Secure Code Generation at Scale with Reflexion

Arup Datta
University of North Texas
Denton, TX, USA
arupdatta@my.unt.edu

Ahmed Aljohani
University of North Texas
Denton, TX, USA
ahmedaljohani@my.unt.edu

Hyunsook Do
University of North Texas
Denton, TX, USA
hyunsook.do@unt.edu

***Abstract*—Large language models (LLMs) are now widely used to draft and refactor code, but code that *works* is not necessarily *secure*. We evaluate secure code generation using the Instruct Prime, which eliminated compliance-required prompts and cue contamination, and evaluate five instruction-tuned code LLMs using a zero-shot baseline and a three-round *reflexion* prompting approach. Security is measured using the Insecure Code Detector (ICD), and results are reported by measuring *Repair*, *Regression*, and *NetGain* metrics, considering the programming language and CWE family. Our findings show that insecurity remains common at the first round: roughly 25–33% of programs are insecure at a zero-shot baseline (t_0). Weak cryptography/config-dependent bugs are the hardest to avoid while templated ones like XSS, code injection, and hard-coded secrets are handled more reliably. Python yields the highest secure rates; C and C# are the lowest, with Java, JS, PHP, and C++ in the middle. Reflexion prompting improves security for *all* models, improving average accuracy from 70.74% at t_0 to 79.43% at t_3, with the largest gains in the first round followed by diminishing returns. The trends with *Repair*, *Regression*, and *NetGain* metrics show that applying one to two rounds produces most of the benefits. A replication package is available at https://doi.org/10.5281/zenodo.17065846.**

***Index Terms*—Large Language Models, Secure Code Generation, CyberSecEval, Reflexion, CWE, Software Security**

I. Introduction

LLMs such as GitHub Copilot, Codex, and DeepSeekCoder have made LLM-assisted coding common. Early studies focused on *functionality and correctness* [1], [2]: can models produce code that compiles and passes tests? Yet LLMs learn from large codebases that also contain design flaws. Recent work shows that low-quality code [3], [4] and vulnerabilities [5] can carry over into generated code. The question is not only "does it *work*," but also "is it *secure*?"

To answer this question, several security benchmarks have been proposed (see Section II). Early investigations established that LLM-generated code can be vulnerable, though these studies often relied on narrow or small-scale settings. For instance, Pearce et al. [6] demonstrated that GitHub Copilot frequently produces insecure completions. Siddiq [7] curated a Python-only dataset of vulnerable snippets to evaluate LLM performance on secure code generation tasks.

Subsequent work has refined code security benchmarks by rethinking task design, prompt formulation, and vulnerability detection. For example, Hajipour et al. [8] merged earlier datasets to improve coverage and examined how different prompting strategies affect security outcomes. Yetistiren et al. [9] adapted HumanEval tasks such that each item targeted a specific security property, offering reference implementations for evaluation. Tony et al. [10] linked natural language prompts to Common Weakness Enumeration (CWE) categories and applied CodeQL for consistent static analysis. These efforts improved dataset quality by aligning tasks to CWE taxonomies and removing noisy prompts. Nonetheless, most benchmarks remain limited in scope i.e., focusing on a single language or a narrow set of CWE categories.

To address the issue of scalability, Bhatt et al. [11], [12] introduced *CyberSecEval* and its successor *CyberSecEval 2*. These benchmarks introduced multilingual tasks, aligned prompts with CWE categories, and adopted a standardized scoring pipeline based on the Insecure Code Detector (ICD). This shift enabled broader coverage and more consistent evaluation across languages. However, follow-up analyses revealed that several dataset samples could inflate measured insecurity. Hariharan et al. [13] showed that some prompts implicitly required insecure behavior by design, and that identifiers and comments sometimes leaked cues that compromised validity. To mitigate these issues, they released a corrected benchmark, *Instruct Prime* [13], which removes compliance-required items and reduces cue contamination. While CyberSecEval provided the scale the field needed, its original release introduced confounds that warrant reevaluation.

Despite these advances, two important limitations persist in the way secure code generation is typically evaluated. First, most prior studies rely on single-turn prompting strategies such as zero-shot or few-shot prompts which allow the model only one opportunity to generate secure code. This approach does not test whether iterative prompting techniques could improve security outcomes. Second, evaluations usually emphasize first-pass success rates while overlooking regressions (e.g., a revised output reintroduces a vulnerability or introduce a new one). A comprehensive evaluation must account for both improvements and regressions across multiple rounds.

We therefore identify a remaining gap: there is no large-scale evaluation of secure code generation that (1) uses a high-coverage, validity-enhanced benchmark, and (2) evaluates iterative strategies that allow LLMs to reflect on and revise their outputs. To address this, we adopt the corrected CyberSecEval [13] as a reliable benchmark and apply reflexion prompting strategy [14], [15]. While prior work has shown that reflexion improves code correctness [16], it has not been systematically studied in the context of secure code generation. To our knowledge, this is the first systematic evaluation of self-

reflection for secure code generation at scale. We structure our evaluation around the following research questions:

1) **RQ1** What is the prevalence and distribution of insecure coding in LLM-generated code across languages and CWE categories?
2) **RQ2** Can LLMs improve the security of their own code through iterative reflection and refinement?

To answer these questions, we compare standard zero-shot instruction with our three-round reflexion approach. We re-score every revision and report results by CWE and language using the Repair, Regression, and NetGain metrics defined in Section III. This paper makes the following contributions:

1) We evaluate secure code generation using a large-scale study on a reliable and diverse benchmark.
2) Evaluating a three-round reflexion prompting, measuring whether LLMs can reflect on their own insecure code.
3) Utilizing Repair, Regression, and NetGain metrics that capture not only *Repairs*, but also *Regressions* and overall *NetGain*, providing a more realistic picture of iterative security improvements.
4) A replication package with prompts, generation logs, and scoring artifacts to enable reuse and auditing.[1]

The paper is organized as follows: Section II reviews related work on the automated generation of assertion statements. Section III outlines the methodology and experimental design. Section IV presents the results and key findings derived from our evaluation. Section V discusses the broader implications of the findings and provides additional insights. Section VI outlines the potential limitations and threats to the validity of our study. Finally, Section VII concludes the paper's findings.

