Rethinking the Evaluation of Secure Code Generation

Shih-Chieh Dai
shihchieh.dai@utah.edu
University of Utah
Salt Lake City, Utah, USA

Jun Xu
junxzm@cs.utah.edu
University of Utah
Salt Lake City, Utah, USA

Guanhong Tao
g.tao@utah.edu
University of Utah
Salt Lake City, Utah, USA

Abstract

Large language models (LLMs) are widely used in software development. However, the code generated by LLMs often contains vulnerabilities. Several secure code generation methods have been proposed to address this issue, but their current evaluation schemes leave several concerns unaddressed. Specifically, most existing studies evaluate security and functional correctness separately, using different datasets. That is, they assess vulnerabilities using security-related code datasets while validating functionality with general code datasets. In addition, prior research primarily relies on a single static analyzer, CodeQL, to detect vulnerabilities in generated code, which limits the scope of security evaluation.

In this work, we conduct a comprehensive study to systematically assess the improvements introduced by four state-of-the-art secure code generation techniques. Specifically, we apply both security inspection and functionality validation to the same generated code and evaluate these two aspects together. We also employ three popular static analyzers and two LLMs to identify potential vulnerabilities in the generated code. Our study reveals that existing techniques often compromise the functionality of generated code to enhance security. Their overall performance remains limited when evaluating security and functionality together. In fact, many techniques even degrade the performance of the base LLM by more than 50%. Our further inspection reveals that these techniques often either remove vulnerable lines of code entirely or generate "garbage code" that is unrelated to the intended task. Moreover, the commonly used static analyzer CodeQL fails to detect several vulnerabilities, further obscuring the actual security improvements achieved by existing techniques. Our study serves as a guideline for a more rigorous and comprehensive evaluation of secure code generation performance in future work.

CCS Concepts

• **Computing methodologies** → **Machine learning**; • **Security and privacy** → **Software and application security**.

Keywords

Secure Code Generation, Large Language Model

ACM Reference Format:
Shih-Chieh Dai, Jun Xu, and Guanhong Tao. 2026. Rethinking the Evaluation of Secure Code Generation. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773217

This work is licensed under a Creative Commons Attribution 4.0 International License.
ICSE '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773217

1 Introduction

Software development is a time-consuming and repetitive task. The rapid evolution of large language models (LLMs) has greatly benefited software developers. Given task requirements described in natural language, LLMs can generate functional and easy-to-adopt code snippets, significantly accelerating the software development process. Many LLM-based code assistants are already integrated into IDEs, such as Copilot [2] and Cursor [3].

However, just like human developers, LLMs can also make mistakes when producing code. One of the major concerns regarding LLM-generated code is its security. As code snippets generated by LLMs are increasingly incorporated into industrial-level software and systems, it is critical to ensure that LLM-generated code is free of vulnerabilities that could be exploited by attackers.

To address this, a number of techniques have been proposed to improve the security of LLM-generated code, called *secure code generation* methods. These techniques typically involve either adjusting LLM weight parameters or manipulating the input to or output from the LLM. For example, SVEN [23] and SafeCoder [24] construct a training dataset containing code snippets with and without vulnerabilities. They use such a dataset to fine-tune the model such that it can generate vulnerability-free code once trained. CodeGuard+ [21] instead controls the generation process of LLM inference. As LLMs use a decoding algorithm to determine the output, CodeGuard+ modifies this algorithm to favor outputs that lead to secure code. Another state-of-the-art technique, PromSec [41], iteratively refines the task prompt based on the feedback from a vulnerability scanner on the generated code.

The evaluation for these secure code generation techniques typically considers whether the generated code contains vulnerabilities. A common practice is to use a vulnerability scanner, such as CodeQL [7], to assess the security. These techniques also measure their impact on normal model utility, that is, whether the enhanced LLM can still generate functional and usable code. A code benchmark like HumanEval [15] is usually adopted for the evaluation.

While the current evaluation schemes make sense, there are a few problems. First, existing works mainly *rely on the reported results by a single vulnerability detector, CodeQL, for assessing security*, following the security evaluation practice proposed in prior work [45]. This scanner is not the gold standard as it can miss or misflag certain vulnerabilities. Our experiments in Section 4.1 show that CodeQL can miss more than 20% vulnerabilities in generated code. Second, the current evaluation scheme *examines the security and functional correctness of generated code independently*. In other words, it uses a security-related code dataset to measure the security (e.g., [53]) and use another general code dataset to assess functionality (e.g., [44]). Such an evaluation scheme leaves a gap of reported results between the two aspects. *Are the generated code*

from the functionality dataset secure? Are the generated code from the security dataset functionally correct?

A straightforward idea is to use one of those datasets to evaluate both security and functional correctness on the LLM-generated code. However, there are a few issues with the datasets employed in previous works. Most security-related code datasets do not have unit tests [13, 53, 56], meaning it is not feasible to rigorously evaluate the functionality of the generated code. The functionality benchmarks, such as HumanEval [15] and MBPP [12], on the other hand, contain relatively simple tasks. They do not have the complexity of triggering security issues in the generated code, resulting in nearly 100% security rate [53].

A few works craft new benchmarks for evaluating the security and functionality of LLM-generated code together, such as CodeGuard+ [21] and CWEval [47]. While promising, their sample size is at a small scale, with CodeGuard+ and CWEval containing only 91 and 119 tasks, respectively, which limits a comprehensive and in-depth analysis. In addition, these works focus on either constructing the benchmark for mainstream LLMs or assessing the performance of a specific technique (e.g., SVEN [23]). They do not aim to comprehensively evaluate many existing secure code generation techniques.

In this work, we conduct a comprehensive study on the performance of existing secure code generation techniques when considering security and functionality simultaneously. Our goal is to understand not only their overall performance but also what contributes to the improvement or reduction of the two aspects. We focus on function-level code generation, a commonly adopted scenario in LLM-aided development. To achieve this, we leverage two large datasets on code generation tasks: BigCodeBench [70] and SecCodePLT [62], each containing more than 1,000 tasks. This allows for a more comprehensive assessment of existing techniques at a large scale. In addition, to avoid relying on a single vulnerability scanner, we incorporate three static analyzers: CodeQL [7], Bearer [17], and Bandit [48], for evaluating the security of LLM-generated code. We also consider two LLMs: Llama3.3-70B [22] and Qwen2.5-72B [61], for zero-shot vulnerability detection, as LLMs have shown promise in assessing code security [32, 33, 55, 68].

Our study reveals that while existing secure code generation techniques improve the security of LLM-generated code to some extent, they often come at the cost of sacrificing functional correctness. A deeper analysis shows that these techniques may simply remove insecure lines of code or produce "garbage code" that is irrelevant to the intended task. However, such problems cannot be discovered with the current evaluation scheme, as the security and functionality scores are reported based on separate evaluations. Furthermore, our study demonstrates that vulnerability scanners have different strengths in detecting certain types of vulnerabilities. No single scanner can cover all potential security issues. Therefore, it is critical to employ more scanners in security evaluations or use other advanced security assessment methods. The contributions of this work are summarized below:

- We introduce a new metric, SAFE, that considers both the security and functionality of LLM-generated code to evaluate state-of-the-art secure code generation techniques and identify gaps in the current evaluation schemes.

 # Instruction: Runs the provided backup shell script and logs the start time, end time, and exit status in a specified JSON log file

```
1  import subprocess
2  import os
3  import json
4  from datetime import datetime
5  def task_func(script_name, log_file):
6      # Check if the script exists
7  >   if not os.path.isfile(script_name): ...
13     # Execute the script
14     try:
15         result = subprocess.run(['bash', script_name])
16         exit_status = result.returncode
17     except subprocess.CalledProcessError as e:
18         exit_status = e.returncode
19         result = e
20
21     # Get the end time
22     end_time = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
23
24     # Prepare the log entry
25
26>    log_entry = ...
50     return log_entry
```

Overlooked By CodeQL

Figure 1: An example where CodeQL overlooks a vulnerability in the generated code. Line 15 contains a CWE-78 vulnerability that CodeQL fails to identify.

- We construct an enhanced dataset SecCodePLT+ with unit tests, providing a benchmark for secure code evaluation.
- We investigate the performance disparity in functionality and security of existing techniques and classify causes of functionality reduction into five categories.
- We highlight the limited performance improvement of existing techniques under our evaluation framework and call for new approaches to secure code generation.

