Can LLMs Patch Security Issues?

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have shown impressive proficiency in code generation. Unfortunately, these models share a weakness with their human counterparts: producing code that inadvertently has security vulnerabilities. These vulnerabilities could allow unauthorized attackers to access sensitive data or systems, which is unacceptable for safety-critical applications. We propose Feedback-Driven Security Patching (FDSP), where LLMs automatically refine generated, vulnerable code. Our approach leverages automatic static code analysis to empower the LLM to generate and implement potential solutions to address vulnerabilities. We address the research community's needs for safe code generation by introducing a large-scale dataset, PythonSecurityEval, covering the diversity of real-world applications, including databases, websites and operating systems. We empirically validate that FDSP outperforms prior work that uses self-feedback from LLMs by up to 17.6% through our procedure that injects targeted, external feedback. Code and data are attached.

1 Introduction

Although Large language models (LLMs), such as GPT-4 (Brown et al., 2020) and CodeLlama (Rozière et al., 2023), are powerful tools for code generation, they are prone to generating vulnerable code (Pearce et al., 2023). LLMs have shown high-competency for a wide variety of code generation tasks, such as for producing code from natural language (Yu et al., 2018), code translation (Lachaux et al., 2020), and code optimization (Shypula et al., 2023). Utilizing LLMs for code generation has been shown to increase developers' productivity with writing and explaining code, and fixing bugs (Wong et al., 2023). To enhance code refinement with LLMs, recent work by (Chen et al., 2023), proposed a self-debugging technique, where LLMs generate code, and then the code is sent back to the same LLM to produce feedback and then refine the code.

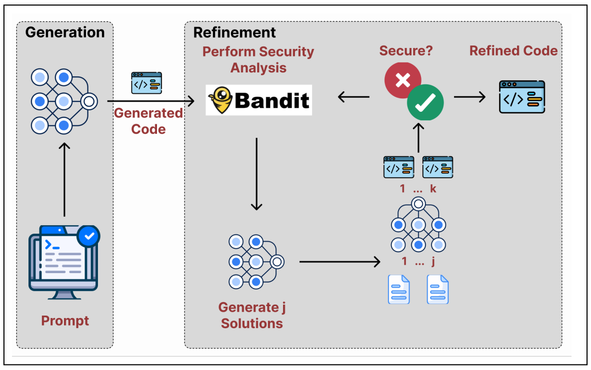

Figure 1: Overview of our approach: Initially, the LLMs generates code. This code is subsequently analyzed for security vulnerabilities using Bandit, a tool for static code analysis, to determine if there are any security issues. Following this, feedback on any identified issues is incorporated into the LLMs to generate possible solutions for resolving the security issues. Finally, each proposed solution is sent back to the LLMs for code refinement.

However, code generated or refined by LLMs could produces security vulnerabilities. Vulnerabilities in code allow unauthorized users to access sensitive data or systems. For example, attackers can manipulate your SQL queries to gain access to the database, a technique known as SQL injection. This is due to the following reasons: 1) LLMs may not always recognize security issues, often producing code with vulnerabilities, particularly when the code interacts with external service and system, and 2) LLMs might struggle to fix security issues in code due to their limited understanding of security vulnerabilities and lack of specific security knowledge, which has been explored in prior literature (Athiwaratkun et al., 2023; Siddiq et al., 2023).

One potential approach to mitigate these security vulnerabilities is to train LLMs to recognize and patch security vulnerabilities. However, this method also has several significant challenges. Firstly, it requires a large labeled dataset that accurately distinguishes between vulnerabilities and non-vulnerabilities to train the LLMs effectively. Collecting such a human-labeled dataset is costly and time-consuming. Additionally, there is a critical need for robust feedback mechanisms during the training process. LLMs require accurate feedback to learn from their mistakes, and this feedback must come from security experts.