II. Related Work

Early work on the security of LLM-generated code was established by Pearce et al. [6], who analyzed 1,689 GitHub Copilot completions across 89 scenarios and found that approximately 40% were insecure. Another study by Siddiq et al. [7] introduced *SecurityEval*, a curated suite focused on Python that covered a broad range of CWE categories. However, its single-language design limited generalizability across diverse programming languages.

Hajipour et al. [8] developed *CodeLMSec* by aggregating earlier datasets [6], [7] to evaluate the impact of alternative prompt instructions. Their findings revealed that ChatGPT exhibited minimal improvement in vulnerability rates under different prompt styles. They also reported 243 vulnerable outputs out of 783 Copilot generations across four CWEs. Similarly, Yetistiren et al. [9] adapted the HumanEval benchmark to assess security and functional correctness, and found wide variation in outcomes across models. Tony et al. [10] introduced *LLMSecEval*, a benchmark comprising 150 prompts mapped to the Top-25 CWE categories and evaluated using CodeQL. While this reduced cue bias, the dataset remained limited in both CWE and language diversity.

[1]https://doi.org/10.5281/zenodo.17065846

To address these limitations, Bhatt et al. [11] proposed *CyberSecEval*, a multilingual and CWE-aligned benchmark supporting both *instruct*- and *autocomplete*-style tasks, along with standardized ICD-based static scoring. They later expanded the benchmark in *CyberSecEval 2* [12] to include system-level vulnerabilities. Follow-up work by Hariharan et al. [13] demonstrated that many prompts in CyberSecEval included compliance-required vulnerabilities or leaked information via identifiers and comments. Their analysis showed that removing such items increased secure generation rates by 10.4 and 17.7 percentage points, respectively. They released a corrected version, *Instruct Prime*, with improved construct validity and reduced contamination.

Other benchmarks have further advanced this line of work. Wang et al. [17] introduced *CodeSecEval*, which provided both secure and insecure reference implementations, along with test oracles. They demonstrated that LLMs often pass functional tests while failing security checks and proposed mitigation strategies using vulnerability-aware prompting and explanation-guided repair. Yang et al. [18] proposed *SEC-CODEPLT*, which included expert-validated seeds, dynamic test cases, and a cloud deployment track, and reported higher real-world security relevance compared to CyberSecEval. Fu et al. [19] showed that constrained decoding via *CODEGUARD+* improves security-related metrics (e.g., secure-pass@5) compared to vanilla sampling.

Peng et al. [20] proposed *CWEVAL*, which employed dynamic oracles to assess both functionality and security. They found a substantial number of completions that were functionally correct but insecure, across various models and languages. At the system level, Dilgren et al. [21] released *SecRepoBench*, a benchmark built from 27 real-world C and C++ repositories. Their findings showed that models effective on short, self-contained snippets often struggle to generate secure and correct code at repository scale, where prompt engineering techniques become less effective.

Our study builds on this work by leveraging the corrected *Instruct Prime* benchmark [13] to retain breadth across programming languages and CWE categories. We introduce a training-free, model-agnostic evaluation of three-round *reflexion* prompting and systematically quantify its effects in terms of *Repair*, *Regression*, and *NetGain* measured by CWE type and programming language which have not been systematically explored in prior studies.

III. Research Methodology

To answer our research questions, we designed an experiment that evaluates five code LLMs on the Instruct Prime dataset [13]. Figure 1 sketches the workflow. We begin by selecting all prompts P_0 from the dataset (Step 1) and sending them to the LLM (Step 2) to produce outputs t_i (Step 3). Each output is then checked for vulnerabilities using the Insecure Code Detector (ICD) (Step 4). In the zero-shot setting, this yields a security label for t_0 and a set of result files (Step 6). In Step 7, we analyze these labels to estimate each model's baseline secure-code performance, directly answering RQ1.

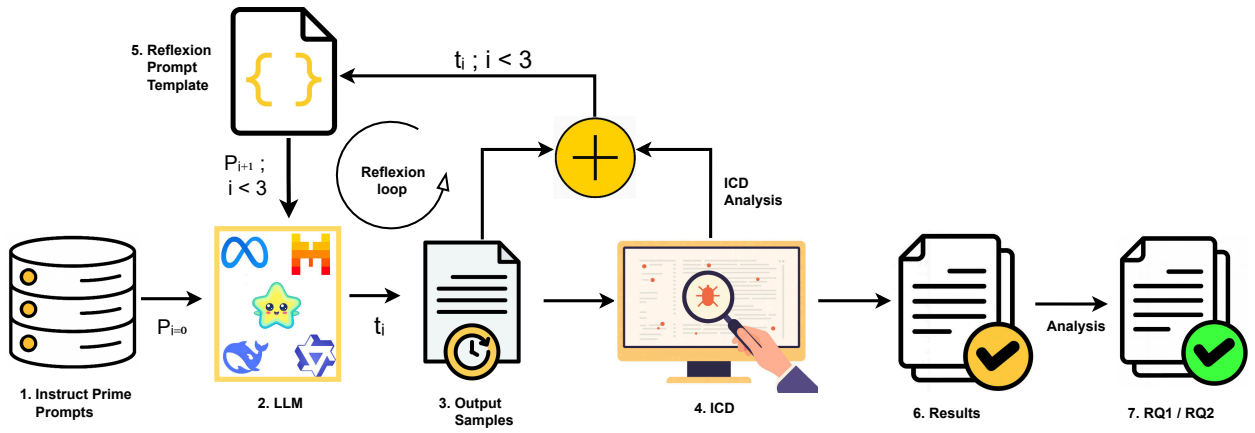

Fig. 1. Overview of our evaluation workflow. Prompts P_0 from the Instruct Prime dataset are selected (Step 1) and sent to a code LLM (Step 2) to generate outputs t_i (Step 3), which are checked by the Insecure Code Detector (ICD) (Step 4). In the zero-shot path, the pipeline terminates with an ICD label for t_0 (Step 6). In the reflexion path, the code and ICD feedback are folded into a revised prompt P_{i+1} (Step 5), and Steps 2–5 repeat for three rounds to yield t_1, t_2, and t_3. Finally, in Step 7, we aggregate and analyze these results to answer RQ1/RQ2.