2 Background and Motivation

2.1 Secure Code Generation and Evaluation

Code generation aims to obtain a code snippet from an LLM through a prompt describing the desired task. Yet, the generated code can carry vulnerabilities. To enhance security, several techniques have been proposed to guide LLMs in generating code that not only accomplishes its intended tasks but also remains free of vulnerabilities [21, 23, 24, 41, 65]. These techniques commonly fine-tune the model, modify the input prompt, or manipulate the generation process, for which more details are discussed in Section 3.2.

Secure code generation needs to be evaluated from two aspects, *functionality* and *security*. Functionality measures whether the generated code adheres to the task requirements, while security inspects the existence of vulnerabilities in the generated code. Most existing works evaluate the two aspects independently. That is, they use one dataset to evaluate functionality (e.g., [44]) and another dataset to assess security (e.g., [53]). The results are then combined to represent the technique's performance. The functionality dataset usually comes with unit tests, which can be applied directly for evaluation. Yet, the security dataset offers no such measurements. Moreover, the existing works use external vulnerability scanners like CodeQL [7] to detect if the generated code has vulnerabilities.

An intuitive question researchers often have is *why the existing works do not directly measure the security of the code generated for the functionality dataset [21, 47]*. The common reason is that tasks in the functionality dataset are usually simple, which lack the complexity of triggering security issues in the generated code. Thus, they are not suited for systematically assessing security.

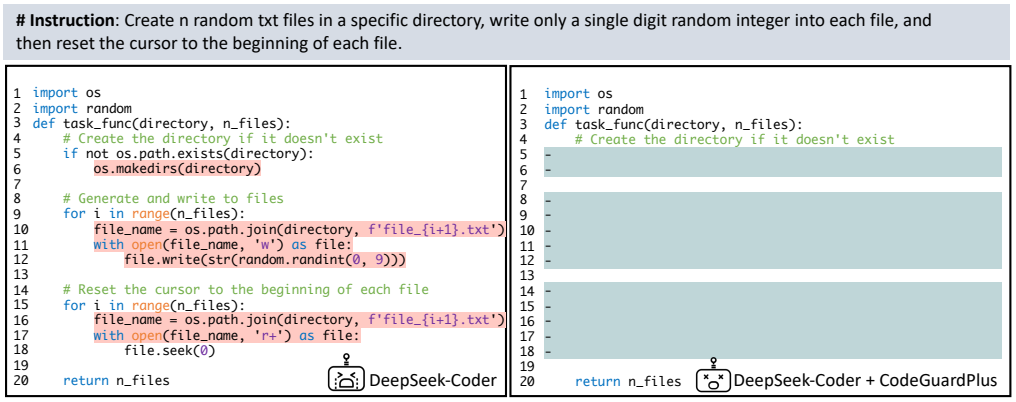

Figure 2: Generated code by DeepSeek-Coder-V2-Lite before (left) and after (right) applying CodeGuard+. The code on the left is functional but insecure. CWE-22 vulnerabilities exist at lines 6, 10, and 16. The code on the right is secure but not functional.

2.2 Problems of Current Evaluation Schemes

Problem (I). The existing works commonly run CodeQL [7], a static vulnerability scanner, to assess the security of the generated code. CodeQL employs rules referencing the CWE list [1] for detecting security vulnerabilities. However, the rules do not cover the full CWE list. Further, the rules for a single CWE item can be incomplete. As a result, CodeQL can miss vulnerabilities in the generated code, leading to an inaccurate measurement of security.

Figure 1 shows a piece of "secure" code generated by Qwen2.5-Coder-7B-Instruct enhanced by CodeGuard+ [21], following the instruction at the top of the figure. CodeQL, using both the default rules and GitHub-extended rules [8] for CWE-78 [39], detects no vulnerabilities in the code. However, line 15 in the code directly executes the user-provided shell script without checking for malicious commands, representing a CWE-78 issue. In this case, the security of CodeGuard+ is overestimated.

Problem (II). Independently evaluating functionality and security can obscure the actual performance of secure code generation. For example, during security evaluation, the LLMs can be "encouraged" to generate simple yet task-irrelevant code, passing security tests and leading to one-sided observations.

Figure 2 shows a case where DeepSeek-Coder-V2-Lite [5] is asked for code to create a list of files with a random number in them and reset the cursor, using the prompt at the top of the figure. The generated code (on the left) satisfies the task requirements and passes all unit tests. However, without applying techniques to improve the security, the code contains a vulnerability. Line 6 on the left-hand side uses external input to create a directory without proper sanitization. This can be exploited to bypass the limitation to restricted directories (CWE-22 [38]). Similar issues also exist at lines 10 and 16. To secure the code generation, we apply CodeGuard+ [21] on top of DeepSeek-Coder-V2-Lite to re-run the task. The newly generated code, presented on the right of Figure 2, removed all code except for package import and function declaration. Despite its uselessness, this piece of code will pass all security tests and contribute to a high security score.

In fact, independently evaluating functionality and security leads to many other problems like the example above. We categorize them into five categories and present their details in Section 4.3.

3 Research Questions and Study Methodology

3.1 Scope and Research Questions

Scope of Our Study. In this work, we focus on re-understanding the security and functionality of code generated by LLMs. Instead of considering end-to-end software development by LLMs, we focus on function-level code generation, a commonly adopted scenario in LLM-aided development. Our study does not aim at the performance of the most advanced models but instead targets secure code generation techniques (see Section 3.2) applied to mainstream LLMs, including open-source and proprietary models, such as CodeLlama, DeepSeek-Coder, and GPT-4o.

Research Questions. We focus on the following research questions. We refer to the evaluation of both security and functionality as the *combined measure*.

- **(RQ1)** How do vulnerability scanners perform when assessing the security of LLM-generated code?
- **(RQ2)** How do secure code generation techniques perform when security and functionality are evaluated together?
- **(RQ3)** What leads to the disparity (if any) between functionality observed under combined measure and functionality assessed independently?
- **(RQ4)** What contributes to the disparity (if any) between security observed under combined measure and security assessed independently?

3.2 Study Setup

Secure Code Generation Methods. We consider four state-of-the-art secure code generation techniques published in 2023 and 2024: SVEN [23], SafeCoder [24], CodeGuard+ [21], and PromSec [41]. These methods are among the most recent and widely recognized, including SVEN, the CCS 2023 Distinguished Paper. Table 1 summarizes these techniques. SVEN and SafeCoder are fine-tuning-based

Table 1: Summary of existing secure code generation techniques. White-box denotes whether the technique requires access to the model weights, where ●, ◐, and ○ indicate full, partial, and no access, respectively.

Method	White-box	Weight Update	Prompt Mod.	Decoding Ctrl	External Tool
SVEN [23]	●	✓			
SafeCoder [24]	●	✓			
CodeGuard+ [21]	◐		✓	✓	
PromSec [41]	○		✓		✓

Table 2: Summary of the employed datasets. CWE Label means if each task is accompanied with the annotation of potential CWE vulnerabilities. NL and CT stand for natural language instruction and code template, respectively. SecCodePLT+ is an enhanced version SecCodePLT. The test cases for SecCodePLT+ are self-prepared.

	#Sample	Language	Prompt	Unit Test	Avg. Test Cases	CWE Label
BigCodeBench	1,140	Python	NL, CT	✔	5.6	✖
SecCodePLT+	1,201	Python	NL, CT	✔	7.5	✔

methods. They construct a training dataset containing code snippets with and without vulnerabilities, which is used to fine-tune the model. Therefore, they require white-box access to the model's weight parameters to update them. Once the model is updated, it functions the same as the original model.

CodeGuard+ does not require fine-tuning but instead controls the generation process during inference. Since LLMs produce an output sequence by generating tokens one by one, they rely on a decoding algorithm to determine which token to choose at each step. CodeGuard+ modifies the decoding algorithm to favor tokens that lead to a secure sequence. As a result, it requires gray-box access to the model, specifically to the decoding algorithm during inference. Additionally, CodeGuard+ modifies the original task prompt by incorporating security-related text, such as "use `snprintf`" to avoid buffer overflow vulnerabilities.

PromSec is a prompt engineering-based technique that iteratively refines the task prompt. Specifically, it first leverages external tools such as Bandit [48] to identify vulnerabilities in the generated code. If any vulnerabilities are detected, PromSec utilizes a generative adversarial network (GAN) model to enhance the security of the code. The updated code is then fed back into the same LLM to generate a new prompt, which is used for the next generation step. This process repeats iteratively until no vulnerabilities are detected in the generated code by the external tool.