In this paper, we address the key limitations of prior work by developing Feedback-Driven Security Patching (FDSP) and the PythonSecurityEval benchmark. In FDSP, LLMs generate potential solutions to fix the security issues in the generated code. This process involves analyzing the generated code through static code analysis to identify any security issues and produce feedback. LLMs utilize the feedback to generate potential solutions, and then each potential solution along with the generated code is sent back to the LLMs to refine the code. Next, we curate an extensive dataset from Stack Overflow, called *PythonSecurityEval*, as existing security evaluation datasets are quite limited and insufficient to evaluate a model's ability to produce non-vulnerable code. Our dataset originates from real-world applications, providing diversity with prompts to generate code for a variety of applications, including databases (such as SQL, MySQL, etc.), URLs, operating systems, and websites (e.g., Flask). We consider these types of applications as primary sources of common vulnerabilities, including SQL injection, cross-site scripting (XSS), broken access control, and command injection.

To summarize, Our work presents three key contributions:

- We proposes (FDSP), a technique that enhances LLMs to generate potential solutions for fixing security issues in the generated code by receiving feedback from static code analysis.

- We demonstrate that FDSP outperforms prior works that use self-feedback by up to 17.6%. We empirically evaluate the capabilities of the most advanced LLMs, including GPT-4, GPT-3.5, and CodeLlama, in generating and refining insecure code. We utilize three benchmarks (including ours) and apply five baseline techniques for this evaluation.

- We present *PythonSecurityEval*, a dataset designed to evaluate the ability of LLMs to generate secure code. Our dataset contains 470 natural language prompts.

2 Related work

We discuss three a categories of previous work: LLMs for code generation and refinement of LLMs, as well as the source of feedback.

Language models for code: Code generation models have become a very 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, where users prompt an LLM with natural language instructions about the coding task, and the LLM generates the corresponding code. Examples include CodeLlama (Rozière et al., 2023) and CodeGeeX (Zheng et al., 2023), which achieve state-of-the-art performance on the Mostly Basic Programming Problems (MBPP) dataset (Austin et al., 2021). The DocPrompting approach furtherdemonstrates that prompting language models with code-documentation improves code generation performance on models such as CodeT5, CodeX, and GPT-Neo on MBPP (Zhou et al., 2023). Beyond code-generation, the proficiency of LLMs also extends to code-translation (Roziere et al., 2020), code-repair (Allamanis et al., 2021), creating documentation (Nam et al., 2024), code-testing (Wang et al., 2024) and defect-prediction (Alrashedy et al., 2023). Our interest lies in exploring how these various capabilities shown by LLMs can be applied towards addressing security issues in LLM-generated code.

Refinement of LLMs: Recent works have demonstrated that LLMs can refine their own output or adapt based on feedback from external tools or human input. Self-Refine (Madaan et al., 2023) generates feedback and refines its output to improve the quality of the generated answers across 7 tasks using state-of-the-art models such as GPT-3.5 and GPT-4. Additionally, a similar technique called self-debug (Chen et al., 2023) enables code-generation models to debug initially generated code using feedback either from the same LLM, unit test results, or compiler error messages. The 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 (Olausson et al., 2023), where sought to produce feedback specifically focusing on why any faulty code-snippet is incorrect. Another study (Gou et al., 2023) 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 fix security issues in generated code.

The source of feedback: Human-feedback is the most effective and accurate source of feedback; however, it is also costly and time-intensive (Elgohary et al., 2021; Yuntao Bai, 2023). An alternative way to obtain feedback is from external tools such as compiler error messages for program repair (Yasunaga and Liang, 2020) and Pylint, a static code analyzer, for improving Python coding standards (Bafatakis et al., 2019). Additionally, previous studies have proposed techniques on how to obtain feedback from LLMs, including the LLM-Augmenter system (Peng et al., 2023) and Recursive Reprompting and Revision framework (Yang et al., 2022). Unlike these works, which utilize feedback from either LLMs or external tools, our approach combines feedback from both external tools and LLMs, where the static code analysis provides feedback about the generated code, and then LLMs generate potential solutions for addressing security issues in code.