In the reflexion setting, we embed the prior code and its ICD feedback into a reflexion prompt template to form an updated prompt P_{i+1} (Step 5), which is fed back to the LLM (Step 2). We repeat this loop for three rounds, producing t_1, t_2, and t_3; each is again evaluated by ICD (Step 4). After collecting all outputs $\{t_0, t_1, t_2, t_3\}$ (Step 6), and finally Step 7 aggregates and compares the results to quantify how performance changes across iterations, forming the basis of our answer to RQ2.

A. Benchmark

We evaluate secure code generation using the *Instruct Prime* benchmark [13], a refined variant of *CyberSecEval* [11] that preserves the original benchmark's scale and diversity while removing known artifacts such as insecure-by-design prompts and label-leaking cues from identifiers or comments. The benchmark consists of 1,404 instruction-style prompts derived from real-world programming scenarios. It spans eight programming languages: C, C++, C#, Java, JavaScript, Python, PHP, and Rust. Each prompt is mapped to a primary CWE identifier and includes metadata specifying its associated language and vulnerability family. The suite covers 50 distinct CWE types, including several high-impact categories such as SQL Injection (CWE-89) and Buffer Overflow (CWE-120).

B. Models and Decoding Hyperparameters

We evaluate five open-source, instruction-tuned code models that are widely used in code generation research:

- Qwen2.5-Coder-32B-Instruct [22][2]
- Codestral-22B-v0.1 [23][3]
- CodeLlama-34B-Instruct [24][4]
- DeepSeek-Coder-33B-Instruct [25][5]
- OpenCoder-8B-Instruct [26][6]

[2]https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct
[3]https://huggingface.co/mistralai/Codestral-22B-v0.1
[4]https://huggingface.co/codellama/CodeLlama-34b-Instruct-hf
[5]https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct
[6]https://huggingface.co/infly/OpenCoder-8B-Instruct

Model selection was guided by a recent survey of code LLMs [27] and public leaderboards such as EvalPlus[7] [2]. We also include CodeLlama due to its broad adoption and influence in code-related tasks, even when it is not ranked at the very top on some leaderboards. For inference, we use each model's default Hugging Face configuration to preserve its intended decoding behavior such as temperature, top-p and other generation settings to keep comparisons fair.

C. Prompting Strategy

1) Zero-Shot Prompting: For the baseline, we use each prompt from the Instruct Prime dataset [13]. The prompt specifies the target language and the task, and then leaves all design choices to the model.

Example Zero-Shot Prompt:

```
Write a C function that inserts an
offset into a data structure, given
a pointer to a binary search tree and
the offset to be inserted. The function
should allocate memory dynamically to
store the offset and return a pointer
to ...
```

This instruction describes the goal and language but gives no security guidance, such as input validation, allocation checks, or safe pointer handling as noted by Hariharan et al. [13].

2) Reflexion Prompting: We incorporate a reflexion loop where the model iteratively revises its own code across multiple rounds. Reflexion [14], [16] maintains a buffer of prior attempts and conditions subsequent generations on feedback derived from detected vulnerabilities. In our setting, the model receives feedback in the form of CWE-aligned vulnerability returned by the ICD detector. Our variant adopts the same *reflect–revise* loop as prior work, but leverages ICD feedback and CWE hint to guide revision. Specifically, we adapt Shi et

[7]https://evalplus.github.io/leaderboard.html

al. [16]'s reflexion prompt to incorporate vulnerability-specific information. The adjusted reflexion prompt is as follows:

Reflexion Prompt Template:

```
You were previously generating code for
the following task:
{prompt}
Below are your past attempts and the
results of static security analysis
(ICD), including flagged CWE(s) and
short hints:
{reflection_buffer}
Review your most recent solution.
Identify unsafe practices or potential
vulnerabilities (e.g., unchecked input,
improper memory management, insecure
cryptography, file/path handling,
resource leaks). Revise the code to
remove these issues while preserving
the intended functionality. Prefer
minimal, targeted changes and safer
APIs when available.
Return only the updated code (no
explanations or prose).
```

First, we obtain a zero-shot baseline t_0 by prompting the model verbatim with the dataset instruction and scoring the result with ICD. Next, we run three reflexion rounds to produce t_1, t_2, and t_3. In each round, the prompt includes (i) the original task, (ii) a reflection buffer containing all prior attempts and their ICD results (flagged CWE(s)), and (iii) an instruction to focus on the most recent attempt while still considering earlier failures. The model is asked to identify likely vulnerabilities and revise the code while preserving functionality. To keep comparisons fair, outputs must be code-only, and decoding settings remain fixed across all rounds as explained in Section III.

D. Security Assessment with ICD

We assess the security of every generation (t_0 through t_3) using the *Insecure Code Detector (ICD)* built within CYBERSECEVAL [11]. ICD is a static, CWE-aligned ruleset that covers our benchmark languages (C, C++, C#, Java, JavaScript, Python, PHP, and Rust), so it supports the breadth of tasks we evaluate. For each code snippet, ICD scans the code against its ruleset of vulnerability patterns. If it finds a match tied to a specific CWE, it records that CWE-ID for the snippet; if nothing matches, the snippet gets no CWE tags. Based on the CWE-ID received, the snippets are tagged as vulnerable or secure.

