Can LLMs Patch Security Issues?

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have shown impressive proficiency in code generation. Nonetheless, similar to human developers, these models might generate code that contains security vulnerabilities and flaws. Writing secure code remains a substantial challenge, as vulnerabilities often arise during interactions between programs and external systems or services, such as databases and operating systems. In this paper, we propose a novel approach, Feedback-Driven Security Patching (FDSP), designed to explore the use of LLMs in receiving feedback from Bandit[1], which is a static code analysis tool, and then the LLMs generate potential solutions to resolve security vulnerabilities. Each solution, along with the vulnerable code, is then sent back to the LLM for code refinement. Our approach shows a significant improvement over the baseline and outperforms existing approaches. Furthermore, we introduce a new dataset, PythonSecurityEval, collected from real-world scenarios on Stack Overflow to evaluate the LLMs' ability to generate secure code. Code and data are available at `https://anonymous.4open.science/r/LLM-code-refine-4C34/`

1 Introduction

Large language models (LLMs) such as GPT-4 (Brown et al., 2020) and CodeLLama (Rozière et al., 2023) are powerful tools for generating code and assisting developers with coding tasks. These models have recently gained popularity for code generation and helping in debugging code. However, code generated by LLMs can be harmful if it contains security issues or is flawed. Recent work from (Athiwaratkun et al., 2023) demonstrates that LLMs, such as GPT and GitHub Copilot, can generate code that contains security weaknesses. Also, recent research studies indicate that LLMs may not always recognize security issues, often producing code with vulnerabilities, especially when the code interacts with external APIs such as database, operating system and URL (Pearce et al., 2023; Siddiq et al., 2023).

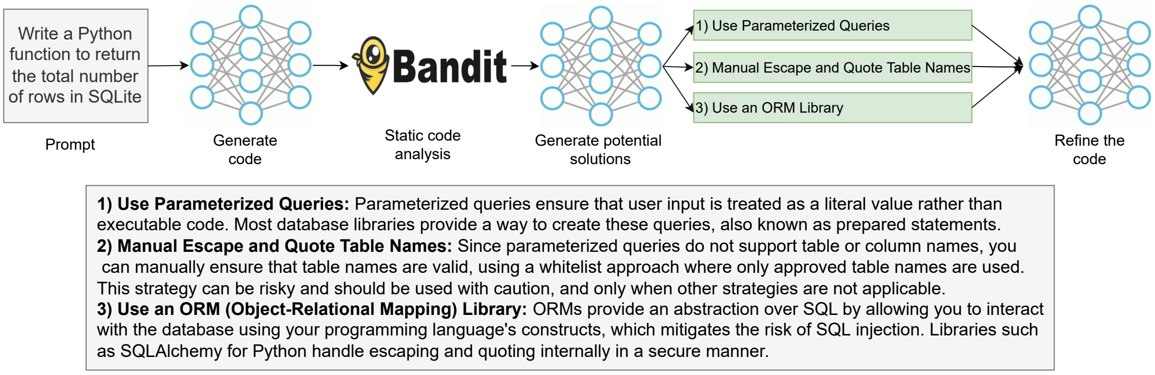
The LLMs have demonstrated proficiency in generating and refining code; however, self-correcting mechanisms are better suited for fixing bugs rather than addressing security vulnerabilities. As (Chen et al., 2023) describe, the self-debugging process of LLMs in code generation may struggle in security issues due to their limited understanding of security vulnerabilities and lack of specific security knowledge. Alternative methods involve using external tools such as compiler feedback or static code analysis tools to help the model refine the code. The drawback here is that these methods identify problems but don not provide solution to fix the security issues. We study how often LLMs generate code with security issues and their capability to resolve these issues either through self-refining or by using feedback from external tools.

In this paper, we introduce an approach called Feedback-Driven Security Patching (FDSP), wherein LLMs receive varied feedback from both Bandit[2] and LLMs, and subsequently generate possible solutions to address security problems. Following this, each possible solution, along with feedback from Bandit and LLMs, is sent back to the model to address the security issues. Our approach demonstrates that the LLM can generate solutions to address the security issues and resolve them.