3 FDSP Framework

Our approach, FDSP, seeks to identify and resolve vulnerabilities in code generated by an LLM. The principal component of FDSP is the use of static code analysis (Bandit) to generate solutions to potentially vulnerable code We take a four-step approach: (i) code generation, (ii) code testing, (iii) solution generation and (iv) code refinement. The complete algorithm for FDSP is provided in Algorithm 1.

3.1 Code generation

Given a natural language description of a Python function denoted as x, an LLM generates a Python program y according to $P_{LM}(y|x)$. Then, the program y is executed and if there is compiler error message, we send the program y with $\{e_c\}$ to the LLMs to fix the error, as describe in Eqn 2. The initial generated code can be describe as follows:

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

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

We consider a zero-shot setting because, in real-world use cases, users prompt LLMs to generate code directly without providing examples.

Algorithm 1 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 $P_{LM}(y_i|x)$

```
Initialize output y_i from P_LM(x)
// Generate potential solutions (Eqn. 4)
S ~ P_LM(y, ℜ, j, p)
//Iteration for each potential solution (Eqn. 5)
for s ∈ S do
    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
```

3.2 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[1], a static code analysis tool designed to detect common vulnerabilities in Python functions. Bandit constructs the Abstract Syntax Tree[2] (AST) for a Python function and conducts analysis on the AST nodes. Subsequently, Bandit, denoted by δ, generates a report $\Re$ about the code y. Then, we pass the report $\Re$ from Bandit to the LLMs to generate potential solutions to fix the vulnerabilities. we can describe the Bandit report as follows:

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

Figure 2 shows an example of Bandit feedback for the code snippet in 1.

[1] https://github.com/PyCQA/bandit
[2] https://greentreesnakes.readthedocs.io/en/latest/

3.3 Generating potential solutions

Static code analysis can detect vulnerabilities but cannot fix them. The primary contribution of FDSP is to leverage static code analysis to enable LLMs to generate or suggest multiple solutions S to address these vulnerabilities, as represented in equation 4.

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

Where the J present the total number of generated potential solutions, and the p is the instruction that given to the LLMs. Figure 1 illustrates this process: the LLM generates code vulnerable to SQL injection, we then provide the LLM with report from Bandit, and subsequently, the LLM generates three different potential solutions: *1) Use Parameterized Queries, 2) Manual Escape and Quote Table Name and 3) Use an ORM (Object-Relational Mapping) Library.*

3.4 Code refinement

We use the generated potential solutions $S \sim s_1 \oplus s_2 \ldots \oplus s_j$, from the previous step as feedback to fix the vulnerable code y. In the previous step the LLMs generate J unique solutions to fix the security issues. Then, each potential solution and vulnerable code is fed back into the LLM multiple times, denoted as K. The idea behind the K iterations for each solution is to allow the LLMs to generate as many fixes as possible for the vulnerable code. The refinement process is terminated when the Bandit detects no security issues or when the iteration reaches the maximum number of iterations, K, for all potential solutions S, as shown in Algorithm 1.

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

4 Experimental Settings

Existing benchmarks, LLMSecEval and SecurityEval, are insufficient for large-scale evaluation due to their limited size. 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 fixing security issues.

```python
# The prompt: Write a Python function
    to return the total number of rows
    in SQLite.

import sqlite3
def get_total_rows(database_name,
    table_name):
    conn = sqlite3.connect(
    database_name)
    cursor = conn.cursor()
    cursor.execute("SELECT COUNT(*)
    FROM {}".format(table_name))
    result = cursor.fetchone()
    conn.close()
    return result[0]
```

Code Snippet 1: An example of a prompt from PythonSecurityEval dataset where GPT-4 generates vulnerable code of the SQL injection type. The report from Bandit is shown in 2.