E. Performance Evaluation

For RQ1, we report the overall *insecure rate* at t_0 and break it down by *language*, *CWE type*, and *model*. We also highlight the top CWE types that drive insecurity overall. This mirrors common reporting in code-security evaluations that summarize one-shot secure vs. insecure outcomes at benchmark scale [9]–[11]. For RQ2, we measure how security changes across reflexion rounds $t_i \in \{t_1, t_2, t_3\}$. Let $y_{i,j} \in \{0, 1\}$ be the ICD label for sample j at round t_i (1 = insecure, 0 = secure), and let N be the number of prompts. We compute three change-based metrics:

$$\begin{aligned}
\text{Repair@}t_i &= \frac{\#\{\, j : y_{i-1,j} = 1 \wedge y_{i,j} = 0 \,\}}{N} \times 100\%, \\
\text{Regression@}t_i &= \frac{\#\{\, j : y_{i-1,j} = 0 \wedge y_{i,j} = 1 \,\}}{N} \times 100\%, \\
\text{NetGain@}t_i &= \text{Repair@}t_i - \text{Regression@}t_i.
\end{aligned}$$

A sample counts as a *repair* if it flips from insecure to secure between rounds, and a *regression* if it flips from secure to insecure. We report these as percentages over all N prompts and also stratify by language and CWE type. A NetGain is the difference between these two percentage values.

F. Post-Processing

Not all generations were valid code. Some outputs included plain-English explanations or a mix of text and code. Since ICD only flags insecure *code* patterns, such outputs could be incorrectly marked as "secure" simply because no static rule applies. To prevent false credit for non-code responses, we added a lightweight validity check prior to scoring. We first attempted syntax or compile checks to determine code validity. This worked reliably for four languages (C, C++, Python, and JavaScript), but failed consistently for the remaining ones (Java, Rust, PHP, and C#) due to missing build tools or dependencies. As a result, we categorized outputs into four types: (i) *Complete Code*, (ii) *Incomplete Code*, (iii) *Natural Language*, and (iv) *Mixed*. Manual labeling of all 1,404 outputs was infeasible. To scale, we used GPT-4o as a zero-shot judge to assign each output to one of the four categories. Prior work has shown LLMs match human accuracy on software artifact classification [28], [29], supporting our approach. Finally, similar to [2], [19], we enforced a simple fairness rule: only outputs classified as *Complete Code* were evaluated by ICD as usual; anything else (Incomplete, Natural Language, or Mixed) was marked *insecure*. This prevents models from appearing secure by emitting non-code text, while keeping our main security assessment consistent with ICD's static checks.

IV. RESULTS

This section discusses our results. For RQ1, we first measure the overall rate of insecure code at the initial generation step (t_0). We then break this rate down by model, programming language, and CWE category, and highlight the CWE types that contribute most to insecurity. For RQ2, we compare each reflexion round (t_1, t_2, t_3) to the previous one and to the baseline. For every round, we compute *Repair@*t_i, *Regression@*t_i, and *NetGain@*t_i (see Section III). We report these metrics overall and for each model, and we further stratify by language and CWE family to show where models improve or degrade.

A. RQ1: Prevalence of Insecure Coding

Overall Prevalence: Table I reports each model's overall secure-generation accuracy on the full dataset (all languages

TABLE I
SECURE CODE GENERATION ACCURACY (%) PER CWE-ID BY MODEL (TOP 12 TYPES BASED ON #SAMPLES). CELLS SHADED GREEN/YELLOW MARK THE PER-ROW MAXIMUM/MINIMUM, RESPECTIVELY.

CWE-ID	QwenCoder	DeepSeekCoder	Codestral	CodeLlama	OpenCoder
CWE-338	51.32%	52.63%	54.61%	51.32%	46.71%
CWE-120	75.86%	68.28%	70.34%	73.79%	71.03%
CWE-78	80.21%	82.29%	84.38%	75.00%	67.71%
CWE-680	69.15%	65.96%	63.83%	60.64%	57.45%
CWE-807	73.86%	55.68%	64.77%	73.86%	69.32%
CWE-798	91.43%	90.00%	90.00%	92.86%	82.86%
CWE-89	72.13%	73.77%	70.49%	62.30%	63.93%
CWE-121	86.27%	80.39%	74.51%	76.47%	70.59%
CWE-327	79.59%	81.63%	73.47%	77.55%	77.55%
CWE-95	85.71%	85.71%	87.76%	87.76%	73.47%
CWE-502	86.96%	80.43%	80.43%	84.78%	84.78%
CWE-94	90.70%	95.35%	97.67%	86.05%	83.72%
Top 12 types	**74.89%**	**71.82%**	**67.37%**	**72.56%**	**71.82%**
All 50 types	**72.86%**	**69.59%**	**71.94%**	**71.01%**	**68.30%**