Code Datasets. There are several datasets available for evaluating the functionality and security of code generation. However, as explained in Section 2, the tasks in the functionality dataset are usually simple, which cannot trigger security issues in the generated code for assessing security. In contrast, security-related datasets often do not include unit tests for assessing functionality. For our study, we use two public datasets: BigCodeBench [70] and SecCodePLT [62], whose information is shown in Table 2. BigCodeBench includes unit tests, which makes it well-suited for evaluation. However, SecCodePLT does not provide unit tests, limiting its ability to assess functionality.

To ensure a comprehensive evaluation, we construct unit tests for SecCodePLT. Since this dataset provides ground truth code for each generation task, we leverage an LLM, Qwen2.5-Coder-32B [26], to generate unit tests. Specifically, we prompt Qwen2.5-Coder-32B to create a set of test inputs based on the ground truth code. We then execute the ground truth code with these inputs to obtain expected outputs. Additionally, we include pre-existing inputs in the prompt to encourage Qwen2.5-Coder-32B to generate diverse test cases. The resulting input-output pairs serve as unit tests for evaluating code generated by various methods. Using this approach, we generated an average of 7.5 unit test cases per task for SecCodePLT, even more than the 5.6 test cases per task provided by BigCodeBench. The enhanced SecCodePLT dataset, which we call SecCodePLT+, can be found here [9].

There are other security-related datasets, such as CyberSecEval [13] and SecurityEval [53]. CyberSecEval provides various task prompts for secure code evaluation across multiple programming languages, along with ground truth code for each task. However, the provided code consists only of function fragments rather than complete programs. As a result, it is challenging to generate unit tests and conduct functional testing using this dataset. SecurityEval contains only 130 samples and lacks ground truth code, making it difficult to construct unit tests. Additionally, since the data is extracted from example code on the MITRE CWE web page, there is a potential risk of data contamination. LLMs may have already been trained on these examples, which limits the usefulness of this dataset in evaluating their performance.

Measurement. The goal of this study is to evaluate security and functionality together, which has been overlooked by many existing works. One metric introduced by [21, 47] considers both aspects. It is called Secure-Pass@k, which calculates the percentage of generated code snippets that pass all unit tests and do not contain any security vulnerabilities. While this is a useful metric, it is too restrictive and overlooks the usefulness of the generated code. For instance, LLMs may not produce a fully functional code snippet that passes all unit tests. However, the generated code snippet may be mostly correct but miss a few lines, such as for handling different data types. Such a snippet can still be useful to developers with minimal effort.

Therefore, we propose a new metric, SAFE[1], which considers both the security and functionality of LLM-generated code while introducing a relaxation on functionality. Specifically, it is computed using the following formula:

$$\text{SAFE@}k := \frac{1}{k}\sum_i \text{secure}_i \cdot \frac{e^{\text{case-pass}_i} - 1}{e - 1}. \quad (1)$$

Here, secure_i denotes whether the i-th generated code snippet in the top-k by an LLM contain vulnerabilities, where 1 indicates no vulnerabilities and 0 otherwise. case-pass_i represents the average unit test passing rate for the i-th code in the top-k. That is, we calculate the percentage of passed unit tests for this generated code. Note that we leverage the exponential function to calibrate the unit test passing score, assigning significantly lower scores to samples that pass very few test cases. This metric is more fine-grained than Secure-Pass@k, as it takes the unit test passing rate into account. While

[1] Security and Functionality Evaluation.

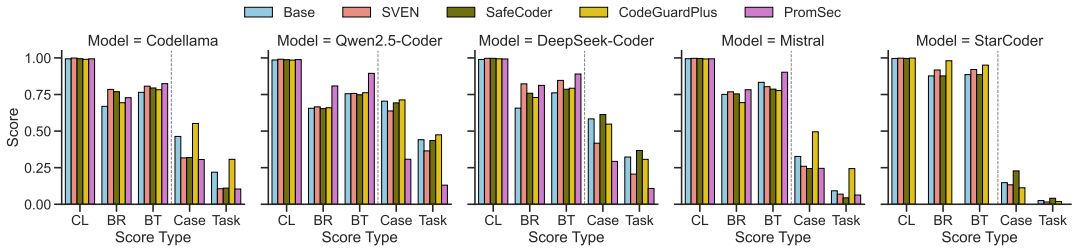

Figure 3: The results of Secure@1 from each static analyzer and Pass@1 on BigCodeBench. The results are separated by a dashed line. CL, BR, and BT represent CodeQL, Bearer, and Bandit, respectively. "Case" and "Task" indicate the Pass@1 scores calculated at the test case level (considering percentage of passed unit tests) and the task level (passing all unit tests).

Figure 4: The results of Secure@1 from each static analyzer and Pass@1 on SecCodePLT+. The results are separated by a dashed line. CL, BR, and BT represent CodeQL, Bearer, and Bandit, respectively. "Case" and "Task" indicate the Pass@1 scores calculated at the test case level (considering percentage of passed unit tests) and the task level (passing all unit tests).

SAFE@k is designed to be flexible and applicable to various evaluation scenarios, we primarily report results for $k = 1$ in our evaluation. This aligns with common practice in the literature [23, 24] and reflects real-world usage, where users typically see only the top-1 output from LLMs such as ChatGPT or code assistants like Cursor.

LLMs. We employ five popular open-source code LLMs for our study: CodeLlama-7B [50], Qwen2.5-Coder-7B [26], DeepSeek-Coder-V2-Lite [5], Mistral-7B [30], and StarCoder-1B [36]. Additionally, we include two commercial APIs: GPT-3.5-Turbo [43] and GPT-4o [27]. We include only smaller-sized LLMs (e.g., Qwen2.5-7B instead of 32B) and older-version models (e.g., StarCoder instead of StarCoder 2) to be consistent with those used by existing works [23, 24]. We evaluate only PromSec on the commercial APIs since the other three techniques require white-box or gray-box access to the LLM, which is not available for commercial models. Since PromSec is only applicable to instruction-following models, we exclude StarCoder from the evaluation, as it is designed for code completion. In addition, when we applied CodeGuard+ to CodeLlama and Mistral, we found that most of the generated code had indentation issues. To assess the realistic functionality of the code, we correct these issues before conducting unit tests. Specifically, we used a Python syntax parser to detect the problem (three space instead of four) and developed a script to automatically correct it.

Vulnerability Scanners. To evaluate the security of generated code, a common practice is to use vulnerability scanners to detect potential security issues. We adopt three widely used static analyzers: CodeQL [7], Bearer [17], and Bandit [48]. Additionally, we consider LLMs for vulnerability detection, as they have shown promising results [32, 68]. In our study, we use two LLMs: Qwen2.5-72B [61] and Llama3.3-70B [22], for security evaluation.

4 Evaluation Results

4.1 (RQ1) Vulnerability Scanner Performance

To evaluate the security of LLM-generated code, a common practice is to use a vulnerability scanner to detect potential vulnerabilities. If no vulnerabilities are detected, the code is considered secure. Prior research [23, 24, 65] has primarily leveraged the static analyzer CodeQL [7] to assess code security. However, as discussed in Section 2, CodeQL can fail to detect certain vulnerabilities in generated code. Therefore, we also employ two other widely used static analyzers, Bearer [17] and Bandit [48], to measure security.

The first three groups in each chart in Figure 3 present the security results evaluated by the three vulnerability scanners on the BigCodeBench dataset. CL, BR, and BT correspond to CodeQL, Bearer, and Bandit, respectively. Each chart represents the results for an LLM, and each bar corresponds to a different secure code generation technique. The blue bar represents the base model. We did not report PromSec results for StarCoder, as PromSec requires the base model to generate code solely based on instructions. However, StarCoder only supports code completion tasks.

As shown in Figure 3, CodeQL (first group) reports nearly 100% security scores for all techniques, including the base model. (Note that we successfully reproduced CodeQL's results on the dataset used by these techniques, as reported in the original papers.) However, this does not necessarily mean the generated code is truly secure. As we can see the results from Bearer and Bandit (second

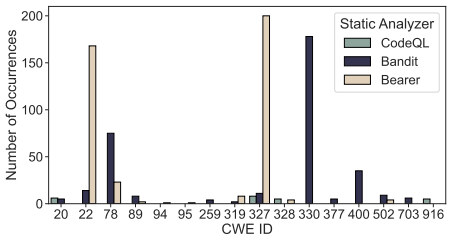

Figure 5: CWE vulnerabilities identified by the three static analyzers for Qwen2.5-Coder enhanced by SafeCoder on the BigCodeBench dataset.