4.1 Benchmarks

LLMSecEval: A dataset containing natural language prompts to evaluate LLMs on generating secure source code (Tony et al., 2023). This dataset covers the majority of the Top 25 Common Weakness Enumeration (CWE) scenarios from 2021, addressing various security concerns. It consists of a total of 150 prompts, where each prompt is a natural language description for generating code.

SecurityEval: This dataset, proposed by (Siddiq and Santos, 2022), is used to evaluate LLMs on their ability to generate secure Python programs. It comprises 121 natural language prompts covering 75 different types of vulnerabilities. Each prompt includes the header of a Python function along with comments that describe the function's purpose.

PythonSecurityEval: We collected a new large dataset from Stack Overflow to address the limitation of the existing dataset. The current datasets are limited in size and diversity, which is not very efficient in evaluating the ability of LLMs to generate secure code and their capacity to fix security vulnerabilities. The PythonSecurityEval includes natural language prompts intended to generate Python functions that cover diverse real-world applications. Our dataset is three times larger than those used in LLMSecEval and SecurityEval, with a total of 470 prompts.

Our benchmark is diverse, 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 the Python code interacts with the operating system, allowing attackers to gain unauthorized access to data and systems. We include examples that generate Python code with access to the operating system and diagnose how the 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.

An example of Bandit report.
Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector through string-based query construction. *Line 7:cursor.execute("SELECT COUNT(*) FROM ".format(table_name))*

Figure 2: An example of the report generated by Bandit, a static code analysis tool, for the vulnerable code in Code Snippet 1.

4.2 Baselines

Direct Prompting: This approach involves sending the generated code back to the LLMs with the instruction: *Does the provided function have a security issue? If yes, please fix the issue.* If the LLMs detect any security issues in the code, they will fix the issue and generate secure code. This serves as a baseline to demonstrate the LLM's ability to detect and resolve security issues without additional feedback.

Self-Debugging: The concept of self-debugging (Chen et al., 2023) involves LLMs generating code and then debugging the generated code themselves without help from humans or external tools. The initial step in self-debugging is that the LLMs generating the code; subsequently, the generated code is sent back to the same LLMs to generate feedbackFinally, both the generated code and the explanations are fed back to the LLM to correct any existing bugs. This approach has demonstrated an improvement in fixing incorrect code that either did not compile or failed to pass unit tests. We are interested in investigating how the self-debugging approach enhances LLMs in addressing security issues. The feedback, which provides an explanation of the code, could enhance the LLMs' understanding of the code and improve their ability to fix security vulnerabilities.

Bandit feedback: Bandit produces a report if there is any security issue in the code as shown in Figure 2, we use this report as a feedback to LLMs to refine the vulnerable code. This is similar to prior works where external tools provide feedback to the LLMs to refine their outputs (Gao et al., 2023; Akyürek et al., 2023). In our baseline, we provide the LLMs with the vulnerable code along with the feedback from Bandit, which includes the type of security issue and indicates the specific line of code where the issue is found. Bandit's feedback does not provide a solution to fix the issue; it simply highlights the problematic line and type of issue.

Verbalization: We verbalize the feedback from Bandit, via an LLM, to produce more understandable and actionable feedback to resolve security issues and defective code. The verbalized feedback provides a detailed explanation in natural language of the specialized output from Bandit, as illustrated in Figure 6. This expanded explanation offers deeper insights into the security issues and may suggest solutions to address the vulnerabilities. We use verbalized feedback as a baseline and compare it with the direct use of Bandit feedback and FDSP.

4.3 Evaluation metrics

In our study, we evaluate the accuracy of how frequently LLMs generate and refine vulnerable code. This is done by dividing the total number of generated vulnerable code by the total number of generated code. We report the accuracy in Table 1. To verify whether the generated code is vulnerable, we use Bandit, an automated tool designed to identify vulnerabilities in Python code.

4.4 Models

We aim to evaluate state-of-the-art LLMs for code generation, including GPT-4, GPT-3.5 *"gpt-3.5-turbo-instruct"*(Brown et al., 2020) using OpenAI API, and CodeLlama *"CodeLLama-Instruct-34B"*(Rozière et al., 2023) from Huggingface, to generate secure code. Additionally, we assess these models' ability to refine insecure code based on feedback from Bandit.