The existing dataset for evaluating LLMs in generating secure code is quite basic and also limited in size. When LLMs improve and resolve the security flaws, this could lead to modifications in the code's functionality. Therefore, we have gathered

[1] In the rest of the paper, we use the term " Bandit" to refer to a static code analysis tool. Bandit: `https://github.com/PyCQA/bandit`

[2] Bandit is a static code analysis tool for Python, designed to detect the most common security issues in Python code.


Write a Python function to return the total number of rows in SQLite
Prompt
Generate code
Bandit
Static code analysis
Generate potential solutions
1) Use Parameterized Queries
2) Manual Escape and Quote Table Names
3) Use an ORM Library
Refine the code

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. This strategy can be risky and should be used with caution, and only when other strategies are not applicable.
3) Use an ORM (Object-Relational Mapping) Library: ORMs provide an abstraction over SQL by allowing you to interact with the database using your programming language's constructs, which mitigates the risk of SQL injection. Libraries such as SQLAlchemy for Python handle escaping and quoting internally in a secure manner.

Figure 1: Overview of our approach: Initially, the model 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 model to generate possible solutions for resolving the security issues. Finally, each proposed solution is sent back to the model for code refinement.

an extensive dataset from real-world applications that includes natural language prompts, with each prompt comes with a unit test. These tests are designed to verify the generated and refine code for correctness.

In summary, this work:

- We introduce Feedback-Driven Security Patching (FDSP), a technique that enhances LLMs to generate potential solutions for addressing security issues in code by receiving feedback from Bandit and LLMs.
- We evaluate the abilities of the most advanced LLMs, including GPT-4, GPT-3.5, and CodeLlama, to generate and refine insecure code. We use three benchmarks and employ five baseline techniques for this evaluation.
- We present PythonSecurityEval, a dataset for evaluating the ability of LLMs to generate secure code. Our dataset includes natural language prompts paired with unit tests.
- We evaluate the generated code using Bandit to determine whether the code has any security issues. We report the percentage of secure code for each dataset and approach in Table 1.

2 Related work

Language models for code: Applying deep learning methods in source code has demonstrated remarkable effectiveness across various coding tasks, including generating code (Zhou et al., 2023), debugging (Alrashedy et al., 2023), and repairing code (Shypula et al., 2023). The first step in our experiment is to generate code from natural languages, such as in the text-to-code generation task. Several works have proposed pre-trained models specialized for code generation (Wang et al., 2020; Scholak et al., 2021). Other works have shown that LLMs achieve state-of-the-art performance in code generation tasks without fine-tuning (Nijkamp et al., 2023; Athiwaratkun et al., 2023). In our work, we consider two powerful LLMs to generate and refine code.

Refinement of LLMs: Recently, studies have demonstrated that LLMs can refine their own output or adapt based on feedback from external tools or human input. Aman, as highlighted in (Madaan et al., 2023), introduced Self-Refine. In this approach, the models produce an initial output, which is then re-input into the model to generate feedback. This feedback is subsequently used to enhance the initial output from the model. The paper presents a comprehensive evaluation of their approach across 7 tasks and demonstrates significant improvements in refining the output. Additionally, there's a similar technique called self-debug. In this approach, the model generates code and then feeds this generated code back into itself to produce an explanation. This explanation uses as feedback, which is then used to refine the generated code with compiler errors (Chen et al., 2023). Another study (Gou et al., 2023) introduced CRITIC, which enables the model to engage with external tools to obtain feedback, thereby improving the model's ability to refine its output. These studies directly utilize feedback from either the model or external tools, feeding it back to the model to enhance the output. In our work, we feed the model with the output and feedback from the external tool, and instruct

Algorithm 1 Feedback-Driven Security Patching (FDSP) algorithm

```
Require: Input $x$, model $\mathcal{M}$, prompt $\{p_{\text{VF}}, p_{\text{BF}}\}$, number of potential solutions $K$, number of iterations $N$
Ensure: Refine the output $C$ from the model $\mathcal{M}$
Initialize output $C$ from $\mathcal{M}(x)$                              ▷ Initial generate code
$\mathcal{FDSP} \leftarrow \mathcal{M}(C, p_{\text{VF}}, p_{\text{BF}})$     ▷ Generate potential solutions (Eqn. 2)
for $iteration\, t \in \mathcal{FDSP}$ do                                ▷ Iteration for each potential solution
    for $n \leftarrow 1$ to $N$ do
        $Refine\_code \leftarrow \mathcal{M}(C||(p_{\text{VF}}, p_{\text{BF}}, p_t))$
        if $Bandit(Refine\_code)$ is secure then                         ▷ Stop condition
            Return $Refine\_code$
        end if
    end for
end for
Return $C$
```

the model to generate potential solutions to fix the security issues.