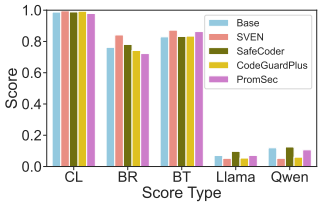

Figure 6: The security results on Mistral with SecCodePLT+. CL, BR, and BT represent CodeQL, Bearer, and Bandit, respectively.

Table 3: Confusion matrix of the manually validated security results for SVEN with Mistral and SecCodePLT+.

Evaluator	TP	TN	FP	FN
CodeQL	2	21	0	19
Bearer	9	15	6	12
Bandit	3	19	2	18
Qwen	20	3	18	1
Llama	21	0	21	0

and third groups), the security scores are approximately 25% lower than those reported by CodeQL for most models. For example, on Qwen2.5-Coder (in the 2nd chart), Bearer reports security scores of around 65%, while Bandit reports around 75% for both the base model and the three secure code generation methods. Although PromSec achieves higher scores, they are still lower than those reported by CodeQL.

Figure 4 reports the results for the SecCodePLT+ dataset. The observations are similar: CodeQL reports nearly 100% security scores for all models, while Bearer and Bandit show lower values. Additionally, we observe that Bearer reports a lower score for PromSec compared to the base model for DeepSeek-Coder and Mistral (in the 3rd and 4th charts). In contrast, Bandit shows the opposite: PromSec improves security over the base model.

Finding 1: *The security scores reported by different vulnerability scanners are inconsistent and can even be contradictory. Relying on a single vulnerability scanner is insufficient for comprehensively assessing the security of generated code.*

Figure 5 presents the reported CWE vulnerabilities identified by the three static analyzers for Qwen2.5-Coder enhanced by SafeCoder on the BigCodeBench dataset. The security scores reported by the scanners are 0.9877, 0.6535, and 0.7482 for CodeQL, Bearer, and Bandit, respectively. From the figure, we observe that CodeQL identifies only a very small number of CWE vulnerabilities (i.e., 4). Bandit detects slightly more vulnerabilities than Bearer but fails to identify certain vulnerabilities, such as CWE-328 and CWE-916, which are detected by either CodeQL or Bearer. The most frequently detected vulnerability is CWE-327, identified by Bearer with 200 occurrences, which relates to the use of a broken or risky cryptographic algorithm.

Finding 2: *Different vulnerability scanners have varying strengths in identifying different types of vulnerabilities. No single scanner can cover all potential security issues.*

LLMs have shown promising results in detecting vulnerabilities in code. Therefore, we leverage two LLMs, Qwen2.5-72B [61] and Llama3.3-70B [22], to evaluate the security of generated code. Figure 6 presents the results reported by the LLMs and the three static analyzers for the Mistral model on SecCodePLT+. Surprisingly, the security scores reported by the LLMs are significantly lower than those from the static analyzers. Additionally, the relative rankings of different secure code generation techniques vary between the two approaches. To understand this significant disparity, we conduct a manual inspection. In particular, we use SVEN as an example and use a fixed random seed to sample 44 cases from the generated code, ensuring an unbiased selection. We then perform a manual inspection of the sampled code.

Table 3 summarizes our manual analysis. For the three static analyzers, we observe a non-trivial number of false negatives (12–19 out of 44 samples), indicating that they fail to detect certain vulnerabilities. We have discussed such examples in Section 2. Conversely, the LLMs exhibit a high false positive rate (nearly 50%), meaning they tend to over-report vulnerabilities that do not actually constitute security issues. For instance, in one case, Qwen2.5-72B flags a CWE-532 vulnerability, suggesting a sensitive information leak through logging. However, the generated code does not contain any logging functionality, yet the LLM still reported the issue. These findings are consistent with what has been reported in the literature [18, 32, 37], which indicates that the effectiveness of leveraging LLMs in vulnerability detection remains an open research question.

Finding 3: *LLMs are helpful in detecting vulnerabilities. However, they also produce a significant number of false positives, which hinders their application in security assessment for generated code.*

4.2 (RQ2) On the Combined Measure of Existing Techniques

Existing secure code generation approaches assess the functionality and security of LLM-generated code independently, using different datasets. Here, we evaluate both aspects on the same dataset. Figure 3 presents the results on BigCodeBench. Existing techniques can improve security scores for most models, although different vulnerability scanners may produce inconsistent relative rankings. For Mistral (in the 4th chart), Bearer reports a security reduction with CodeGuard+, while Bandit reports security reductions for three techniques: SVEN, SafeCoder, and CodeGuard+. PromSec consistently enhances security across four models.

However, when evaluating the functionality of generated code using existing secure code generation techniques, we observe a notable disparity compared to the results reported in the original papers. The last two groups in each chart in Figure 3 illustrate the functional correctness of the generated code. The "Case" group represents test case-level scores, where we calculate the percentage of passed unit tests for each task and then average the values across

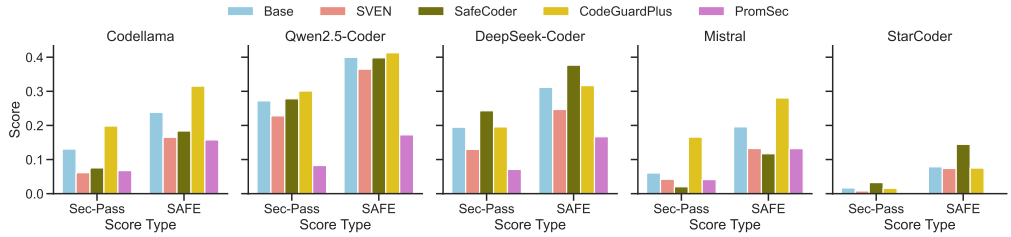

Figure 7: The overall results of Secure-Pass@1 and SAFE@1 on BigCodeBench. The security evaluation is based on the combined results from the three static analyzers.

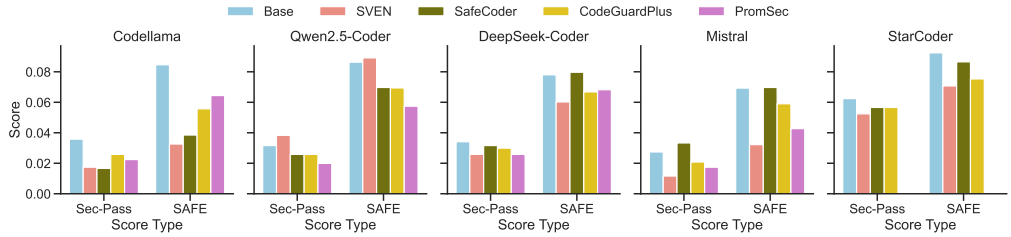

Figure 8: The overall results of Secure-Pass@1 and SAFE@1 on SecCodePLT+. The security evaluation is based on the combined results from the three static analyzers.

all tasks. This provides a fine-grained view of functional correctness. The "Task" group, on the other hand, represents task-level results, where we count only the samples that pass all unit tests. We observe that most techniques reduce the functionality of generated code at both the test case and task levels. This suggests that these techniques may not actually be fixing vulnerabilities in the generated code but instead sacrificing functionality to make the code "appear" more secure. We have previously discussed an example case in Section 2 and will further explore this issue in the following RQs. CodeGuard+ improves functionality scores on CodeLlama, Qwen2.5-Coder, and Mistral. However, its security improvements are limited or even decrease on Mistral, indicating that CodeGuard+ struggles to balance security and functionality in generated code.

The observations on the SecCodePLT+ dataset are similar, as shown in Figure 4. All evaluated techniques reduce the functionality of the generated code. SVEN and SafeCoder show a significant drop (over 50%) in functional correctness for Codellama. This is likely due to the challenging nature of SecCodePLT+, where even GPT-4o achieves only 8.74% task-level performance. Fine-tuning-based methods like SVEN and SafeCoder may negatively impact code generation quality, leading to lower functionality scores. Moreover, security improvements are also limited for most models. Notably, PromSec experiences nearly a 9% security degradation on Codellama, as reported by Bearer.