5 Experimental Results

In this section, we evaluate our approach to fix the vulnerabilities in code with three LLMs in subsection 5.1, and we discuss the main finding in subsection 5.2.

5.1 Results

Table 1 presents the summary of our results on how frequently LLMs generate and refine vulnerable code across five approaches.

For the LLMSecEval and SecurityEval datasets, between 28% and 46% of the code generated have vulnerabilities. The methods of direct prompting and self-debugging slightly help to fix the vulnerabilities, with improvement percentages of less than 10% for GPT-3.5 and CodeLlama. However, for GPT-4, the average improvement is 15%. This suggests that LLMs can provide feedback to fix their generated vulnerable code without external input, but the extent of improvement is not satisfactory. In the third approach, where LLMs receive feedback from the Bandit, there's about a 30% improvement for GPT-4 and up to a 24% improvement for GPT-3.5 and CodeLlama. Additionally, verbalization, which involves articulating feedback from the Bandit, shows a slight improvement of about 2%. The FDSP approach consistently shows improvement in fixing vulnerabilities across the three LLMs and the two datasets, LLMSecEval and SecurityEval, when compared to the baseline.

We can observe that more than 40% of the code generated by PythonSecurityEval has security issues across varying LLMs. The effectiveness of fixing the vulnerable code is somewhat consistent for both direct prompting and self-debugging techniques across all LLMs. In other approaches, where we provide the LLMs with feedback from Bandit, the results improve by more than 15% compared to direct prompting and self-debugging. The FDSP approach 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 21.0% to 13.6%). This suggests that LLMs can propose potential solutions and provide useful feedback to fix 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.

5.2 Analysis

In this subsection, we discuss the key findings of our results regarding the frequency with which LLMs generate and fix vulnerable code. We also discuss the most common types of code vulnerabilities generated by LLMs, as well as unresolved vulnerabilities in the PythonSecurityEval dataset.

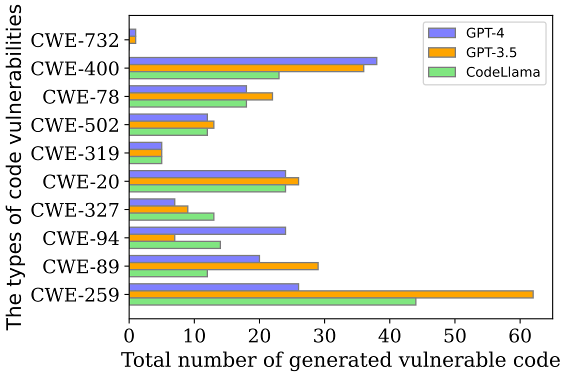

Figure 3: The figure illustrates the total number of the most common types of security issues (Top 10) in code generated for the PythonSecurityEval dataset.

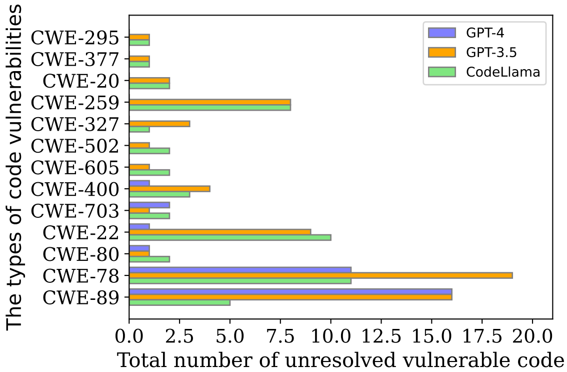

Figure 4: The figure displays the total number of top unresolved security issues for the PythonSecurityEval dataset.

