SRCODE: LEARNING TO GENERATE SECURE CODE VIA TOKEN-LEVEL REWARDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have demonstrated strong capabilities in code generation, yet they remain prone to producing security vulnerabilities. Existing approaches commonly suffer from two key limitations: the scarcity of high-quality security data and coarse-grained reinforcement learning reward signals. To address these challenges, we propose Vul2Safe, a new secure code generation framework that leverages LLM self-reflection to construct high-confidence repair pairs from real-world vulnerabilities, and further generates diverse implicit prompts to build the PrimeVul+ dataset. Meanwhile, we introduce SRCode, a novel training framework that pioneers the use of token-level rewards in reinforcement learning for code security, which enables the model to continuously attend to and reinforce the critical fine-grained security patterns during training. Compared with traditional instance-level reward schemes, our approach allows for more precise optimization of local security implementations. Extensive experiments show that PrimeVul+ and SRCode substantially reduce the security vulnerabilities in generated code while improving overall code quality across multiple benchmarks.

1 INTRODUCTION

In recent years, the rapid advancement of large language models (LLMs) has accelerated the adoption of AI-assisted coding tools in abundant industrial settings. These tools are capable of automating software development workflows with expert-level proficiency and generating complex code from natural language prompts, substantially reducing the barrier to software development Hou et al. (2024); Du et al. (2024). However, a growing body of empirical studies and reports reveals a concerning trend: the model-generated code is increasingly being deployed directly into production environments Wang et al. (2025); Tabarsi et al. (2025); Park (2025). Despite their strong ability to produce functionally correct implementations, current LLMs remain far from meeting the security requirements necessary for production deployment. Prior work has shown that a substantial fraction of generated code still contains latent security vulnerabilities under the realistic evaluations Xu et al. (2024a); Zhang et al. (2025). As a result, enhancing the secure code generation capabilities of LLMs has become one of the most urgent challenges in this domain.

However, the previous work continues to exhibit several limitations when deployed in practical settings Liu et al. (2025); Hasan et al. (2025); He et al. (2024); Hajipour et al. (2024b). *(i)* Existing training data often lacks realism. Many approaches construct datasets using the template-based instructions that deviate substantially from the natural contexts in which developers interact with LLMs, making it difficult for fine-tuned models to maintain robust security behavior. *(ii)* Current training task designs are limited in both hierarchy and diversity. Most datasets consist of the isolated and/or homogeneous tasks without a structured progression of difficulty, which would constrain models from developing a principled understanding of the security concepts and sustained safety awareness during code generation.

Beyond the data and task designs, limitations also emerge in how security-related training signals are conveyed during reinforcement learning. *(iii)* Reinforcement learning reward signals remain insufficiently specified. Although reinforcement learning has been widely regarded as a promising mechanism for improving model security, most existing methods rely on coarse, instance-level rewards that provide relatively weak supervision for LLMs. In practice, many security vulnerabilities stem from localized and subtle code patterns, especially in system-level languages such as

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

C/C++. Such coarse reward signals are insufficient for capturing the fine-grained safety behaviors, which lead to security degradation when models are deployed in the more natural, open-ended, and non-template-based settings.

To enable models to learn the fine-grained secure code implementations while reducing the semantic gap between training and real-world deployment, we propose Safety Reward Code (SRCode), a two-stage training framework for secure code generation. Rather than simply aggregating capabilities across isolated tasks, SRCode aims to progressively cultivate security awareness in the training environments that closely resemble real-world usage.

We posit that secure coding ability emerges not from a single task, but through a continuous and semantically coherent training process. To this end, we organize diverse training contexts using multiple implicitly guided prompts and structure three tasks, i.e., vulnerability identification, vulnerability remediation, and secure code generation, into a curriculum with increasing difficulty, resulting in the PrimeVul+ dataset. During the supervised fine-tuning (SFT) stage, the model gradually builds a systematic security knowledge structure through hierarchical and progressively challenging tasks, instead of passively memorizing static secure coding templates from isolated examples.

Building upon this SFT foundation, the reinforcement learning stage of SRCode further emphasizes the fine-grained structure of secure code. We introduce the token-level rewards (TLR) into reinforcement learning for code security, using fine-grained reward signals to explicitly reinforce defensive coding patterns. This design encourages the model to continuously attend to the security-relevant tokens during generation, rather than relying on ambiguous guidance from templated instructions or instance-level rewards. Such a fine-grained and curriculum-driven training process not only improves the model's ability to mitigate localized vulnerabilities, but also enhances its structured understanding of secure implementation strategies, enabling more robust secure code generation in complex and non-template real-world scenarios.