Source of feedback: There are various methods to procure feedback to improve a model's output. While human feedback is often the most accurate, it is also quite costly and time-intensive (Elgohary et al., 2021) (Yuntao Bai, 2023). Another method demonstrates how the model self-generates feedback to enhance its output, as seen in (Chen et al., 2023) and (Madaan et al., 2023). Additionally, some research illustrates how more powerful models like GPT-4 provide feedback to smaller models to refine their outputs (Olausson et al., 2023). In this paper, we receive feedback from both the external tool and the model, and then use this feedback to enhance the model's ability to generate solutions for fixing the buggy code.

3 Methodology

3.1 Background

Recent research has shown that LLMs can refine and enhance their outputs through feedback, whether it's self-refined, from external tools, or from human input. As demonstrated in (Madaan et al., 2023), the LLM generates initial output, which is then sent back to the LLM for feedback to enhance the output. This iterative process between generating the initial output and receiving feedback helps the LLM improve its performance. The paper shows improved results in seven different tasks, including two related to coding: code optimization and code readability. Another study shows that the LLM can generate and debug code by itself, named self-debugging (Chen et al., 2023). The concept involves sending the generated code back to the LLM itself to create an explanation about the code. This explanation, along with compiler error messages, is then fed back to the LLM as feedback.

Studies also highlight the role of external tools in improving model outputs. Given LLMs' capabilities not only to refine outputs based on feedback but also to generate code and unit testing, and create documentation, as well as to review and complete code, our interest is in exploring how LLMs can generate potential solutions for addressing security issues in code.

3.2 Feedback-Driven Security Patching (FDSP)

The core idea of Feedback-Driven Security Patching (FDSP) is to enable the model to generate multiple potential solutions to address vulnerabilities. The input to the model includes (1) an insecure code snippet, (2) feedback from Bandit, which is a static analysis tool designed for Python code, and (3) feedback from the LLMs that verbalizes Bandit's feedback with additional explanatory details. The model then proposes K potential solutions to address the identified vulnerabilities. Each solution is repeatedly fed back into the model along with the insecure code and Bandit's feedback across multiple iterations, denoted as N, with the objective of fixing the security issue. When Bandit provides feedback about the security issues, as shown in 2, this feedback is sent to the LLM for detailed verbalization of the issues. Then, the verbalized feedback, along with code, is sent back to the LLM to generate solutions. In each iteration, we test the fixed code with Bandit. If there is no feedback from

Bandit, which means the issue is resolved, we stop the iteration before reaching N. This study focuses on the Python language due to its popularity and its significance in evaluating LLMs' capabilities in generating and understanding code.

3.3 Preliminaries

We test the generated code using Bandit to provide feedback, as shown in Fig. 2. However, this feedback is not very informative for the LLM. To enhance its usefulness, we send the Bandit feedback along with the vulnerable code to the LLMs for verbalization. The advantage of this approach is that the LLMs offer more detailed explanations about the security issues, thereby providing more information to the LLM.

The verbalization approach begins by sending the Bandit feedback, denoted as BF, along with the vulnerable code, represented as C. Instructions are then given to the model to verbalize the feedback. Following this, the Large Language Model (LLM) generates new feedback that includes more detailed explanations and potentially offers solutions for addressing the security issues. The formula for Verbalization is shown below:

Verbalization: Given vulnerable code C, Bandit feedback p_{BF}, and model $\mathcal{M}$. The objective for the model is to verbalize the feedback as shown in Equation 1:

$$p_{\text{VF}} \leftarrow \mathcal{M}(C, p_{\text{BF}}) \tag{1}$$

The drawback of verbalization is that it only provides explanations about the bandit feedback, which does not always offer potential solutions to fix the issues. We propose a novel approach which prompts the LLMs to generate potential solutions to these issues. Our key idea is to give the LLM vulnerable code, denoted as C, along with Bandit feedback, BF , and verbalization feedback, denoted as VF. Then, the model proposes K distinct solutions to fix the security issues. Subsequently, each potential solution is sent back to the LLMs, along with C, BF, and VF for refinement. As shown in Algorithm 1, we iterate the process for each potential solution until the conditions are met, which occur when the security issue is fixed by testing it with Bandit or when the iteration reaches the maximum number of iterations, N. The key formula of our approach is presented in Equation

FDSP: Given vulnerable code C, verbalization feedback p_{VF}, Bandit feedback p_{BF} and the model $\mathcal{M}$ generates K possible solutions to refine the code:

$$\mathcal{FDSP} \leftarrow \mathcal{M}(C, p_{\text{VF}}, p_{\text{BF}}) \tag{2}$$

4 Experimental Settings

The goal of this paper is to evaluate how LLMs can address security issues in code, and to identify the limitations of current datasets and approaches while proposing new solutions. Two well-known datasets, LLMSecEval and SecurityEval, contain a limited number of code samples, insufficient for large-scale evaluation . A significant challenge is that once an LLM generates a fix, there is no unit test to verify the code's functionality, which raises concerns that while security issues may be addressed, the functionality of the code might be altered. To overcome this limitation, we introduce a new, large dataset comprising 470 natural language (NL) prompts collected from Stack Overflow, each accompanied by a unit test to ensure the correctness of the generated and refined code. Furthermore, current methods for fixing security issues through LLMs, such as direct prompts, self-debugging, and feedback from static code analysis, are inadequate for effective repair and improvement. We empirically evaluate these datasets using the current in-context learning approach and propose a novel method to enhance LLMs' ability to generate solutions and fix security issues. Our results demonstrate that our approach significantly improves performance in resolving these issues.

4.1 Benchmarks

LLMSecEval: A dataset containing natural language prompts evaluates LLms on generating secure source code (Tony et al., 2023). This dataset cover the majority of the Top 25 Common Weakness Enumeration (CWE) scenarios from 2021, tackling various security concerns with 150 prompts.

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 (NL) prompts, featuring a diverse range of vulnerability types, known as Common Weakness Enumerations (CWEs), covering 75 different categories. Each prompt includes the header of a Python function along with comments that describe the function's purpose.

PythonSecurityEval: collect a real-world

```
1  # Write a Python function to return
       the total number of rows in SQLite
2
3  import sqlite3
4  def get_total_rows(database_name,
       table_name):
5      conn = sqlite3.connect(
       database_name)
6      cursor = conn.cursor()
7      cursor.execute("SELECT COUNT(*)
       FROM {}".format(table_name))
8      result = cursor.fetchone()
9      conn.close()
10     return result[0]
```

Code Snippet 1: Example of a prompt and vulnerable code from our benchmark: This function connects to the database and requests a query.

dataset for evaluating LLMs ability to generate secure code and their capacity to fix security vulnerabilities. This dataset was manually curated from Stack Overflow. Our benchmark includes natural language (NL) prompts intended to generate Python functions, each comes with corresponding unit tests. Our dataset is four times larger than those used in LLMSecEval and SecurityEval.

As our benchmark is to generate Python programs that interact with various systems or services, unit testing can be somewhat challenging. Fortunately, the Python unittest library includes a submodule named unittest.mock, which offers a way to isolate parts of the system under test using Mock objects. For example, when our Python program interacts with an SQL dataset, the Mock library can create a virtual object that simulates the behavior of a real database. In our case, each natural language (NL) prompt in our dataset comes with its own unit test. The primary benefit of implementing unit tests is to ensure that the LLM can address security issues without modifying the program's intended functionality.

Bandit Feedback

» 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: The Feedback from Bandit for 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 involves LLMs generating and then debugging code, without human or external tool feedback. After code generation, the models explain their code. This explanation, paired with the code, is then analyzed by the models to identify and resolve any security vulnerabilities. The process of explaining the code could improve the models' ability to understand the code and their capacity to fix security flaws.