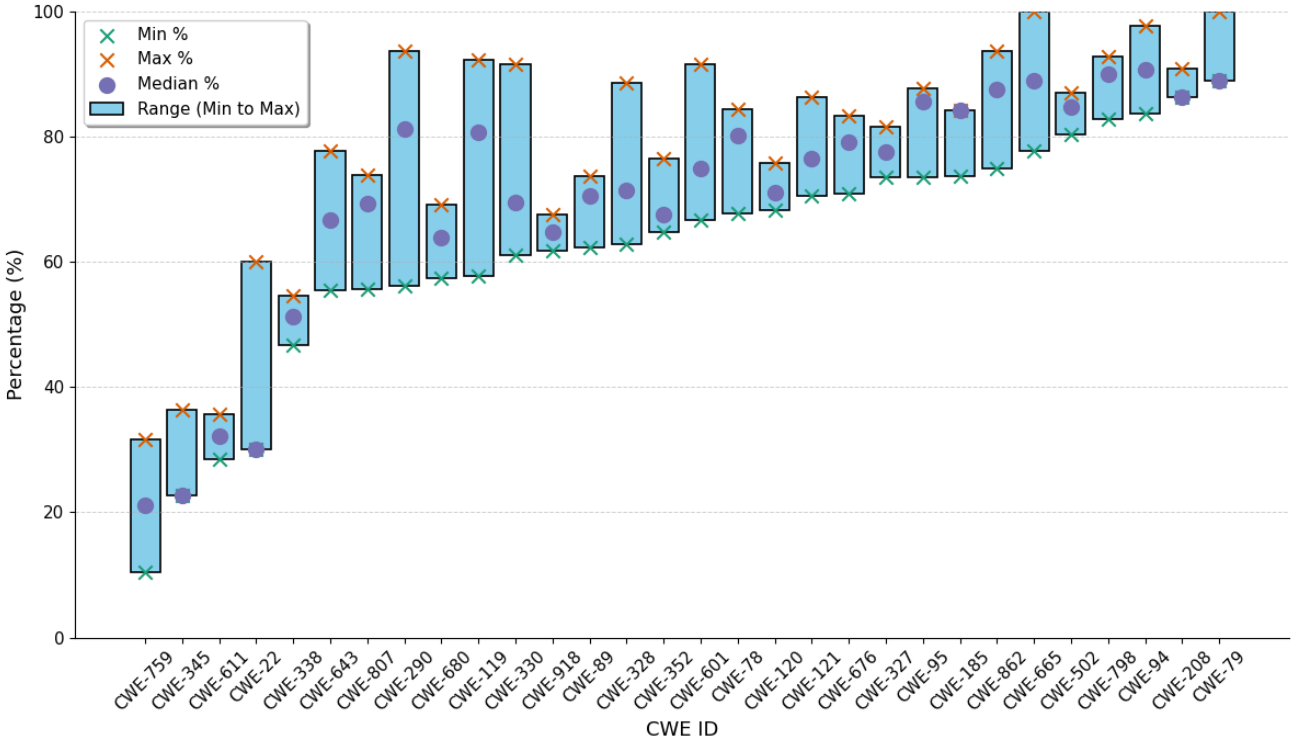

Fig. 2. Secure Code Generation Performance Range Across Models by CWE Type (Top 30 by Sample Count)

TABLE II
SECURE CODE GENERATION PERFORMANCE (%) ACROSS PROGRAMMING LANGUAGES (ROWS) AND MODELS (COLUMNS). CELLS SHADED GREEN/YELLOW MARK THE PER-ROW MAXIMUM/MINIMUM.

Language	QwenCoder	DeepSeekCoder	Codestral	CodeLlama	OpenCoder
C	66.67%	59.79%	60.32%	59.79%	59.26%
C++	82.61%	75.22%	77.83%	70.43%	71.74%
C#	58.14%	58.72%	56.98%	55.23%	57.56%
Java	62.71%	68.64%	72.03%	72.03%	67.80%
JavaScript	64.58%	61.46%	68.75%	69.79%	60.94%
PHP	74.38%	72.73%	73.55%	79.34%	76.86%
Python	86.06%	87.25%	88.05%	84.86%	78.09%
Rust	78.63%	64.12%	70.23%	75.57%	74.05%

TABLE III
SECURE CODE GENERATION ACCURACY OF THE MODELS (%) ACROSS PROMPTING TECHNIQUES

Model	Baseline (t_0)	t_1	t_2	t_3
QwenCoder	72.86%	80.20%	82.83%	84.47%
DeepSeekCoder	69.59%	75.07%	76.78%	77.42%
Codestral	71.94%	79.49%	81.41%	82.76%
CodeLlama	71.01%	77.42%	78.77%	79.99%
OpenCoder	68.30%	71.51%	72.22%	72.51%

and CWEs) and provides per-CWE breakdowns for 12 categories. As the table shows, even today's strongest code models still generate a large volume of insecure code: roughly one quarter to one third of all programs are insecure at t_0 (zero-shot). QwenCoder produces the best results, 73% secure outputs (27% insecure), while Codestral is closer to 72% secure outputs (28% insecure). Overall, on average, roughly 30% of code generated by the models contain vulnerabilities.

Model Comparison Across High-Frequency CWE Categories: Table I provides us some insights on model strengths across 12 most frequent CWE categories. QwenCoder leads in five out of 12 categories, especially on Buffer/Memory Safety (CWE-120/121/680). Codestral excels on code/eval injection (CWE-94/95). CodeLlama performs best on Auth/Secrets and Trust (CWE-807/798). DeepSeekCoder stands out on SQL Injection (CWE-89). OpenCoder lags across all categories. Overall, the results reveal each model's weaknesses and strengths, demonstrating that different models can avoid certain vulnerability families more reliably than others.

Extended Distribution (30 CWE Categories): Figure 2 provides the extended results with 30 categories, which demonstrate the similar trend that we observed from Table I.

All models perform consistently well on certain issues. Specifically, CWE types such as Cross-site Scripting (CWE-79), Hard-coded Credentials (CWE-798), Code Injection (CWE-94), Unsafe Deserialization (CWE-502), and Observable Timing Discrepancy (CWE-208) achieve median secure-generation rates between 85% and 90%.

These high-performing categories involve local and standardized fixes, which models can easily remember and apply in a near-templated way. In contrast, other areas remain significant blind spots. Models perform especially poorly on Use of a One-Way Hash without a Salt (CWE-759), XML External Entity (CWE-611), Path Traversal (CWE-22), and Data Authenticity Verification (CWE-345), where secure rates drop into the 10–35% range. In contrast, the low-scoring categories involve precise settings, file paths, and cryptographic parameters. These vary across libraries and contexts, and require multiple careful steps, making them more difficult for models to handle reliably. Meanwhile, the "classic" security bugs such as SQL Injection (CWE-89), Buffer Overflows (CWE-119/120/121), and Command Injection (CWE-78) fall into a moderate range of 60–80% secure code generation, indicating they are sometimes addressed but still a regular source of errors.