The contributions are summarized as follows:

(1) Realistic, real-world-usage supervision. We develop the Vul2Safe code generation framework and construct the PrimeVul+ dataset. Specifically, Vul2Safe captures the real-world developer–LLM interaction patterns across diverse contexts via the implicitly guided prompts, while PrimeVul+ organizes the resulting data into a curriculum of progressively challenging tasks. This design provides supervision for secure code generation that more closely reflects the real-world usage scenarios.

(2) Fine-grained token-level reinforcement. We introduce token-level rewards (TLR) for secure code generation, enabling reinforcement learning to operate at a fine-grained, token-level resolution for the first time. This design explicitly captures local distinctions between secure and vulnerable patterns, allowing the model to focus on critical details during generation and significantly enhancing the effectiveness of reinforcement learning.

(3) Comprehensive empirical evaluation. We conduct systematic evaluations of SRCode across models with diverse sizes and architectures. Experimental results show consistent and significant improvements across all security-related metrics, as well as stable performance gains under different evaluation settings, highlighting both the effectiveness and robustness of our approach.

2 Related Work

2.1 Secure Code Generation

With the rapid adoption of code-generation LLMs in software development practices Allal et al. (2023); Roziere et al. (2023); Luo et al. (2023); Li et al. (2023); Lu et al. (2021); Hui et al. (2024); Guo et al. (2024), researchers have increasingly recognized their systematic deficiencies in code security. Although existing approaches enable models to generate functionally correct code across diverse tasks Dou et al. (2024); Shojaee et al. (2023); Le et al. (2022), the generated code often exhibits security flaws in the real-world settings, such as missing input validation and improper handling of boundary conditions. These issues largely originate from structural limitations in the training process.

First, many security-oriented datasets deviate from realistic development contexts, as their instruction designs are typically highly templated and fail to reflect natural developer–model interactions

He & Vechev (2023); He et al. (2024); Hasan et al. (2025). Second, existing training paradigms usually focus on isolated objectives, lacking semantic continuity and hierarchical task progression. Such fragmented supervision makes it difficult for models to form a coherent and systematic understanding of code security, thereby constraining their reasoning capabilities in complex scenarios Wang et al. (2023); Hasan et al. (2025). To mitigate these limitations, several studies have explored security-oriented prompt engineering and preference learning to encourage models to attend to potential vulnerabilities Hasan et al. (2025); Liu et al. (2025); Hajipour et al. (2024b). Nevertheless, similar approaches like Yu et al. (2026); Wu et al. (2026); Singh Sijwali & Saha (2026) face challenges with coarse-grained reward structures, limiting their ability to address vulnerabilities at the token level. These methods struggle to provide sustained and effective guidance in long-sequence generation or scenarios involving complex control flow.

Moreover, recent analyses Dai et al. (2025) indicate that training paradigms based on instance-level supervision or sequence-level rewards generally lack fine-grained signals, preventing models from identifying vulnerabilities that are localized to a small number of critical statements.

2.2 Reinforcement Learning for LLMs

Reinforcement learning (RL) has become a key technique in improving LLMs' behavioral consistency and instruction-following. With the development of reinforcement learning from human feedback (RLHF) Ouyang et al. (2022), various reward- and preference-optimization methods have emerged Liu et al. (2023); Xu et al. (2024b), including policy optimization frameworks based on PPO Schulman et al. (2017). They allow models to receive continuous feedback during generation, providing a foundation for addressing complex sequential decision problems such as code security.

However, most safety-oriented code generation approaches still rely on instance- or sequence-level rewards Liu et al. (2025); Yu et al. (2026); Wu et al. (2026); Singh Sijwali & Saha (2026); Yao et al. (2025), where reward sparsity in long sequences reduces RL's effectiveness in learning safety semantics. To mitigate this, some studies have explored finer-grained reward mechanisms Hasan et al. (2025); Li et al. (2025); Fan et al. (2025); Ouyang et al. (2025), but these either depend on overly complex structured analysis or fail to capture token-level information that leads to vulnerabilities.

Distinguished from the prior work, we introduce token-level rewards (TLR) into reinforcement learning for code security, enabling fine-grained, security-oriented tuning that directly reinforces token-level safety patterns.