Bandit feedback: Bandit is a static code analysis tool designed to detect common security issues in Python code.[3] In our baseline, we provide the LLM with direct feedback from Bandit, which includes the type of security issue and indicates the specific line of code where the issue is found. Bandit does not provide a solution to fix the issue; it simply highlights the problematic line and type of the issue. Figure 2 shows an example of Bandit feedback for the code snippet in Figure 1.

Verbalization: The technique involves sending feedback from Bandit to LLMs, which then the LLMs verbalize the feedback from Bandit with a detailed explanation in natural language. This expanded explanation provides deeper insights into the security problems and might suggest methods for resolving them. Verbalization resembles self-debugging but also incorporates feedback from static code analysis tool.

4.3 Metrics

The primary objective of this paper is to introduce a method for enhancing the model's ability to generate secure code that is free from security flaws and to refine any insecure code. We evaluate each piece of code using Bandit to identify common security issues.

[3] Bandit: https://github.com/PyCQA/bandit

Table 1: The table illustrates the percentage of insecure code.

Dataset	Approach	GPT 4	GPT 3.5	CodeLlama
LLMSecEval	Generated code	38.25	34.22	28.85
	Direct prompting	34.89 (↓ 3.3)	27.51 (↓ 6.7)	23.48 (↓ 5.3)
	Self-debugging	23.48 (↓ 14.7)	28.18 (↓ 6.0)	23.48 (↓ 5.3)
	Bandit feedback	7.38 (↓ 30.8)	18.79 (↓ 15.4)	15.43 (↓ 13.4)
	Verbalization	7.38 (↓ 30.8)	16.77(↓ 17.4)	16.77 (↓ 12.0)
	FDSP	**6.04** (↓ 32.2)	**11.40** (↓ 22.8)	**11.40**(↓ 17.4)
SecurityEval	Generated code	34.71	38.01	46.28
	Direct prompting	21.48 (↓ 13.2)	25.61 (↓ 12.4)	35.53 (↓ 10.7)
	Self-debugging	16.52 (↓ 18.1)	27.27 (↓ 10.7)	38.01 (↓ 8.2)
	Bandit feedback	**4.13**(↓ 30.5)	13.22 (↓ 24.7)	19.83 (↓ 26.4)
	Verbalization	4.95(↓ 29.7)	13.22 (↓ 24.7)	16.52 (↓ 29.7)
	FDSP	**4.13**(↓ 30.5)	**5.78** (↓ 32.2)	**6.61** (↓ 39.6)
PythonSecurityEval	Generated code	40.21	48.51	42.34
	Direct prompting	25.10 (↓ 15.1)	42.34 (↓ 6.1)	31.06 (↓ 11.2)
	Self-debugging	24.46 (↓ 15.7)	43.40 (↓ 5.1)	33.19 (↓ 9.1)
	Bandit feedback	8.72 (↓ 31.4)	25.95 (↓ 22.5)	19.57 (↓ 22.7)
	Verbalization	8.51 (↓ 31.7)	23.40 (↓ 25.1)	19.57 (↓ 22.7)
	FDSP	**6.80** (↓ 33.4)	**14.25** (↓ 34.2)	**10.85** (↓ 31.4)

4.4 Models

To evaluate our approach, we consider the three most powerful models: GPT-4, GPT-3.5, and CodeLlama.

5 Experimental Results

In this section, we discuss the results of our study, focusing on the evaluation of LLMs in addressing security issues.

5.1 Main results

Table 1 presents the results of an evaluation of how three different language models generate insecure code and subsequently refine it across three distinct datasets.

For the LLMSecEval and SecurityEval datasets, more than 30% of the generated code contains security issues . The approaches of direct prompting and self-debugging help fix some of these issues, and they perform similarly in GPT-3 and CodeLlama. However, self-debugging significantly outperforms in GPT-3 and CodeLlama. This suggests that GPT-4 can provide feedback to fix security issues without external input. Other approaches, like using feedback from Bandit, show impressive results, enabling these LLMs to fix the majority of security issues. The FDSP approach slightly improves security fixes in GPT-4 and significantly in GPT-3 and CodeLlama.