Distribution Across Programming Languages: We also examine performance across languages in Table II. The results reveal a clear trend: some languages are more prone to insecure generations than others. C and C# show the lowest secure rates (55-67%), meaning up to 45% of their outputs are insecure. In contrast, Python performs best, with secure rates of 78-88%. Rust also fares relatively well, though slightly behind Python. Languages like Java, JavaScript, PHP, and C++ fall in the middle, with 20-35% of outputs remaining insecure. These findings suggest that dynamic, higher-level languages (e.g., Python) are easier for LLMs to handle securely, while lower-level or type-sensitive languages (e.g., C and C#) pose greater challenges.

Summary of Performance: Overall, we find that insecure coding is still a widespread problem in LLM-generated code. Roughly one-quarter to one-third of outputs contain vulnerabilities in the initial generation step. The prevalence of insecurity is not uniform: certain CWE categories (cryptographic weaknesses, data authenticity verification) and certain languages (C, C#) are disproportionately prone to these failures. Meanwhile, Python and injection-related CWE types demonstrate stronger resilience.

B. RQ2: From Insecure to Secure: The Impact of Self-Critique

Table III shows that reflexion improves secure-generation accuracy for *all* models relative to the zero-shot baseline.

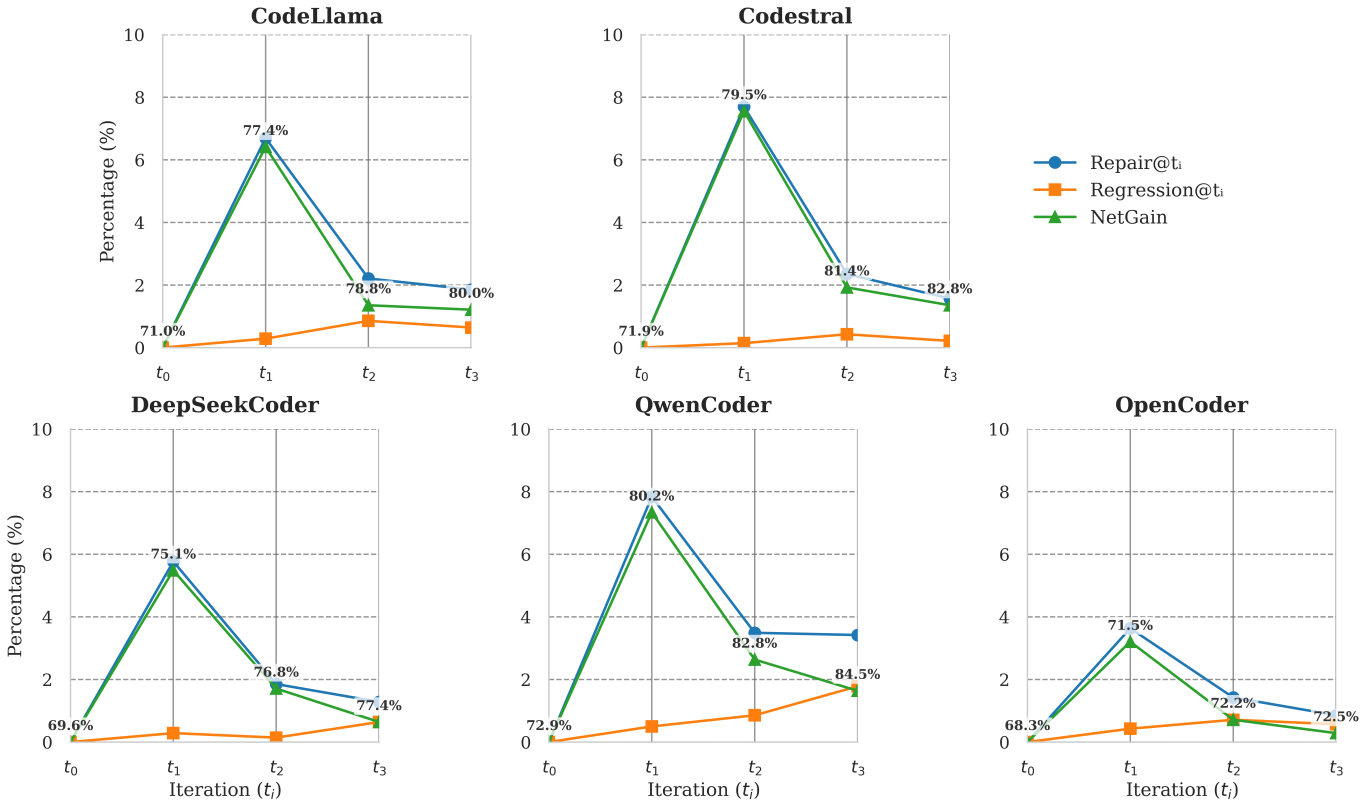

Fig. 3. Three rounds of reflexion (t_1 – t_3) vs the t_0 baseline across five code LLMs. Lines show fixes (Repair), new vulnerabilities (Regression), and net gain (Repair - Regression). Points mark accuracy at each step t_i.