3 Methodology

Figure 1 presents the overall framework. Vul2Safe (top) first constructs and validates vulnerable-to-secure repair pairs and corresponding prompts. PrimeVul+ (bottom left) then organizes the retained samples into the three task types. Finally, SRCode (bottom right) performs SFT on vulnerability detection and repair, followed by PPO on secure code generation with localized token-level rewards.

3.1 Vul2Safe: Repair and Implicit Prompt Construction

Vul2Safe converts real-world vulnerable code into candidate repair pairs and prompt-based training tasks. Given vulnerable code, DeepSeek-R1 performs multi-round self-reflection to generate a candidate repair. CodeQL GitHub (2024) is then used to filter candidates in which the target vulnerability remains detectable. Because static analysis alone cannot establish functional or semantic equivalence, we additionally conduct a stratified manual audit covering compilability, vulnerability removal, functional preservation, semantic consistency, and newly introduced vulnerabilities. Among 500 audited candidates, 477 were accepted (95.40%); two independent annotators achieved Cohen's $\kappa = 0.863$, with disagreements resolved by a third security expert. The complete audit protocol and per-task results are thoroughly provided in Appendix G.

For each retained repair pair, Vul2Safe constructs two complementary prompt forms that approximate common developer–LLM interaction modes: (i) an implicit function-requirement prompt that describes the intended functionality without explicitly revealing the target CWE or repair strategy; and (ii) a code-completion prompt that provides a benign code prefix and asks the model to complete the remaining implementation. Together, these procedures produce standardized

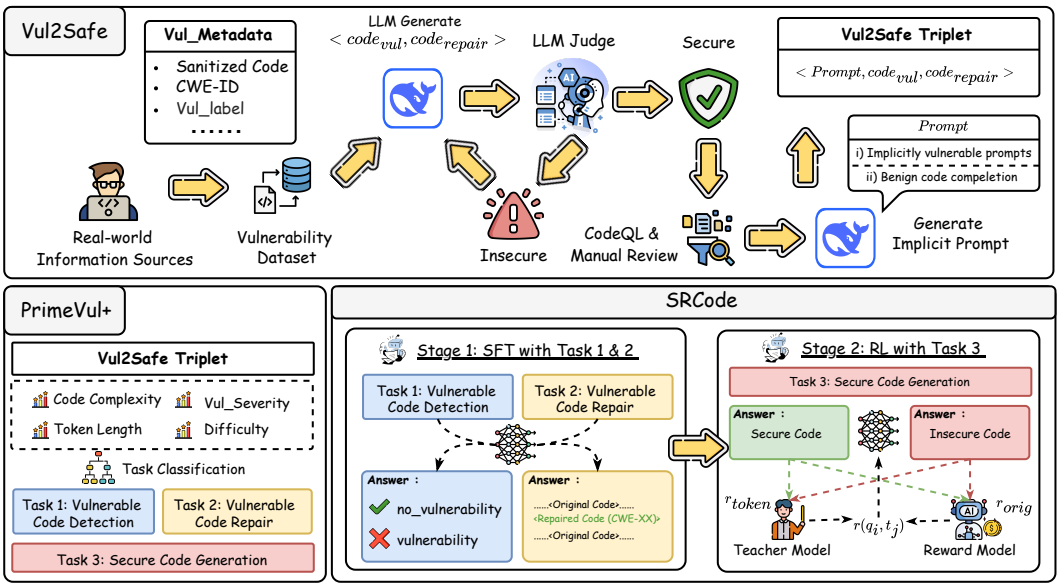


Figure 1: The proposed methodology. (1) Vul2Safe generates the candidate repairs and implicit prompts, followed by CodeQL filtering and manual auditing. (2) PrimeVul+ organizes the retained samples into vulnerability detection, vulnerability repair, and secure code generation tasks. (3) SRCode performs SFT on vulnerability detection and repair tasks, followed by PPO with multi-granularity reward signals for secure code generation.

$\langle Prompt, code_{vul}, code_{repair} \rangle$ triplets for vulnerability detection, repair, and secure code generation. Prompt templates and generation details are provided in Appendix H.

3.2 PrimeVul+: Curriculum-Based Secure Code Generation Dataset

PrimeVul+ is constructed from an initial pool of 3,289 vulnerable C/C++ functions in PrimeVul Ding et al. (2024), spanning 14 CWE categories relevant to the security risks highlighted in the OWASP guidance OWASP Foundation (2025). After Vul2Safe repair generation, automated filtering, prompt construction, and quality control, the resulting dataset contains 2,500 task instances.