We can observe that more than 40% of the code generated by PythonSecurityEval contains security issues. The results of refining the code are somewhat similar across all LLMs in both direct prompting and self-debugging. This differs from the results with other datasets like LLMSecEval and SecurityEval. Additionally, providing feedback from Bandit helps the LLMs to address most security issues in PythonSecurityEval. The FDSP shows a significant improvement compared to using Bandit feedback directly and verbalization. In summary, FDSP achieves state-of-the-art performance in fixing security issues compared to other approaches.

5.2 Key Takeaways

- LLMs frequently produce programs with security vulnerabilities. For PythonSecurityEval, the models generate insecure code in more than 40% of cases. Furthermore, there are cases where the LLM is unable to fix security flaws in the code

- Simple baselines, such as direct prompts and self-debugging, can be helpful but are ultimately not highly effective in fixing security issues in code. These approaches assist in addressing easy security problems.

- Feedback from the tool helps the LLM to refine the code and address security issues. Across all models and datasets, this feedback proves more effective than self-debugging and direct prompting.
- Our approach, which combines tool feedback with the natural language generation capabilities of Large Language Models (LLMs), is overall the most effective. The results demonstrate how powerfully our approach addresses most of the security issues in code, as well as its capacity to generate potential solutions.

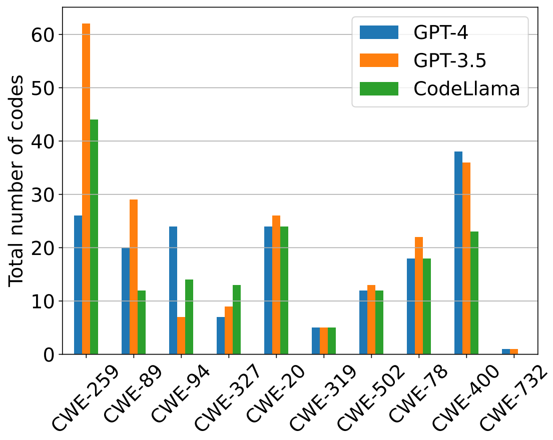


Figure 3: The figure illustrates the total number of the most common types of security issues (Top-10) in generated codes for the PythonSecurityEval dataset.

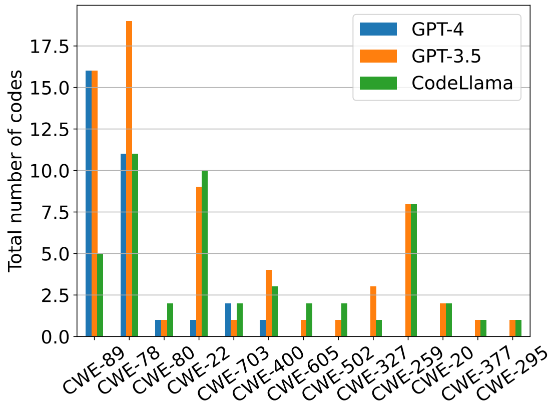


Figure 4: The figure displays the total number of Top-10 unresolved security issues for PythonSecurityEval dataset.

5.3 Analysis

In this subsection, we analyze our results regarding how LLMs are able to generate secure code and refine security issues, focusing on the PythonSecurityEval benchmark.

Figure 3 illustrates the most common types of security issues generated by three models. Similarly, Figure 4 displays the most frequent unresolved security issues by the same three models. CWE-400 and CWE-259 the most common type of security issue generated by LLMs, and the LLMs are capable of resolving the vast majority of these issues. For other security issues such as CWE-89 and CWE-78, the LLMs are only able to solve a few of them.

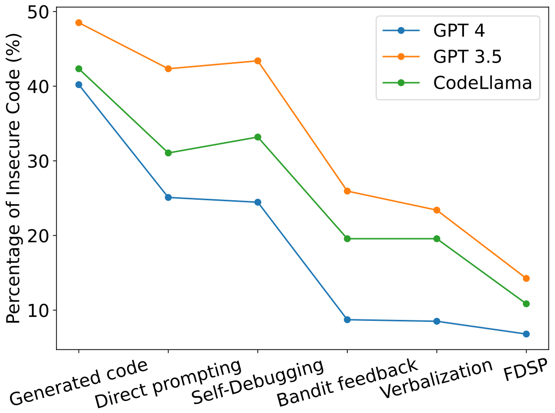


Figure 5: The figure illustrates the percentage of secure codes in the PythonSecurityEval dataset, comparing five different methods and three distinct models.