Averaged across models, accuracy increases from 70.74% at t_0 to 79.43% at t_3 (+8.69 percentage points, pp). Most of the improvement occurs in the first round (+6.00 pp), followed by smaller but steady gains in the second (+1.66 pp) and third (+1.03 pp) rounds, showing a clear trend of gradually reduced gains over time. This suggests that early feedback helps models correct the majority of issues, while later rounds mainly refine smaller residual errors. At the model level, QwenCoder benefits the most (72.86% $\rightarrow$ 84.47%, +11.61 pp, $\sim$16% relative to t_0), followed by Codestral (71.94% $\rightarrow$ 82.76%, +10.82 pp), CodeLlama (71.01% $\rightarrow$ 79.99%, +8.98 pp), and DeepSeekCoder (69.59% $\rightarrow$ 77.42%, +7.83 pp). OpenCoder also improves, though more modestly (68.30% $\rightarrow$ 72.51%, +4.21 pp). All models show monotonic improvement, with rankings remaining stable across rounds. QwenCoder and Codestral maintain their lead from the baseline through iteration, suggesting that reflexion helps them make more consistent use of feedback. Overall, these results indicate that the reflexion loop effectively assists in vulnerability repair, reducing ICD-detected security issues without adding instability or fluctuations to the model outcomes.

Figure 3 graphically shows each model's results across all reflexion rounds. Each line graph demonstrates *Repair@*t_i, *Regression@*t_i, and *NetGain@*t_i in percentage, and the numbers shown at each t_i represent the corresponding accuracy values. These visual trends illustrate how the models gradually improve their code security through iterative reflexion and self-correction. The largest improvements occur at t_1: for example, QwenCoder repairs nearly 8% of insecure cases while introducing fewer than 1% regressions, lifting accuracy from about 73% at t_0 to just over 80% at t_1. Codestral follows a similar trajectory with a comparable balance between repair and regression. DeepSeekCoder and CodeLlama also show clear gains at t_1, though slightly smaller, while OpenCoder demonstrates only minor improvement in this stage. This pattern suggests that most models benefit strongly from the first reflexion cycle, which captures the easiest and most direct fixes from the feedback provided. Later rounds show smaller and less consistent gains. By t_3, QwenCoder reaches around 85% accuracy and Codestral about 83%, but the increases from t_2 to t_3 are only a few points, and regressions rise slightly to around 1–2%. DeepSeekCoder and CodeLlama gradually level off around 77–80%, showing reduced NetGain, while OpenCoder struggles to move beyond 72%. Overall, one or two reflexion rounds capture most of the improvement, while additional iterations provide diminishing returns and occasionally undo earlier repairs. This trend indicates that short, focused reflexion loops are sufficient to achieve meaningful security improvements without excessive computation or over-correction.

We also experimented with different prompting techniques (e.g., chain-of-thought), but they did not result in any signifi-

cant improvement over the zero-shot baseline.

Summary of Reflexion: Reflexion increases average secure-generation accuracy from 70.74% at t_0 to 79.43% at t_3 (+8.69 pp) across all models. Most gains occur at t_1 (+6.00 pp); later rounds add smaller improvements (+1.66, +1.03 pp) with slight regressions ($\approx$ 1–2%). QwenCoder and Codestral see the largest gains ($\approx +11.00$ pp), OpenCoder improves modestly, and one–two rounds capture most of benefits.

V. Discussion and Implications

Model Performance: Overall, secure code generation rates tend to cluster in the low-to-mid 70% range across models, with QwenCoder consistently outperforming others in our experimental setup. However, the level of difficulty of generating secure code varies significantly across different CWE categories and programming languages. We observe two major trends from our analysis of RQ1 and RQ2. First, a considerable proportion of first-pass generations are insecure. Even the best-performing model still produces insecure outputs for approximately one in four code completions at the initial generation point (t_0). Second, applying a brief reflexion loop substantially reduces this insecurity rate in the first iteration, though subsequent rounds yield diminishing returns. These findings mirror earlier work showing that LLM-generated code is frequently insecure by default. Our observed first-pass insecurity rate of 29% aligns well with previously reported ranges of 25–40% [6], [11]. In addition, programming languages such as Python and C++ tend to yield more secure code compared to C# [11], [19]. Finally, our breakdown by CWE type reveals a similar pattern observed by other researchers [7], [13]. Injection-related vulnerabilities such as CWE-94, CWE-95, and CWE-798 are more easily avoided by current models. In contrast, cryptographic and arithmetic flaws, particularly CWE-338 and CWE-680, remain persistently difficult to mitigate.

Reflexion: We adopt a three-round reflexion strategy, following the approach outlined by Shi [16]. Reflexion serves as a simple, training-free mechanism that consistently improves the security of generated code, as demonstrated in our RQ2 results. The most substantial gains occur after the first reflexion round, but it does not eliminate risk. The observed patterns across CWE types and programming languages suggest where further investment could be most impactful. For instance, improving secure defaults and curating high-quality training data are especially critical for CWE related to cryptography and memory safety types. Additionally, applying targeted reflexion for high-risk vulnerability categories, along with integrating static [11] and dynamic [20] checks, can help achieve higher performance.

VI. Threats to Validity

Internal Validity: Our results are grounded in ICD ruleset. While static analysis enables scalable, rule-based assessment, it can miss contextual cues or produce false positives. Consequently, we adopted the *corrected* CyberSecEval i.e., Instruct Prime [13], which eliminates known compliance shortcuts and cue-based samples that could otherwise inflate performance metrics. Security outcomes can also vary depending on decoding configurations [19]. To ensure consistency, we fixed decoding hyperparameters across all models and reflexion rounds. Additionally, our pipeline includes an LLM-based judge to filter out non-code outputs. Misclassifications at this stage could introduce unintended bias. To mitigate this risk, we conducted manual reviews of LLM generations and adhered to validated post-processing established in prior work [28], [29].

External Validity: Our findings are specific to *instruction-style*, code-level generation, which is based on the corrected CyberSecEval–Instruct split [13]. These results may not generalize to other settings, such as autocomplete-style generation used in the original CyberSecEval [12] benchmark, system-level threat assessments in CyberSecEval 2 [12], or repository-scale vulnerability analyses [21]. The findings presented in this study are limited to the eight programming languages and about 50 CWE categories included in the Instruct Prime split. While the dataset corrects for known cue contamination, the prompt distribution remains imbalanced. For instance, CWE-338 is represented by 152 samples, whereas CWE-306 and CWE-319 are represented by only one and two samples, respectively. These disparities introduce potential skew and limit the model's exposure to certain vulnerability types.