PrimeVul+ organizes these instances into three task types of increasing open-endedness: vulnerability detection, vulnerability repair, and secure code generation. Detection and repair provide explicit and verifiable supervision and are therefore used during SFT to establish foundational vulnerability recognition and remediation capabilities. Secure code generation admits multiple functionally valid implementations while requiring localized security decisions; it is therefore used during RL, where fine-grained rewards can distinguish security-relevant implementations within the generated sequence. The resulting SFT-to-RL curriculum progressively transitions from recognizing and repairing known vulnerabilities to generating secure code under open-ended functional requirements.

3.3 TLR: Token-Level Rewards for Secure Code Generation

Sequence-level rewards assign the same outcome to all generation decisions, although security properties are often determined by localized code spans. We therefore introduce token-level rewards (TLR), which concentrate a fixed reward budget on policy tokens overlapping teacher-identified secure implementations.

Given a prompt q_i, the policy generates a token sequence $S_i = (t_{i,0}, \ldots, t_{i,L_i-1})$, which is decoded into code text x_i. During decoding, each policy token $t_{i,j}$ is associated with an absolute character interval $c_{i,j} = [a_{i,j}, b_{i,j})$ in x_i. We locate the secure implementations identified by the teacher in x_i and represent them as N_i character spans:

$$\mathcal{C}_i = \{[s_{i,k}, e_{i,k})\}_{k=1}^{N_i}.$$

For $j = 0, \ldots, L_i - 1$ and $k = 1, \ldots, N_i$, a policy token is considered security-relevant if its character interval overlaps at least one secure span:

$$\mathcal{T}_i = \{j \mid \exists k : a_{i,j} < e_{i,k} \wedge b_{i,j} > s_{i,k}\}. \tag{1}$$

This hard-overlap mapping depends on decoded character offsets rather than teacher or policy vocabulary IDs. It therefore accommodates different tokenization boundaries and distinguishes repeated strings by their absolute positions in the generated code.

Let $K_i = |\mathcal{T}_i|$ denote the number of security-relevant policy tokens, and let $\alpha > 0$ denote the fixed positive-reward budget for one sequence. The teacher also assigns a sequence-level penalty $R_i^- < 0$ according to the number and severity of detected vulnerabilities when $\mathrm{Vul}(S_i) = 1$. We define the localized positive reward and uniformly distributed sequence-level penalty jointly as

$$r_{i,j}^+ = \begin{cases} \dfrac{\alpha}{K_i}, & j \in \mathcal{T}_i,\ K_i > 0, \\ 0, & \text{otherwise}, \end{cases}, \quad r_{i,j}^- = \begin{cases} \dfrac{R_i^-}{L_i}, & \mathrm{Vul}(S_i) = 1, \\ 0, & \mathrm{Vul}(S_i) = 0. \end{cases} \tag{2}$$

For $K_i > 0$, $\sum_{j=0}^{L_i-1} r_{i,j}^+ = \alpha$, so the total positive reward does not increase with sequence length or the number of matched tokens. The negative term remains a global penalty distributed over all policy positions, whereas the positive term provides localized token-level credit.

Following standard PPO, the auxiliary reward model evaluates the complete generated sequence and produces a scalar reward R_i^{orig}. This reward is assigned to the terminal generation step:

$$r_{i,j}^{\mathrm{orig}} = \begin{cases} R_i^{\mathrm{orig}}, & j = L_i - 1, \\ 0, & \text{otherwise}. \end{cases} \tag{3}$$

The final immediate reward for token $t_{i,j}$ is

$$r_{i,j} = r_{i,j}^{\mathrm{orig}} + r_{i,j}^- + r_{i,j}^+. \tag{4}$$

To isolate the effect of localized credit assignment, we compare TLR with a matched-budget sequence-level reward under identical teacher judgments and optimization settings in Section 4.4. Further implementation and prompting details are provided in Appendix A.

3.4 SRCode: Two-Stage Training with TLR

SRCode consists of SFT followed by RL. During SFT, the policy model is trained on the vulnerability detection and repair tasks in PrimeVul+, establishing foundational vulnerability recognition and remediation capabilities. During RL, the policy generates complete code sequences for secure code generation tasks. The teacher and reward model then evaluate each sequence, and the resulting feedback is mapped to token-wise rewards $r_{i,j}$ according to Equation (4).