The feedback provided by Bandit significantly enhances the LLMs' capabilities in addressing security issues, unlike other methods that exclude Bandit's feedback. While simple baselines such as direct prompts and self-debugging can be helpful, but ultimately are not very effective in fixing security issues in code. These methods are somewhat beneficial in tackling straightforward vulnerabilities. As Table 1 shows that all approaches incorporating feedback from Bandit significantly improve accuracy over simple methods across all models and datasets. The FDSP method boosts the LLMs' ability to generate potential solutions based on Bandit's feedback. It is evident that LLMs lack an understanding of code vulnerabilities and struggle to fix these issues without feedback from Bandit. Our FDSP approach significantly enhances the performance of GPT-3.5 and CodaLlama, exceeding the results achieved by either directly incorporating

Dataset	Approach	GPT 4	GPT 3.5	CodeLlama
LLMSecEval	Generated code	38.2	34.2	28.6
	Direct prompting	35.3 ($\downarrow$ 2.6)	28.0 ($\downarrow$ 6.0)	24.0 ($\downarrow$ 4.6)
	Self-debugging	24.0 ($\downarrow$ 14.0)	28.0 ($\downarrow$ 6.0)	24.6 ($\downarrow$ 4.0)
	Bandit feedback	8.0 ($\downarrow$ 30.0)	18.6 ($\downarrow$ 15.33)	18.0 ($\downarrow$ 10.6)
	Verbalization	7.3 ($\downarrow$ 30.6)	18.0 ($\downarrow$ 16.0)	16.6 ($\downarrow$ 12.0)
	FDSP	**6.0** ($\downarrow$ 32.0)	**12.6** ($\downarrow$ 21.33)	**14.6** ($\downarrow$ 14.0)
SecurityEval	Generated code	34.7	38.0	46.2
	Direct prompting	21.4 ($\downarrow$ 13.2)	25.6 ($\downarrow$ 12.4)	38.0 ($\downarrow$ 8.2)
	Self-debugging	16.5 ($\downarrow$ 18.1)	27.2 ($\downarrow$ 10.7)	38.8 ($\downarrow$ 7.4)
	Bandit feedback	**4.1** ($\downarrow$ 30.5)	13.2 ($\downarrow$ 24.7)	21.4 ($\downarrow$ 24.7)
	Verbalization	4.9 ($\downarrow$ 29.7)	13.22 ($\downarrow$ 24.7)	17.3 ($\downarrow$ 28.92)
	FDSP	**4.1** ($\downarrow$ 30.5)	**5.7** ($\downarrow$ 32.2)	**8.2** ($\downarrow$ 38.0)
PythonSecurityEval	Generated code	40.21	48.51	42.34
	Direct prompting	25.1 ($\downarrow$ 15.1)	42.5 ($\downarrow$ 5.9)	31.7 ($\downarrow$ 10.6)
	Self-debugging	24.8 ($\downarrow$ 15.3)	43.4 ($\downarrow$ 5.1)	33.4 ($\downarrow$ 8.9)
	Bandit feedback	9.3 ($\downarrow$ 30.8)	26.3 ($\downarrow$ 22.1)	20.4 ($\downarrow$ 21.9)
	Verbalization	8.7 ($\downarrow$ 31.4)	23.6 ($\downarrow$ 24.8)	21.0 ($\downarrow$ 21.2)
	FDSP	**7.4** ($\downarrow$ 32.7)	**15.7** ($\downarrow$ 32.7)	**13.6** ($\downarrow$ 28.7)

Table 1: The table illustrates the percentage of vulnerable code, with the number in parentheses representing the percentage of vulnerable code that was fixed relative to the percentage of generated vulnerable code.

Bandit's feedback or verbalizing it. We evaluate the effectiveness of each method in addressing the most common security issues in CodeLlama, as depicted in Figure 5. These results suggest that self-refinement or directly passing the feedback from static code analysis is not very useful for CodeLlama; however, the approach of verbalizing the feedback from static code analysis or FDSP performs well for CodeLlama.