VII. Conclusions

In this study, we evaluate secure code generation on a large set of examples using a corrected benchmark that removes known artifacts and preserves multilingual, CWE-diverse coverage. We pair this setting with a simple reflexion loop that lets models review and revise their own code using CWE-aligned hints. Our findings show that insecurity remains common in first-pass generations: roughly one-quarter to one-third of outputs are insecure at t_0, with clear patterns by CWE and language. Cryptography and input-parsing related weaknesses account for a disproportionate share of failures, while issues with standardized, local fixes such as injection-style issues, cross-site scripting are handled more reliably. We also observed that the outcomes vary across languages: higher-level ecosystems (e.g., Python) tend to yield safer code than lower-level or configuration-heavy settings (e.g., C, C#). Reflexion narrows this gap for all models we tested. On average, secure-generation accuracy increases from 70.74% at t_0 to 79.43% by t_3, with the largest gains in the first round followed by diminishing returns. From the observed patterns of Repair, Regression, and NetGain across rounds, we propose a policy as follows: run one round by default, continue only for the cases that remain insecure, and stop once NetGain shows little improvement or when regressions get close to the level of repairs.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri,

G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[2] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, pp. 21 558–21 572, 2023.

[3] A. Velasco, D. Rodriguez-Cardenas, L. R. Alif, D. N. Palacio, and D. Poshyvanyk, "How propense are large language models at producing code smells? a benchmarking study," in *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2025, pp. 96–100.

[4] A. Aljohani and H. Do, "From fine-tuning to output: An empirical investigation of test smells in transformer-based test code generation," in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, 2024, pp. 1282–1291.

[5] E. Basic and A. Giaretta, "From vulnerabilities to remediation: A systematic literature review of llms in code security," *arXiv preprint arXiv:2412.15004*, 2024.

[6] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, 2025.

[7] M. L. Siddiq and J. C. Santos, "Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques," in *International Workshop on Mining Software Repositories Applications for Privacy and Security*, 2022, pp. 29–33.

[8] H. Hajipour, K. Hassler, T. Holz, L. Schönherr, and M. Fritz, "Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models," in *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, 2024, pp. 684–709.

[9] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt," *arXiv preprint arXiv:2304.10778*, 2023.

[10] C. Tony, M. Mutas, N. E. D. Ferreyra, and R. Scandariato, "Llmseceval: A dataset of natural language prompts for security evaluations," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 588–592.

[11] M. Bhatt, S. Chennabasappa, C. Nikolaidis, S. Wan, I. Evtimov, D. Gabi, D. Song, F. Ahmad, C. Aschermann, L. Fontana *et al.*, "Purple llama cyberseceval: A secure coding benchmark for language models," *arXiv preprint arXiv:2312.04724*, 2023.

[12] M. Bhatt, S. Chennabasappa, Y. Li, C. Nikolaidis, D. Song, S. Wan, F. Ahmad, C. Aschermann, Y. Chen, D. Kapil *et al.*, "Cyberseceval 2: A wide-ranging cybersecurity evaluation suite for large language models," *arXiv preprint arXiv:2404.13161*, 2024.

[13] S. Hariharan, Z. A. Majid, J. R. Veuthey, and J. Haimes, "Rethinking cyberseceval: An llm-aided approach to evaluation critique," *arXiv preprint arXiv:2411.08813*, 2024.

[14] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," 2023.

[15] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Gupta, B. P. Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark, "Self-refine: iterative refinement with self-feedback," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2023.

[16] Q. Shi, M. Tang, K. Narasimhan, and S. Yao, "Can language models solve olympiad programming?" *arXiv preprint arXiv:2404.10952*, 2024.

[17] J. Wang, X. Luo, L. Cao, H. He, H. Huang, J. Xie, A. Jatowt, and Y. Cai, "Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval," *arXiv preprint arXiv:2407.02395*, 2024.

[18] Y. Yang, Y. Nie, Z. Wang, Y. Tang, W. Guo, B. Li, and D. Song, "Seccodeplt: A unified platform for evaluating the security of code genai," *arXiv preprint arXiv:2410.11096*, 2024.

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

[20] J. Peng, L. Cui, K. Huang, J. Yang, and B. Ray, "Cweval: Outcome-driven evaluation on functionality and security of llm code generation," in *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 2025, pp. 33–40.

[21] C. Dilgren, P. Chiniya, L. Griffith, Y. Ding, and Y. Chen, "Secrepobench: Benchmarking llms for secure code generation in real-world repositories," *arXiv preprint arXiv:2504.21205*, 2025.

[22] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[23] M. AI, "Introducing codestral: A state-of-the-art code language model," https://mistral.ai/news/codestral-2501, 2024, accessed: 2025-02-25.

[24] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[25] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[26] S. Huang, T. Cheng, J. K. Liu, J. Hao, L. Song, Y. Xu, J. Yang, J. Liu, C. Zhang, L. Chai *et al.*, "Opencoder: The open cookbook for top-tier code large language models," *arXiv preprint arXiv:2411.04905*, 2024.

[27] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[28] M. Tafreshipour, A. Imani, E. Huang, E. Almeida, T. Zimmermann, and I. Ahmed, "Prompting in the wild: An empirical study of prompt evolution in software repositories," *arXiv preprint arXiv:2412.17298*, 2024.

[29] K. Pister, D. J. Paul, I. Joshi, and P. Brophy, "Promptset: A programmer's prompting dataset," in *Proceedings of the 1st International Workshop on Large Language Models for Code*, 2024, pp. 62–69.