Poster: Rethinking the Evaluation of Secure Code Generation

Shih-Chieh Dai, Jun Xu, and Guanhong Tao
Kahlert School of Computing, University of Utah
Salt Lake City, Utah, 84112
shihchieh.dai@utah.edu, junxzm@cs.utah.edu, g.tao@utah.edu

***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. 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%.**

1. Introduction

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 [1], [2], [3], [4]. These techniques commonly fine-tune the model, modify the input prompt, or manipulate the generation process.

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 and another dataset to assess security. 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 to detect if the generated code has vulnerabilities.

Problem (I). Existing works commonly run CodeQL, a static vulnerability scanner, to assess the security of the generated code. CodeQL employs rules referencing the CWE list 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.

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.

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?

2. Study Setup

Secure Code Generation Methods. We consider four state-of-the-art secure code generation techniques published in 2023 and 2024: SVEN [1], SafeCoder [2], CodeGuard+ [3], and PromSec [4]. These methods are among the most recent and widely recognized, including SVEN, the CCS 2023 Distinguished Paper. SVEN and SafeCoder are fine-tuning-based methods. They construct a training dataset containing code snippets with and without vulnerabilities, which is used to fine-tune the model. CodeGuard+ does not require fine-tuning but instead controls the generation process during inference. PromSec is a prompt engineering-based technique that iteratively refines the task prompt.

Code Datasets. We use two public datasets: BigCodeBench and SecCodePLT. 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, to generate unit tests.

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 prior work [3] 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.

LLMs. We employ five popular open-source code LLMs: CodeLlama-7B, Qwen2.5-Coder-7B, DeepSeek-Coder-V2-Lite, Mistral-7B, and StarCoder-1B. Additionally, we include two commercial APIs: GPT-3.5-Turbo and GPT-4o.

Vulnerability Scanners. We adopt three widely used static analyzers: CodeQL, Bearer, and Bandit. Additionally, we consider LLMs for vulnerability detection, as they have shown promising results. In our study, we use two LLMs: Qwen2.5-72B and Llama3.3-70B, for security evaluation.

3. Evaluation Results

(RQ1) Vulnerability Scanner Performance. The security scores reported by different vulnerability scanners are often inconsistent and sometimes even contradictory, indicating that *relying on a single scanner is insufficient* for comprehensively assessing the security of generated code. Moreover, different scanners exhibit varying strengths in detecting specific types of vulnerabilities, and *no single tool can cover all potential security issues*. Although LLMs can assist in identifying vulnerabilities, they tend to generate a substantial number of false positives, which limits their practical effectiveness in security assessment for generated code.

(RQ2) On the Combined Measure of Existing Techniques. Existing secure code generation techniques can improve the security of generated code to some extent; however, these *improvements often come at the cost of reduced functional correctness*. When security and functionality are evaluated simultaneously, the overall effectiveness of these techniques appears limited, suggesting that their benefits may not hold under more comprehensive assessment criteria. Furthermore, the use of *commercial LLM APIs does not provide additional advantages* to existing secure code generation techniques, indicating that simply leveraging proprietary models does not fundamentally address the underlying challenges.

(RQ3) On the Performance Disparity of Functionality. *Existing secure code generation techniques can negatively impact the base model*, turning code that was originally functional into non-functional output. The observed security improvements often come at the expense of functionality, as these techniques tend to remove vulnerability-related code without preserving intended behavior and may even generate irrelevant or "garbage" output. As a result, the apparent *gains in security are frequently achieved by sacrificing the practical usability of the generated code*.

(RQ4) On the Performance Disparity of Security. The security improvement provided by existing techniques is not monotonic; they may *introduce vulnerabilities into code that was previously secure*.

1. Security and Functionality Evaluation.

4. Conclusion

We 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.

Attribution: This work has been accepted to 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE) [5].

References

[1] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1865–1879.

[2] J. He, M. Vero, G. Krasnopolska, and M. Vechev, "Instruction tuning for secure code generation," in *International Conference on Machine Learning*. PMLR, 2024, pp. 18 043–18 062.

[3] Y. Fu, E. Baker, Y. Ding, and Y. Chen, "Constrained decoding for secure code generation," *arXiv preprint arXiv:2405.00218*, 2024.

[4] M. Nazzal, I. Khalil, A. Khreishah, and N. Phan, "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*, 2024, pp. 2266–2280.