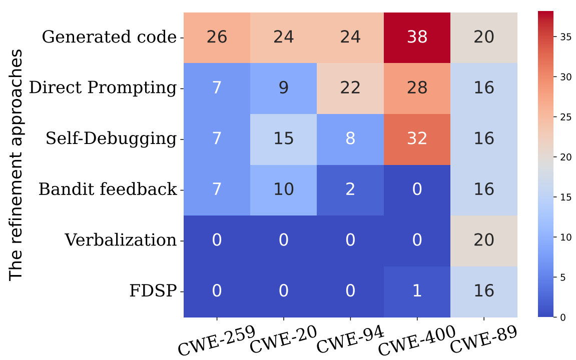

Figure 5: The figure illustrates the total number of the five most common types of security issues for CodeLLama across five refinement approaches for the PythonSecurityEval dataset.

We analyze the most common vulnerabilities in generated code, as well as those that remain unresolved, for the PythonSecurityEval dataset. Figure 3 illustrates the most common types of code vulnerabilities generated by three LLMs, with the top two being CWE-259 (Use of Hard-coded Password) and CWE-400 (Uncontrolled Resource Consumption). However, the LLMs are able to fix most of these types of vulnerabilities, as shown in Figure 4. We visualize the most frequent unresolved security issues by the same three models in Figure 4, where the top two are related to injection: CWE-78 (OS Command Injection) and CWE-89 (SQL Injection), with percentage of %61.1 and %80.0 respectively for GPT-4. Additionally, these injection vulnerabilities are also among the most frequent vulnerabilities generated by LLMs.

The refinement process aimed at fixing vulnerable code may alter the code's functionality or produce unrealistic code. We manually review each piece of refined code and compare it with the originally generated vulnerable code. If the refined code is unrealistic, we classify it as unfixed when reporting our results in Table 1. In the PythonSecurityEval dataset, we observed that GPT-4 generated 9 instances of unrealistic code among all approaches, with 3 of these instances arising from the FDSP approach. GPT-3.5 produced approximately 11 instances of unrealistic code, 10 of which were associated with the FDSP approach. CodeLlama had the highest number of unrealistic instances, totaling 28, with 13 of these resulting from the FDSP approach. This is due to the FDSP method generating three multiple solutions from LLMs, some of which may be unrealistic. Our approach demonstrates that GPT-4 can generate potentially more effective solutions than both CodeLlama and GPT-3.5.

To better understand the effectiveness of FDSP in outperforming the baseline, we conducted an evaluation experiment using an external vulnerability detection tool other than Bandit. We used CodeQL[3], an open-source codebase utilized to discover the similarity of vulnerability patterns in code. In Table 2, we report the percentage of vulnerable code in the PythonSecurityEval dataset across the three LLMs and the refinement approaches. The FDSP outperformed the baseline across the LLMs.

	GPT4	**GPT3.5**	**CodeLlama**
Generated code	31.4	20.8	25.5
Direct prompting	18.5	13.6	12.1
Self Debug	16.1	13.8	15.9
Bandit feedback	13.8	9.3	10.4
Verbalization	12.9	10.6	11.9
FDSP	**11.9**	**9.1**	**8.2**

Table 2: The table illustrates the percentage of vulnerable code in PythonSecurityEval dataset using CodeQL.

6 Conclusion

As LLMs are capable of generating code, we assess the frequency with which they produce vulnerable code. We conduct an empirical evaluation of LLMs using existing datasets and approaches, and discuss their limitations. We propose a novel approach and introduce a new dataset to address the current limitations of both datasets and approaches. Our approach is called FDSP, where the LLMs generate potential solutions to fix their own generated vulnerable code by leveraging feedback from static code analysis. Our results demonstrate that the FDSP approach outperforms the baselines across all three benchmarks and three models, achieving an improvement of about 17.6% in the PythonSecurityEval dataset for GPT-4.

