May 2023.
24. Nijkamp, E.; Hayashi, H.; Xiong, C.; Savarese, S.; Zhou, Y. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. In Proceedings of the International Conference on Learning Representations (ICLR), Kigali, Rwanda, 1–5 May 2023.

25. Allamanis, M.; Jackson-Flux, H.; Brockschmidt, M. Self-supervised bug detection and repair. *Adv. Neural Inf. Process. Syst.* **2021**, *34*, 27865–27876.
26. Rasooli, M.S.; Tetreault, J.R. Yara Parser: A Fast and Accurate Dependency Parser. *arXiv* **2015**, arXiv:1503.06733. [CrossRef]
27. Nam, D.; Macvean, A.; Hellendoorn, V.; Vasilescu, B.; Myers, B. Using an LLM to Help With Code Understanding. In Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE), Lisbon, Portugal, 14–20 April 2024; IEEE Computer Society: Piscataway, NJ, USA, 2024; pp. 1–13.
28. Wang, J.; Huang, Y.; Chen, C.; Liu, Z.; Wang, S.; Wang, Q. Software testing with large language models: Survey, landscape, and vision. *IEEE Trans. Softw. Eng.* **2024**, *50*, 911–936. [CrossRef]
29. Aggarwal, P.; Madaan, A.; Yang, Y.; Mausam. Let's Sample Step by Step: Adaptive-Consistency for Efficient Reasoning and Coding with LLMs. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, Singapore, 6–10 December 2023; pp. 12375–12396.
30. Alrashedy, K.; Hellendoorn, V.J.; Orso, A. Learning Defect Prediction from Unrealistic Data. *arXiv* **2023**, arXiv:2311.00931.
31. Chakraborty, S.; Krishna, R.; Ding, Y.; Ray, B. Deep learning based vulnerability detection: Are we there yet? *IEEE Trans. Softw. Eng.* **2021**, *48*, 3280–3296. [CrossRef]
32. Andrew, G.; Gao, J. Scalable training of L1-regularized log-linear models. In Proceedings of the 24th International Conference on Machine Learning, Corvallis, OR, USA, 20–24 June 2007; pp. 33–40.
33. Ando, R.K.; Zhang, T. A Framework for Learning Predictive Structures from Multiple Tasks and Unlabeled Data. *J. Mach. Learn. Res.* **2005**, *6*, 1817–1853.
34. Bhatt, M.; Chennabasappa, S.; Nikolaidis, C.; Wan, S.; Evtimov, I.; Gabi, D.; Song, D.; Ahmad, F.; Aschermann, C.; Fontana, L.; et al. Purple llama cyberseceval: A secure coding benchmark for language models. *arXiv* **2023**, arXiv:2312.04724.
35. Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegreffe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; et al. Self-refine: Iterative refinement with self-feedback. In Proceedings of the Neural Information Processing Systems, New Orleans, LA, USA, 10–16 December 2023.
36. Olausson, T.X.; Inala, J.P.; Wang, C.; Gao, J.; Solar-Lezama, A. Is Self-Repair a Silver Bullet for Code Generation? In Proceedings of the Twelfth International Conference on Learning Representations, Kigali, Rwanda, 1–5 May 2023.
37. Gou, Z.; Shao, Z.; Gong, Y.; Shen, Y.; Yang, Y.; Duan, N.; Chen, W. CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing. *arXiv* **2023**, arXiv:2305.11738.
38. Elgohary, A.; Meek, C.; Richardson, M.; Fourney, A.; Ramos, G.; Awadallah, A.H. NL-EDIT: Correcting Semantic Parse Errors through Natural Language Interaction. In Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL), Online, 6–11 June 2021.
39. Bai, Y.; Jones, A.; Ndousse, K. Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback. *arXiv* **2023**, arXiv:2204.05862.
40. Yasunaga, M.; Liang, P. Graph-based, self-supervised program repair from diagnostic feedback. In Proceedings of the International Conference on Machine Learning, Virtual, 13–18 July 2020; PMLR: New York, NY, USA, 2020; pp. 10799–10808.
41. Bafatakis, N.; Boecker, N.; Boon, W.; Salazar, M.C.; Krinke, J.; Oznacar, G.; White, R. Python coding style compliance on stack overflow. In Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 25–31 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 210–214.
42. Peng, B.; Galley, M.; He, P.; Cheng, H.; Xie, Y.; Hu, Y.; Huang, Q.; Liden, L.; Yu, Z.; Chen, W.; et al. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv* **2023**, arXiv:2302.12813. [CrossRef]
43. Yang, K.; Tian, Y.; Peng, N.; Klein, D. Re3: Generating longer stories with recursive reprompting and revision. In Proceedings of the Conference on Empirical Methods in Natural Language Processing, Abu Dhabi, United Arab Emirates, 7–11 December 2022.
44. Wang, B.; Shin, R.; Liu, X.; Polozov, O.; Richardson, M. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Online, 5–10 July 2020; Association for Computational Linguistics: Stroudsburg, PA, USA, 2020; pp. 7567–7578.
45. Scholak, T.; Schucher, N.; Bahdanau, D. PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models. In Proceedings of the Conference on Empirical Methods in Natural Language Processing, Punta Cana, Dominican Republic, 7–11 November 2021; Association for Computational Linguistics: Stroudsburg, PA, USA, 2021.
46. Gusfield, D. *Algorithms on Strings, Trees and Sequences*; Cambridge University Press: Cambridge, UK, 1997.
47. Zhuo, T.Y.; Vu, M.C.; Chim, J.; Hu, H.; Yu, W.; Widyasari, R.; Yusuf, I.N.B.; Zhan, H.; He, J.; Paul, I.; et al. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *arXiv* **2024**, arXiv:2406.15877.
48. Gao, L.; Madaan, A.; Zhou, S.; Alon, U.; Liu, P.; Yang, Y.; Callan, J.; Neubig, G. Pal: Program-aided language models. In Proceedings of the International Conference on Machine Learning, Honolulu, HI, USA, 23–29 July 2023; PMLR: New York, NY, USA, 2023; pp. 10764–10799.

49. Akyürek, A.F.; Akyürek, E.; Kalyan, A.; Clark, P.; Wijaya, D.; Tandon, N. RL4F: Generating natural language feedback with reinforcement learning for repairing model outputs. In Proceedings of the Annual Meeting of the Association of Computational Linguistics 2023, Toronto, ON, Canada, 9–14 July 2023; Association for Computational Linguistics (ACL): Stroudsburg, PA, USA, 2023; pp. 7716–7733.
50. Aho, A.V.; Ullman, J.D. *The Theory of Parsing, Translation and Compiling*; Prentice-Hall: Englewood Cliffs, NJ, USA, 1972; Volume 1.
51. CodeQL. Available online: https://codeql.github.com (accessed on 4 March 2025).