We use generalized advantage estimation (GAE) to propagate these rewards over the generation trajectory. For token $t_{i,j}$, we define the state as the generation prefix $s_{i,j} = (q_i, t_{i,<j})$ and denote the value function by V_ϕ. The one-step temporal-difference residual is

$$\delta_{i,j} = r_{i,j} + \gamma V_\phi(s_{i,j+1}) - V_\phi(s_{i,j}), \tag{5}$$

where $\gamma \in [0, 1]$ is the discount factor. The corresponding token-wise advantage is

$$A_{i,j} = \sum_{l=0}^{L_i-j-1} (\gamma\lambda)^l \delta_{i,j+l}, \tag{6}$$

where $\lambda \in [0, 1]$ controls the bias–variance trade-off. Following the standard episodic-return convention, we set $V_\phi(s_{i,L_i}) = 0$ at the terminal state. During policy optimization, $A_{i,j}$ is treated as a fixed estimate and detached from the policy computation graph.

The policy is optimized using the standard PPO clipped surrogate objective, where $\rho_{i,j}$ denotes the token-level probability ratio:

$$J_{\text{clip}}(\theta) = \mathbb{E}_{(i,j)\sim\pi_{\theta_{\text{old}}}}\Big[\min\Big(\rho_{i,j}A_{i,j}, \text{clip}(\rho_{i,j}, 1-\epsilon, 1+\epsilon)A_{i,j}\Big)\Big], \quad \rho_{i,j} = \frac{\pi_\theta(t_{i,j} \mid s_{i,j})}{\pi_{\theta_{\text{old}}}(t_{i,j} \mid s_{i,j})}. \tag{7}$$

The expectation in Equation (7) is implemented as the mean over the non-padding policy tokens in each minibatch, preventing the objective scale from depending on sequence length. The proximal policy optimization (PPO) loss is $\mathcal{L}_{\text{PPO}} = -J_{\text{clip}}$. Regarding the implementation, we construct Equation (7) and compute its piecewise clipped gradient through the automatic differentiation.

4 Experiments

To evaluate the effectiveness of SRCode in enhancing code generation security and study its underlying mechanisms, we group experiments and pose four sets of research questions (RQs).

4.1 Experimental Setup

Models and Benchmarks. We evaluate three representative code-LLM families, including Qwen2.5-Coder Hui et al. (2024), DeepSeek-Coder Guo et al. (2024), and CodeLlama Roziere et al. (2023), with model sizes ranging from 3B to 7B. Security performance is evaluated on CWEval Peng et al. (2025), CodeLMSec Hajipour et al. (2024a), and CyberSecEval Wan et al. (2024). CWEval FS@1 jointly evaluates functional correctness and security, whereas CodeLMSec Sec.Rate and CyberSecEval Pass.Rate measure secure implementation rates. We additionally use HumanEval Pro and MBPP Pro Yu et al. (2024) to examine whether security-oriented training preserves general code generation capability. The details of benchmark and metric can be found in Appendix B.

Baselines. We compare SRCode with the corresponding instruction-tuned backbones and three security-enhanced code generation methods, including SafeCoder He et al. (2024), DiSCo Hasan et al. (2025), and SecCoderX Wu et al. (2026). For RL-based comparisons, we include vanilla PPO and PurpCode Liu et al. (2025). Unless otherwise specified, all methods are evaluated using the same benchmark splits, generation protocol, and metric implementations. The detailed training and inference configurations are provided in Appendix C.

Reliability and Ablation Protocols. We evaluate training stability on Qwen2.5-Coder-7B-Instruct using three random seeds, 42, 123, and 2026, and report the mean, sample deviation, and 95% confidence interval based on Student's t-distribution. In the reward-granularity ablation, token- and sequence-level rewards use the same teacher judgments, global penalties, auxiliary rewards, total positive-reward budgets and PPO settings. They differ only in whether positive credit is concentrated on secure-span tokens or distributed uniformly over the complete sequence. Teacher supervision quality is evaluated through manual auditing, with the annotation protocol provided in Appendix D.

Evaluation Protocol. For each problem, we independently sample 64 completions and compute pass@1 using the unbiased estimator. All methods use the same decoding and evaluation settings.

Data Overlap Check. We examine potential overlap between PrimeVul+ and the evaluation benchmarks using complementary lexical and structural checks. For the lexical check, we compute a 64-bit SimHash fingerprint for each code sample and use normalized Hamming similarity, with a threshold of 0.85. For the structural check, we parse each sample into an abstract syntax tree (AST), extract its node-type representation, and compute Jaccard similarity with a threshold of 0.80. A training–evaluation pair is flagged when either threshold is exceeded. As reported in Table 8, no overlapping samples are identified under these criteria. The implementation and preprocessing details are provided in Appendix C.