7 Limitations

One of the limitations of our study is that our evaluation may not identify all security issues in the code. Detecting vulnerabilities in code is an active area of research, where recent studies have shown that some vulnerabilities cannot be detected by either static code analysis or deep learning-based vulnerability detection methods (Chakraborty et al., 2021). Additionally, we evaluate LLMs in generating and refining vulnerable code at the function level; however, in real-life scenarios, vulnerabilities often occur when code interacts with other code or services. Lastly, our approach to fixing security issues involves making changes to the code, which might inadvertently render the program incorrect. It is very challenging to ensure the correctness of the code or to write unit tests, as all the generated code interacts with other services or systems such as databases or operating systems.

[3] https://codeql.github.com/

Acknowledgments

References

Afra Feyza Akyürek, Ekin Akyürek, Ashwin Kalyan, Peter Clark, Derry Wijaya, and Niket Tandon. 2023. Rl4f: Generating natural language feedback with reinforcement learning for repairing model outputs. In *Annual Meeting of the Association of Computational Linguistics 2023*, pages 7716–7733. Association for Computational Linguistics (ACL).

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876.

Kamel Alrashedy, Vincent J. Hellendoorn, and Alessandro Orso. 2023. Learning defect prediction from unrealistic data. *arXiv preprint arXiv:2311.00931*.

Ben Athiwaratkun, Sanjay Krishna Gouda, and Zijian Wang. 2023. Multi-lingual evaluation of code generation models. *The International Conference on Learning Representations (ICLR)*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Nikolaos Bafatakis, Niels Boecker, Wenjie Boon, Martin Cabello Salazar, Jens Krinke, Gazi Oznacar, and Robert White. 2019. Python coding style compliance on stack overflow. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 210–214. IEEE.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam Girish Sastry, Amanda Askell, Sandhini Agarwa, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Clemens Winter Jeffrey Wu, Christopher Hesse, Mark Chen, Eric

Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug.

Ahmed Elgohary, Christopher Meek, Matthew Richardson, Adam Fourney, Gonzalo Ramos, and Ahmed Hassan Awadallah. 2021. NL-EDIT: Correcting semantic parse errors through natural language interaction. In *Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. 2023. Critic: Large language models can self-correct with tool-interactive critiquing. *Preprint*, arXiv:2305.11738.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback.

Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 881–881. IEEE Computer Society.

Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation? In *The Twelfth International Conference on Learning Representations*.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models.

Baolin Peng, Michel Galley, Pengcheng He, Hao Cheng, Yujia Xie, Yu Hu, Qiuyuan Huang, Lars Liden, Zhou Yu, Weizhu Chen, et al. 2023. Check your facts and try again: Improving large language models with external knowledge and automated feedback. *arXiv preprint arXiv:2302.12813*.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in neural information processing systems*, 33:20601–20611.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*.

Mohammed Siddiq and Joanna Santos. 2022. 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)*.

Mohammed Latif Siddiq, Beatrice Casey, and Joanna C. S. Santos. 2023. A lightweight framework for high-quality code generation.

Catherine Tony, Markus Mutas, Nicolas Díaz Ferreyra, and Riccardo Scandariato. 2023. Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*.

Man-Fai Wong, Shangxin Guo, Ching-Nam Hang, Siu-Wai Ho, and Chee-Wei Tan. 2023. Natural language generation and understanding of big code for ai-assisted programming: A review. *Entropy*, 25(6):888.

Kevin Yang, Yuandong Tian, Nanyun Peng, and Dan Klein. 2022. Re3: Generating longer stories with recursive reprompting and revision. *n Conference on Empirical Methods in Natural Language Processing*.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. In *International Conference on Machine Learning*, pages 10799–10808. PMLR.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Kamal Ndousse Yuntao Bai, Andy Jones. 2023. Training a helpful and harmless assistant with reinforcement learning from human feedback.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. In *International Conference on Learning Representations (ICLR)*, Kigali, Rwanda.

A Example Appendix

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's 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 6: An example of how GPT-4 verbalizes the feedback from the Bandit shown in the Figure 2.