[5] S.-C. Dai, J. Xu, and G. Tao, "Rethinking the evaluation of secure code generation," in *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE '26)*. ACM, 2026.

Rethinking the Evaluation of Secure Code Generation

THE UNIVERSITY of UTAH

GitHub

Shih-Chieh Dai, Jun Xu , Guanhong Tao

shihchieh.dai@utah.edu

ICSE Paper: https://arxiv.org/abs/2503.15554

Limitations of Existing Evaluation Schemes

Current evaluation schemes are flawed:

- Security and functionality are evaluated **separately**
- Most studies rely on **only CodeQL**
- Some techniques sacrifice functionality for "security"

Secure Code Generation

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

Research Questions

- **(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?

Study Setup

Dataset:

	#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	✔

Measurement:

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

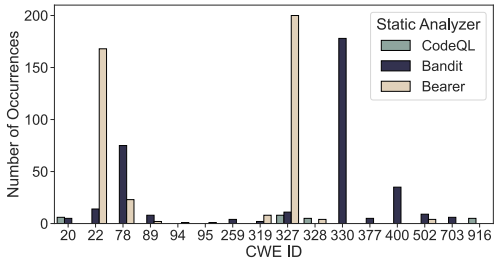
secure_i : 1 = secure, 0 = vulnerable

case-pass_i : Unit test passing rate of the i-th generated code.

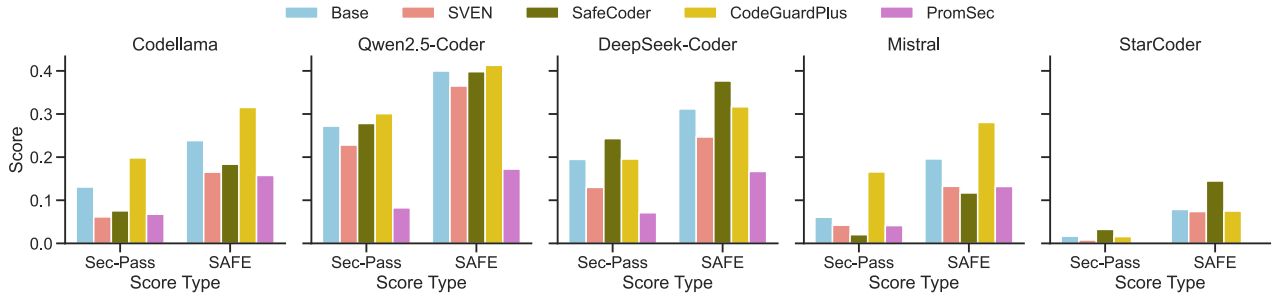
Results

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

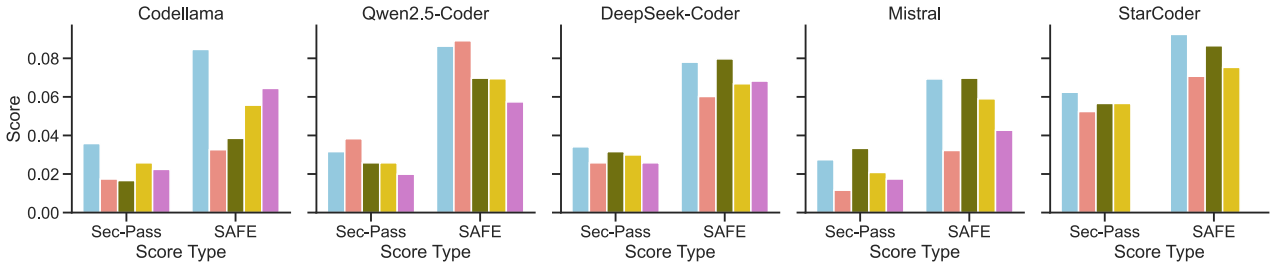
Confusion matrix of the manually validated security results (RQ1)



Vulnerabilities identified by different static analyzers. (RQ1)



Result on BigCodeBench (RQ2)



Result on SecCodePLT+ (RQ2)

Case Study on generated Code (RQ3)

	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%

NFI: Not Following Instructions. **FN**: False Negative

On the Performance Disparity of Security (RQ4)

- Secure code can become insecure after applying security techniques, showing a non-monotonic security effect (**4.11%** BigCodeBench, **6.87%** SecCodePLT+ on average).
- The impact varies by method and model, reaching **24.39%** (PromSec) secure→insecure cases.