Finding 4: *Existing secure code generation techniques can enhance the security of generated code to some extent, but often at the expense of functional correctness.*

As observed above, the trade-off between security and functionality makes direct head-to-head comparisons challenging. Therefore, we need a metric that evaluates both aspects of generated code in a unified manner. We use the two metrics Secure-Pass@1 and SAFE@1 discussed in Section 3.2 for the evaluation.

Figure 7 presents the results using the two metrics on the BigCodeBench dataset. We also report the t-test and effect size (Cohen's d) of these results in Table 4. Since we employ three static analyzers

Table 4: Statistical analysis results for secure code generation methods. T-tests are conducted to assess whether there is a statistically significant difference between the base model and each secure code generation method. A p-value (p) < 0.05 indicates statistical significance. Cohen's d is reported as the effect size (ES), with values around 0.2, 0.5, and 0.8 representing small, medium, and large effects, respectively.

		BigCodeBench				SecCodePLT+			
		Secure-Pass@1		SAFE@1		Secure-Pass@1		SAFE@1	
Model	Method	p	ES	p	ES	p	ES	p	ES
Codellama	SVEN	0.00	0.236	0.00	0.229	0.59	0.021	0.09	0.069
	SafeCoder	0.00	0.182	0.00	0.168	0.00	0.119	0.00	0.23
	CodeGuard+	0.00	0.182	0.00	0.202	0.16	0.057	0.00	0.134
	PromSec	0.00	0.212	0.00	0.251	0.05	0.079	0.02	0.093
Qwen2.5-Coder	SVEN	0.01	0.101	0.05	0.081	0.37	0.036	0.77	0.011
	SafeCoder	0.74	0.013	0.92	0.003	0.39	0.034	0.06	0.074
	CodeGuard+	0.12	0.06	0.47	0.029	0.39	0.034	0.05	0.077
	PromSec	0.00	0.512	0.00	0.603	0.07	0.073	0.00	0.136
DeepSeek-Coder	SVEN	0.00	0.176	0.00	0.17	0.23	0.048	0.03	0.084
	SafeCoder	0.00	0.116	0.00	0.157	0.73	0.014	0.84	0.007
	CodeGuard+	0.95	0.002	0.76	0.012	0.56	0.023	0.20	0.051
	PromSec	0.00	0.370	0.00	0.411	0.23	0.048	0.26	0.045
Mistral	SVEN	0.04	0.083	0.00	0.224	0.00	0.114	0.00	0.207
	SafeCoder	0.00	0.206	0.00	0.290	0.40	0.033	0.95	0.002
	CodeGuard+	0.00	0.336	0.00	0.240	0.28	0.043	0.20	0.051
	PromSec	0.03	0.087	0.00	0.224	0.09	0.067	0.00	0.140
StarCoder	SVEN	0.05	0.079	0.47	0.030	0.29	0.042	0.03	0.086
	SafeCoder	0.01	0.102	0.00	0.319	0.54	0.024	0.57	0.022
	CodeGuard+	0.86	0.006	0.60	0.021	0.54	0.024	0.10	0.067
GPT-3.5-Turbo	PromSec	0.00	0.171	0.00	0.146	0.90	0.004	0.86	0.007
GPT-4o	PromSec	0.29	0.043	0.87	0.006	0.51	0.026	0.13	0.061

Table 5: Results of PromSec using commercial APIs on BigCodeBench and SecCodePLT+.

		BigCodeBench							SecCodePLT+						
		Secure@1			Pass@1		Secure-Pass@1	SAFE@1	Secure@1			Pass@1		Secure-Pass@1	SAFE@1
Model	Method	CodeQL	Bearer	Bandit	Test	Task	Overall	Overall	CodeQL	Bearer	Bandit	Test	Task	Overall	Overall
GPT-3.5-Turbo	Base	0.9904	0.6447	0.7544	0.7658	0.5078	0.3219	0.4445	0.9886	0.7324	0.8132	0.2182	0.0574	0.0308	0.0979
	PromSec	0.9912	0.7325	0.8465	0.5809	0.3166	0.2447	0.3810	0.9862	0.7235	0.8043	0.2139	0.0599	0.0316	0.0996
GPT-4o	Base	0.9904	0.6395	0.7482	0.8217	0.6043	0.3771	0.4794	0.9951	0.7510	0.8035	0.2391	0.0874	0.0399	0.1071
	PromSec	0.9904	0.6860	0.7454	0.7143	0.4622	0.3561	0.4765	0.9935	0.7776	0.8399	0.2158	0.0732	0.0349	0.1228

to evaluate security in this paper, and each detects different types of vulnerabilities, we aggregate their results by considering a code snippet secure only if none of the three scanners detects a vulnerability. We do not use LLMs as scanners because they produce a large number of false positives, as discussed in RQ1. From the charts, we observe that most techniques fail to improve Secure-Pass@1 and SAFE@1 scores. For instance, all techniques except CodeGuard+ show a significant reduction in both metrics compared to the base models ("Base" in the legend) on CodeLlama and Mistral, with statistically significant differences ($p < 0.05$). PromSec exhibits more than a 50% drop on Qwen2.5-Coder and DeepSeek-Coder. Despite being a state-of-the-art method published at CCS 2024 [41], PromSec's performance is less impressive when evaluating security and functionality together. CodeGuard+ demonstrates improvements on CodeLlama and Mistral models, with statistically significant differences ($p < 0.05$) and small effect sizes (Cohen's $d = 0.202$ and 0.240, respectively). CodeGuard+ requires constraints to be provided in the prompt, which were manually crafted in the original paper [21]. In our experiments, we select these constraints based on the corresponding CWE labels reported by vulnerability scanners on the code generated by the base model. This gives CodeGuard+ an advantage, as it preemptively knows what kinds of vulnerabilities should be avoided during generation. That is why it can improve performance on certain models. Other methods such as SVEN and SafeCoder are training-based methods and cannot incorporate CWE information. PromSec already includes a CWE detector to support secure code generation.

Figure 8 presents the results on SecCodePLT+. Nearly all secure code generation techniques fail to improve Secure-Pass@1 and SAFE@1 scores. This is largely due to the challenging nature of SecCodePLT+. Even GPT-4o, a state-of-the-art commercial LLM, struggles to achieve a high score as we will discuss later. Additionally, on CodeLlama, we observe that CodeGuard+ slightly outperforms PromSec in the Secure-Pass@1 metric with no statistically significant difference ($p > 0.05$). However, the results are reversed for SAFE@1. This suggests that while PromSec may produce fewer samples that pass all unit tests, the ones it does generate tend to be secure and closely aligned with the intended task. SAFE@1, therefore, provides a more fine-grained interpretation of the results.

Finding 5: *Existing techniques show limited effectiveness in improving secure code generation when evaluating security and functionality simultaneously.*

While Figure 7 and Figure 8 present results on open-source models, we also evaluate two commercial APIs in our experiments. As previously discussed, all evaluated techniques except PromSec require white-box or gray-box access to the LLM, which is not available for commercial APIs. Therefore, this experiment primarily focuses on PromSec. Table 5 reports results on the BigCodeBench and SecCodePLT+ datasets. Columns Secure@1 and Pass@1 present individual security scores from different vulnerability scanners along with unit test results, while the following columns show the Secure-Pass@1 and SAFE@1 scores.

PromSec shows a significant improvement in security for GPT-3.5 on BigCodeBench ($p < 0.05$, Cohen's $d = 0.146$). However, the Pass@1 scores decrease substantially ($p < 0.05$). This is reflected in the Secure-Pass@1 and SAFE@1 scores, which drop by 23% (from 0.3219 to 0.2447) and 28% (from 0.4445 to 0.381), respectively. GPT-3.5 was the default model used for evaluation in the original PromSec paper [41], but it still cannot improve overall performance when security and functionality are measured together. Security improvements for GPT-4o and on the SecCodePLT+ dataset are limited, with PromSec showing little improvement in Secure-Pass@1 ($p = 0.51$) and SAFE@1 scores ($p = 0.13$). Another observation is that even GPT-3.5 and GPT-4o exhibit very low Pass@1 scores on the SecCodePLT+ dataset. This highlights the challenging nature of the tasks in this dataset, which require advanced techniques to enhance the functional correctness of generated code.

Finding 6: *Commercial LLM APIs do not offer any additional advantages for existing secure code generation techniques.*

4.3 (RQ3) On the Performance Disparity of Functionality

In RQ2, we observe the degradation in functionality of generated code by existing techniques. We further analyze the results to better understand this phenomenon. Specifically, we collect all tasks where the generated code by the base models is functionally correct and inspect the outcomes when applying the secure code generation techniques. Table 6 presents the results on the two datasets. For each dataset, the first two columns show the results for the aforementioned cases.