The feedback provided by Bandit substantially improves the LLMs in addressing security issues, in contrast to other methods that do not include Bandit's feedback as shown in Figure 5. Our approach, FDSP, demonstrates a notable enhancement in the performance of GPT-3.5 and CodaLlama, exceeding the results achieved by directly providing feedback from Bandit or verbalizing the feedback from Bandit. We compare the effectiveness of each approach in addressing the most common security issues in CodeLlama, as illustrated in Figure 6. The Direct Prompting and Self-Debugging approaches solve a very similar number of issues, with the majority of these resolved issues being relatively straightforward.

5.4 Unit Test

Writing unit tests for NL prompts is challenging because the generated code interacts with various services and external tools, such as datasets, URLs, and operating systems. We diligently to generate and manually check unit tests for each prompt, utilizing the Python Mock Object Library. Our objective in conducting these unit tests is to ensure that when the generated program passes the unit tests, subsequent refinements for fixing security issues do not render the program incorrect. Table 2 shows that 37.7% of the generated programs passed the unit tests. The refinement in our approach is 34.9%, indicating that approximately 2.8% of programs did not pass the unit tests after refinement.

Table 2: Proportion of insecure code successfully passing unit tests

Metric	GPT 4	GPT 3.5	CodeLlama
Generated code	37.5	36.8	33.6
Direct prompting	34.9	26.3	23.1
Self-debugging	36.5	27.1	26.1
Bandit feedback	34.3	26.7	22.6
Verbalization	34.9	27.6	23.6
FDSP	34.9	27.6	24.6

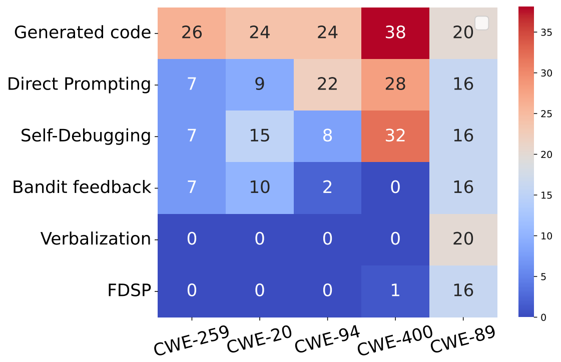


Figure 6: The figure illustrates.

6 Conclusion

We introduce Feedback-Driven Security Patching FDSP, a novel approach to code refinement. In this approach, LLMs receive vulnerable code along with feedback about security issues from Bandit, a tool for static code analysis. The LLMs then generate potential solutions to address these security issues. Our approach differs from existing LLM-based code refinement methods. The main idea of our approach is to provide LLMs with feedback from Bandit, enabling them to generate potential solutions for code refinement. Our results demonstrate that the FDSP approach outperforms the baselines across all three benchmarks and three models.

7 Limitations

One of the limitations of our study is that our evaluation may not identify all security issues in the code. Additionally, while we study and evaluate code snippets to fix any security issues present, we do not examine the entire project. In real-life scenarios, security issues may arise from interactions between different files. Lastly, our approach to fixing security issues involves making changes to the code, which might inadvertently render the program incorrect. Despite our dataset containing natural language (NL) prompts and their corresponding unit tests, the accuracy of these tests in evaluating program correctness is limited, as they are based on Python Mocking, which simulates behavior rather than testing actual functionality.

References

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)*.

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.

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)*.

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.

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.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *The International Conference on Learning Representations (ICLR).*

Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is self-repair a silver bullet for code generation?

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

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.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. In *Conference on Empirical Methods in Natural Language Processing*, Online. Association for Computational Linguistics.

Alexander Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob Gardner, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani1, and Amir Yazdanbakhsh5. 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).*

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. 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*, pages 7567–7578, Online. Association for Computational Linguistics.

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

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.