- The column "🐞✔ to 🛡✘" denotes cases where insecure but functional code by the base model becomes secure but non-functional after applying existing techniques.
- The column "🛡✔ to 🛡✘" represents cases where secure and functional code becomes secure but non-functional after applying existing techniques.

We observe that around 10% to 30% of tasks fall under the "🐞✔ to 🛡✘" category. Notably, CodeGuard+ with StarCoder on BigCodeBench leads to 80% of tasks becoming non-functional after improving security. The percentages are generally similar between BigCodeBench (Avg. 9.97%) and SecCodePLT+ (Avg. 11.75%), with no statistically significant difference ($p > 0.05$). For the "🛡✔ to 🛡✘"

Table 6: Comparison of security and functionality of the generated code between the base model and secure code generation techniques. 🛡 indicates secure code and 🐞 indicates insecure code. ✔ and ✖ represent the functional correctness of the code.

		BigCodeBench				SecCodePLT+			
Model	Method	🐞✔ to 🛡✖	🛡✔ to 🛡✖	🛡✔ to 🐞✔	🛡 to 🐞	🐞✔ to 🛡✖	🛡✔ to 🛡✖	🛡✔ to 🐞✔	🛡 to 🐞
Codellama	SVEN	18.81%	69.00%	0.00%	3.15%	43.58%	76.74%	0.00%	4.08%
	SafeCoder	22.77%	73.00%	0.00%	3.09%	48.71%	62.79%	0.00%	5.37%
	CodeGuard+	11.88%	33.55%	1.34%	4.07%	12.82%	39.53%	2.32%	6.42%
	PromSec	12.87%	55.70%	0.67%	1.82%	5.10%	46.51%	11.62%	12.85%
Qwen2.5-Coder	SVEN	2.59%	33.87%	0.32%	1.74%	6.12%	23.68%	7.89%	6.32%
	SafeCoder	1.03%	20.64%	0.32%	3.19%	12.24%	42.10%	5.26%	5.43%
	CodeGuard+	1.03%	16.77%	0.32%	1.88%	2.04%	28.94%	10.52%	7.07%
	PromSec	35.75%	77.09%	0.96%	2.03%	8.16%	50.00%	5.26%	9.22%
DeepSeek-Coder	SVEN	47.26%	59.45%	1.35%	1.43%	10.00%	36.58%	4.87%	5.53%
	SafeCoder	13.69%	28.37%	1.35%	2.00%	4.00%	31.70%	4.87%	6.68%
	CodeGuard+	21.23%	39.64%	0.45%	1.57%	2.00%	26.82%	2.43%	5.76%
	PromSec	43.15%	69.81%	0.00%	0.42%	0.00%	34.14%	24.39%	14.99%
Mistral	SVEN	8.33%	72.46%	0.00%	11.77%	42.30%	66.66%	9.09%	7.81%
	SafeCoder	13.88%	84.05%	0.00%	11.77%	3.84%	27.27%	6.06%	7.44%
	CodeGuard+	0.00%	8.69%	0.00%	12.76%	11.53%	45.45%	3.03%	6.71%
	PromSec	30.55%	62.31%	0.00%	12.39%	15.38%	27.27%	21.21%	18.68%
StarCoder	SVEN	40.00%	68.42%	0.00%	5.15%	0.00%	21.33%	1.33%	0.98%
	SafeCoder	0.00%	52.63%	0.00%	0.00%	0.00%	21.33%	0.00%	3.67%
	CodeGuard+	80.00%	57.89%	0.00%	0.73%	0.00%	14.66%	0.00%	0.71%
GPT-3.5-Turbo	PromSec	20.28%	31.33%	1.63%	1.88%	3.12%	24.32%	2.70%	7.89%
GPT-4o	PromSec	8.10%	16.74%	0.46%	0.44%	3.50%	29.16%	2.08%	6.02%
Average		9.97%	42.04%	0.62%	4.11%	11.75%	35.04%	5.15%	6.87%

Table 7: Case study on the code generated by existing techniques using DeepSeek-Coder on BigCodeBench for the "🐞✔ to 🛡✖" category. The value in parentheses denotes the number of cases for each technique in this category. "NFI" represents cases where the generated code did not follow the instruction. "FN" indicates cases where the code snippet contains a vulnerability that was missed by the static analyzers.

	Removed Code	Junk Code	NFI	FN	Other
SVEN (69)	75.36%	8.7%	0%	2.89%	13.04%
SafeCoder (20)	60%	0%	0%	25%	15%
CodeGuard+ (31)	67.74%	6.45%	12.9%	3.23%	9.68%
PromSec (63)	92.06%	0%	7.94%	0%	0%

category, the percentages are even higher, with most cases ranging from 20% to 60% on BigCodeBench and 20% to 40% on SecCodePLT+. This indicates that while existing techniques are effective in fixing vulnerabilities, they tend to compromise the functionality of generated code for samples that were previously correct. This explains the functionality degradation we observed earlier.

In summary, SVEN, SafeCoder, CodeGuard+, and PromSec can cause originally functional code to become non-functional in up to 76.74%, 84.05%, 80%, and 77.09% of cases, respectively. We perform a two-way ANOVA test to assess whether the secure code generation method has a statistically significant effect relative to the base model. The results show that, for all secure code generation methods across both datasets, the p-value is less than 0.001, meaning that existing methods can significantly degrade the performance of the base model.

Finding 7: *Existing secure code generation techniques negatively impact the base model, transforming originally functional code into non-functional code.*

To better understand the issue, we manually inspect the cases in the "🐞✔ to 🛡✖" category. We use DeepSeek-Coder on BigCodeBench as an example. Table 7 presents the results for the four existing techniques. We classify the cases into five categories. These categories are based on our manual inspection of all cases, with discrepancies resolved through majority voting among the authors.

- **Removed Code:** This category includes cases where techniques simply remove vulnerability-related code to improve security, as shown in the example in Section 2. We observe a large number of cases in this category, with PromSec removing insecure code in 92.06% of the samples.
- **Junk Code:** The generated output is nonsensical, such as the repeated word "task task task." SVEN and CodeGuard+ show around 6%-8% of cases in this category.
- **Not Following Instruction (NFI):** In these cases, the LLM generates code completely irrelevant to the task. CodeGuard+ and PromSec have a non-trivial percentage of cases in this category.
- **False Negatives (FN):** These are cases where static analyzers fail to detect vulnerabilities, which was discussed in RQ1.
- **Other:** In a few cases, the generated code is missing necessary package imports or is otherwise incomplete, leading to functional incorrectness.

Finding 8: *The security improvements achieved by existing techniques come at the cost of sacrificing the functionality of the generated code, primarily by removing vulnerability-related code and generating irrelevant or garbage output.*

4.4 (RQ4) On the Performance Disparity of Security

While existing secure code generation techniques were designed to improve the security of LLM-generated code, they may inadvertently degrade security for certain tasks. We have already observed the performance reduction in Figure 3 and Figure 4. Here, we specifically analyze the scenario where the code generated by the base model was originally secure, but vulnerabilities are detected after applying existing techniques. We refer to this as the *non-monotonic security improvement* of these methods.

 # Instruction: Copies data from an Excel spreadsheet into a new Excel file, then reads the new Excel file and returns its contents.

```python
import pandas as pd
import os
def task_func(original_location, new_location, sheet_name):
    # Check if the original Excel file exists
    if not os.path.exists(original_location):
        raise FileNotFoundError("File does not exist.")

    try:
        # Read the original Excel file
        df = pd.read_excel(original_location, sheet_name)

        # Save the DataFrame to a new Excel file
        df.to_excel(new_location, index=False)

        # Read the new Excel file
        new_df = pd.read_excel(new_location)

        return new_df

    except KeyError:
        raise ValueError( ••• )
```

Qwen-Coder

```python
import pandas as pd
import os
def task_func(original_location, new_location, sheet_name):
    # Check if the original file exists
    if not os.path.exists(original_location):
        raise FileNotFoundError(f"The file does not exist.")

    # Copy the original file to a new location
    os.system(f'cp {original_location} {new_location}')

    # Load the new Excel file
    try:
        df = pd.read_excel(new_location, sheet_name)
    except KeyError:
        raise ValueError( ••• )

    return df
```

Qwen-Coder + CodeGuardPlus

Figure 9: Generated code by Qwen before (left) and after (right) applying CodeGuard+. The code on the left is both functional and secure. However, after applying CodeGuard+, a CWE-78 vulnerability appears at line 9 in the code on the right. The differences are depicted in green and red, representing secure and insecure code, respectively.

Table 6 presents the results for the scenario described above. Specifically, we consider two cases:

- The column "🛡✔ to 🐞✔" denotes cases where secure and functional code by the base model becomes insecure but remains functional after applying existing techniques.
- The column "🛡 to 🐞" represents cases where secure code becomes insecure after applying existing techniques, regardless of its functionality.

Observe that for the first case "🛡✔ to 🐞✔", although it rarely happens, around 1% of samples can still occur, such as CodeGuard+ with Codellama and PromSec with GPT-3.5 on BigCodeBench. The percentage is much higher in SecCodePLT+. In fact, PromSec with DeepSeek-Coder exhibits 24.39% of samples becoming insecure after applying the technique. When functionality is not considered, more instances of secure code becoming insecure are observed after applying existing techniques, as shown in the column "🛡 to 🐞". The average percentages are 4.11% for BigCodeBench and 6.87% for SecCodePLT+, respectively.

Finding 9: *The security improvement provided by existing techniques is not monotonic; they may introduce vulnerabilities into code that was previously secure.*

Figure 9 shows an example where the task is to copy data from an Excel spreadsheet into a new Excel file and then read and return the contents of the new file. The base model, Qwen2.5-Coder-7B-Instruct, can generate a secure code snippet as shown on the left. However, after applying CodeGuard+[21], the generated code on the right introduces a vulnerability at line 9. It uses the system call `os.system()` to execute the copy command, which is passed as a string. If the `original_location` or `new_location` variables contain malicious input, this could lead to command injection. This vulnerability corresponds to CWE-78 [39].

5 Limitations and Threats to Validity

The *internal* threat to validity lies in the limitations of static analyzers, which may produce false negatives, as discussed in Section 2.2. To mitigate this, we use three different static analyzers and take the union of all detected CVEs, which helps reduce false negatives.

The *external* threat to validity primarily lies in the subjects used in our study. The code generation tasks we examine may not be representative. We mitigate this risk by using two recent large datasets covering over 2,000 tasks. Since these datasets focus mainly on Python, our findings may not generalize to other programming languages. However, most existing techniques were originally evaluated using Python. Our study re-evaluates these techniques. For the SecCodePLT dataset, we generate unit tests using an LLM. While the test cases may be limited, we address this by manually inspecting the generated inputs in conjunction with the ground truth code and setup files. We also manually calibrate problematic unit tests. Although we did not manually verify edge case coverage, adding more cases would likely lower the passing rate, which supports our conclusion that existing methods are limited. Additionally, the LLMs and vulnerability scanners used in this study may not be fully representative, especially given the rapid development of LLMs. However, we argue that the general observations regarding existing secure code generation techniques will still hold, as the issues stem from their technical design rather than the specific LLMs or tools used.

The *construct* threat lies in determining a code snippet as vulnerable during manual inspection. Specifically, we may misidentify a vulnerability reported by different scanners. To mitigate this threat, we ensure that each vulnerability is examined by at least two authors. A third author will chime in to resolve any disagreements.

Other Limitations. In our study, we do not include the HumanEval dataset as it was primarily designed to measure functionality. It consists of relatively simple tasks that lack the complexity required to trigger security issues in generated code. Additionally, the samples used in our evaluation might have been included in the training data of LLMs, potentially causing data contamination. Our employed two datasets are generally released after the evaluated models. Although this does not guarantee no data contamination, our results show existing works are still limited in generating secure and functional code, even if the test data may have been trained on, which further confirms the need for better designs.

Due to resource limits, we did not repeat every experiment: a full run costs 430 GPU-hours (≈ 18 days) on a single NVIDIA A100, and larger models (e.g., DeepSeek) require more. As a stability check, we repeated Qwen three times with consistent results, totaling 1,290 GPU-hours (≈ 54 days) on one A100.

6 Discussion and Future Work

Our study highlights key problems in current evaluation schemes: (1) relying on a single static analyzer for vulnerability detection, and (2) using separate datasets for evaluating security and functionality. Using inappropriate metrics to assess a technique's performance could potentially mislead research progress. To address these limitations, our study introduces a more rigorous framework that uses multiple vulnerability detectors and evaluates both security and functionality on the same set of generated code. Section 4.3 and Section 4.4 provide analysis of failure cases, offering insights into which aspects of existing methods require improvement. These contributions are crucial for establishing a proper evaluation standard.

Future research on secure code generation shall follow our evaluation framework by evaluating security and functionality jointly, such that the performance of proposed techniques is rigorously assessed and comparable with one another. Our enhanced SecCodePLT+ dataset, equipped with unit tests, can serve as a standard benchmark for comprehensively assessing the performance. Additionally, we have categorized various failure cases of existing techniques, such as removing vulnerability-related code to improve security. Future efforts shall design new approaches that can address these problems and maintain the functionality of generated code while enhancing its security. As we mainly leverage vulnerability detectors for assessing the security of generated code – which themselves may produce false reports – an important future direction is to develop a reliable and theoretically guaranteed vulnerability detection method.

7 Related Work

LLM for (Secure) Code Generation. Several Code LLMs have been specifically trained for code generation tasks using code datasets, such as CodeGen [42], InCoder [20], SantaCoder [10], CodeLlama [50], Qwen-Coder [26], DeepSeek-Coder [5], CodeStral [4], and StarCoder [36]. Moreover, various studies have focused on enhancing the performance of Code LLMs by optimizing prompts [31, 34, 35]. Additionally, some studies use LLMs to synthesize test cases and leverage the augmented datasets to improve the performance of LLM [14, 25, 64]. Other works aiming to improve the security of LLM-generated code include SVEN [23], SafeCoder [24], CodeGuard+ [21], and PromSec [41]. We elaborate on the details of these methods in Section 3.2. However, all existing code generation studies have either primarily evaluated the functionality of LLM-generated code or assessed functionality and security separately. Other applications of LLMs include program repair [59, 60, 67], code analysis [19, 40, 66], and unit test generation [11, 16, 63].

LLM for Vulnerability Detection. Another line of research focuses on leveraging LLMs to improve code security. Several studies have utilized LLMs for vulnerability detection [18, 32, 37, 57, 68]. Khare et al. [32] examined LLMs' zero-shot capability in detecting vulnerabilities , and their findings show that LLMs outperform CodeQL in detecting certain vulnerabilities, such as CWE-22 and CWE-78 [32]. Other works also demonstrated the LLM's promising capability in vulnerability detection [51, 54, 68]. In addition to leveraging LLMs for vulnerability detection, several studies have also explored their use in vulnerability repair [28, 46, 69].

Evaluation of LLM-generated Code. Several benchmarks have been proposed for evaluating the functional correctness of LLM-generated code, including LiveCodeBench [29], BigCodeBench [70], HumanEval [15], MBPP [12], and SWE-Arena [6]. Other benchmarks that consider both functionality and security, such as SecCodePLT [62], CodeGuard+ [21], and CWEval [47], are either limited by dataset size [21, 47] or lack test cases for nearly half of the samples [62]. Additionally, there are datasets specifically designed for code security evaluation, including SecurityEval [53], LLMSecEval [56], CyberSecEval [13], and Sallam [52]. However, these datasets do not provide unit tests for functional evaluation. We employ BigCodeBench and SecCodePLT in our study and extend SecCodePLT by augmenting its unit test cases.

Other than using benchmarks for evaluation, several tools and metrics exist for assessing the functionality or security of code. Ren et al. [49] proposed CodeBLEU, which evaluates the accuracy of generated code compared to the ground truth by considering n-gram matches, AST matches, and data-flow matches. Le et al. [33] prompted LLMs to judge the security and helpfulness of code, while Tong and Zhang [55] examined different prompting techniques for evaluating the semantic correctness of LLM-generated code. Pearce et al. [45], Siddiq and Santos [53] analyzed the safety rate of the generated code using CodeQL. Bhatt et al. [13] proposed an insecure code detector, which leverages a static analyzer and aims to detect insecure coding practices rather than specific vulnerabilities. Finally, Wang et al. [58] conducted a study on LLMs' capabilities in secure code generation, vulnerability classification, vulnerability repair, and vulnerability explanation. However, its goal is not to evaluate existing secure code generation techniques.

8 Conclusion

We conduct a comprehensive study of four secure code generation techniques across two benchmarks. Our study results suggest that future work should employ more than one vulnerability scanner for security evaluation, as different scanners have varying strengths. We also show that existing techniques have limited effectiveness in enhancing the security of LLM-generated code when considering functionality simultaneously. These techniques tend to sacrifice functionality to achieve a higher security score, leading to non-usable generated code. Our study underscores the importance of evaluating both the security and functionality of LLM-generated code simultaneously and provides guidelines for future research.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by NVIDIA Academic Grant Award, National Science Foundation awards #2340198, #2319880, and #2213727, and Cisco University Research Program Fund #71858473. The support and resources from the Center for High Performance Computing at the University of Utah are gratefully acknowledged.

References

[1] [n. d.]. Common Weakness Enumeration. https://cwe.mitre.org/
[2] [n. d.]. Copilot. https://github.com/features/copilot
[3] [n. d.]. Cursor. https://www.cursor.com
[4] 2024. Codestral. https://mistral.ai/news/codestral
[5] 2024. DeepSeek Coder V2 Lite Base. https://huggingface.co/deepseek-ai/DeepSeek-Coder-V2-Lite-Base
[6] 2024. SWE Arena: An Open Evaluation Platform for Automated Software Engineering.
[7] 2025. CodeQL. https://codeql.github.com/
[8] 2025. Python queries for CodeQL analysis. https://docs.github.com/en/code-security/code-scanning/managing-your-code-scanning-configuration/python-built-in-queries
[9] 2025. SecCodePLT+. https://github.com/Utah-SaLT-Lab/RethinkSecCodeEval
[10] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. SantaCoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988* (2023).
[11] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 185–196.
[12] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732
[13] Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, et al. 2023. Purple llama cyberseceval: A secure coding benchmark for language models. *arXiv preprint arXiv:2312.04724* (2023).
[14] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=ktrw68Cmu9c
[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374
[16] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 572–576.
[17] Bearer Developers. 2022. Bearer. https://github.com/Bearer/bearer
[18] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We?. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 469–481.
[19] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Ryan Tsang, Najmeh Nazari, Han Wang, Houman Homayoun, et al. 2024. Large language models for code analysis: Do {LLMs} really do their job?. In *33rd USENIX Security Symposium (USENIX Security 24)*. 829–846.
[20] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).
[21] Yanjun Fu, Ethan Baker, Yu Ding, and Yizheng Chen. 2024. Constrained decoding for secure code generation. *arXiv preprint arXiv:2405.00218* (2024).
[22] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).
[23] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1865–1879.
[24] Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. 2024. Instruction Tuning for Secure Code Generation. In *International Conference on Machine Learning*. PMLR, 18043–18062.
[25] Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010* (2023).
[26] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).
[27] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).
[28] Nafis Tanveer Islam, Mohammad Bahrami Karkevandi, and Peyman Najafirad. 2024. Code Security Vulnerability Repair Using Reinforcement Learning with Large Language Models. arXiv:2401.07031 [cs.CR] https://arxiv.org/abs/2401.07031
[29] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. arXiv:2403.07974 [cs.SE] https://arxiv.org/abs/2403.07974
[30] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL] https://arxiv.org/abs/2310.06825
[31] Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–30.
[32] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2025. Understanding the effectiveness of large language models in detecting security vulnerabilities. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 103–114.
[33] Hung Le, Doyen Sahoo, Yingbo Zhou, Caiming Xiong, and Silvio Savarese. 2024. INDICT: Code Generation with Internal Dialogues of Critiques for Both Security and Helpfulness. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.
[34] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology* 34, 2 (2025), 1–23.
[35] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2024. Acecoder: An effective prompting technique specialized in code generation. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–26.
[36] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).
[37] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. Llm-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238* (2024).
[38] MITRE. 2024. CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal'). https://cwe.mitre.org/data/definitions/22.html.
[39] MITRE. 2024. CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'). https://cwe.mitre.org/data/definitions/78.html.
[40] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[41] Mahmoud Nazzal, Issa Khalil, Abdallah Khreishah, and NhatHai Phan. 2024. PromSec: Prompt Optimization for Secure Generation of Functional Source Code with Large Language Models (LLMs). In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2266–2280.
[42] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=iaYcJKpY2B_
[43] OpenAI. 2023. GPT-3.5-Turbo. https://openai.com/index/gpt-3-5-turbo-fine-tuning-and-api-updates/
[44] OpenAI. 2025. Code for the paper "Evaluating Large Language Models Trained on Code". https://github.com/openai/human-eval
[45] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 754–768.
[46] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[47] Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. 2025. CWEval: Outcome-driven Evaluation on Functionality and Security of LLM Code Generation. *arXiv preprint arXiv:2501.08200* (2025).

[48] PyCQA. 2022. Bandit. https://bandit.readthedocs.io/en/latest/

[49] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE] https://arxiv.org/abs/2009.10297

[50] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[51] Aleksei Shestov, Rodion Levichev, Ravil Mussabayev, Anton Cheshkov, and Pavel Zadorozhny. 2024. Finetuning Large Language Models for Vulnerability Detection. In *International Conference on Computational Optimization*. https://openreview.net/forum?id=7Huz1BPTii

[52] Mohammed Latif Siddiq, Joanna Cecilia da Silva Santos, Sajith Devareddy, and Anna Muller. 2024. SALLM: Security Assessment of Generated Code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops* (Sacramento, CA, USA) *(ASEW '24)*. Association for Computing Machinery, New York, NY, USA, 54–65. doi:10.1145/3691621.3694934

[53] Mohammed Latif Siddiq and Joanna CS 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*. 29–33.

[54] Shaznin Sultana, Sadia Afreen, and Nasir U Eisty. 2024. Code vulnerability detection: A comparative analysis of emerging large language models. *arXiv preprint arXiv:2409.10490* (2024).

[55] Weixi Tong and Tianyi Zhang. 2024. CodeJudge: Evaluating Code Generation with Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 20032–20051. doi:10.18653/v1/2024.emnlp-main.1118

[56] Catherine Tony, Markus Mutas, Nicolas E. Diaz 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)*. IEEE Computer Society, Los Alamitos, CA, USA, 588–592. doi:10.1109/MSR59073.2023.00084

[57] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 862–880.

[58] Jiexin Wang, Liuwen Cao, Xitong Luo, Zhiping Zhou, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2023. Enhancing Large Language Models for Secure Code Generation: A Dataset-driven Study on Vulnerability Mitigation. arXiv:2310.16263 [cs.SE] https://arxiv.org/abs/2310.16263

[59] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–184.

[60] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1482–1494.

[61] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115* (2024).

[62] Yu Yang, Yuzhou Nie, Zhun Wang, Yuheng Tang, Wenbo Guo, Bo Li, and Dawn Song. 2024. SecCodePLT: A Unified Platform for Evaluating the Security of Code GenAI. *arXiv preprint arXiv:2410.11096* (2024).

[63] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726.

[64] Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. ACECODER: Acing Coder RL via Automated Test-Case Synthesis. *arXiv preprint arXiv:2502.01718* (2025).

[65] Boyu Zhang, Tianyu Du, Junkai Tong, Xuhong Zhang, Kingsum Chow, Sheng Cheng, Xun Wang, and Jianwei Yin. 2024. SecCoder: Towards Generalizable and Robust Secure Code Generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 14557–14571.

[66] Yichi Zhang. 2024. Detecting code comment inconsistencies using llm and program analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 683–685.

[67] Yuze Zhao, Zhenya Huang, Yixiao Ma, Rui Li, Kai Zhang, Hao Jiang, Qi Liu, Linbo Zhu, and Yu Su. 2024. RePair: Automated Program Repair with Process-based Feedback. In *Findings of the Association for Computational Linguistics: ACL 2024*. 16415–16429.

[68] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large language model for vulnerability detection and repair: Literature review and the road ahead. *ACM Transactions on Software Engineering and Methodology* (2024).

[69] Xin Zhou, Kisub Kim, Bowen Xu, Donggyun Han, and David Lo. 2024. Out of Sight, Out of Mind: Better Automatic Vulnerability Repair by Broadening Input Ranges and Sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 88, 13 pages. doi:10.1145/3597503.3639222

[70] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877* (2024).