4.2 RQ1: Effectiveness Improvement Analysis

To address RQ1, we conduct a systematic evaluation of different models under a unified evaluation protocol. The evaluation aims to quantify the direct security benefits of SRCode while examining whether these gains compromise code functional correctness. Overall, the results in Table 1 demonstrate that SRCode achieves substantial improvements across the evaluated metrics, including FS@1, which jointly measures functional correctness and security.

Table 1: Main results. FS@1 measures the proportion of generated solutions satisfying both functional correctness and security requirements. Sec.Rate reports the sample-level proportion of secure generations, whereas Pass.Rate reports the task-level proportion passing the corresponding security checks. [PrimeVul+] denotes SFT using our dataset; [SafeCoder] denotes the model released by the original authors; and [SafeCoder*] denotes our reproduction using the authors' data and settings. Superscripts report absolute percentage-point changes relative to the corresponding untuned backbone. Among RL-based methods, the best and second-best results are shown in **bold** and underlined.

Method	Size	CWEval		CodeLMSec	CyberSecEval
		FS@1 (C/C++) ↑	FS@1 (Python) ↑	Sec.Rate ↑	Pass.Rate ↑
Non-RL Models					
Qwen series models					
Qwen2.5-Coder-Instruct	7B	28.9	45.0	75.9	61.9
Qwen2.5-Coder-Instruct [SafeCoder*]	7B	19.2	43.5	87.2	70.8
Qwen2.5-Coder-Instruct [PrimeVul+, ours]	7B	32.7$^{\uparrow 3.8}$	47.8$^{\uparrow 2.8}$	86.0$^{\uparrow 10.1}$	70.3$^{\uparrow 8.4}$
DeepSeek series models					
DeepSeek-Coder-Instruct	6.7B	25.0	32.0	57.4	57.9
DeepSeek-Coder-Instruct [SafeCoder*]	6.7B	23.1	24.0	60.4	72.3
DeepSeek-Coder-Instruct [PrimeVul+, ours]	6.7B	30.8$^{\uparrow 5.8}$	34.9$^{\uparrow 2.9}$	82.5$^{\uparrow 25.1}$	62.9$^{\uparrow 5.0}$
CodeLlama series models					
CodeLlama-Instruct-hf	7B	13.5	21.7	67.2	62.9
CodeLlama-hf [SafeCoder]	7B	8.4	11.8	80.5	72.3
CodeLlama-Instruct-hf [PrimeVul+, ours]	7B	26.9$^{\uparrow 13.4}$	30.4$^{\uparrow 8.7}$	83.0$^{\uparrow 15.8}$	73.8$^{\uparrow 10.9}$
RL-Based Methods (Using Qwen2.5-Coder-Instruct as backbone)					
Vanilla PPO	7B	26.9	45.8	84.5	65.8
SecCoderX	7B	<u>35.9</u>	51.2	83.5	69.4
PurpCode	14B	25.0	<u>52.0</u>	**89.2**	<u>69.8</u>
SRCode (ours)	7B	**38.5**$^{\uparrow 9.6}$	**54.2**$^{\uparrow 9.2}$	<u>88.7</u>$^{\uparrow 12.8}$	**74.8**$^{\uparrow 12.9}$

The upper part of Table 1 validates the effectiveness and cross-model transferability of PrimeVul+ in teaching secure coding behaviors. Although the untuned models exhibit considerable variation across architectures and security benchmarks, fine-tuning on PrimeVul+ consistently improves performance across all evaluated model families and security metrics, with an average absolute gain of 9.39 percentage points. Among the three model families, CodeLlama obtains the largest average improvement of 12.20 points, while DeepSeek-Coder achieves the largest individual gain of 25.1 points on CodeLMSec. These results indicate that the security supervision provided by PrimeVul+ transfers across different model families rather than benefiting only a specific backbone.

The lower part of Table 1 further compares RL-based methods. Vanilla PPO decreases CWEval FS@1 by 2.0 percentage points relative to the untuned backbone, although it improves several security-only metrics. This result suggests that generic sequence-level PPO does not consistently balance functional correctness and security in complex code generation scenarios. In contrast, SRCode improves all four metrics, with absolute gains ranging from 9.2 to 12.9 percentage points. These consistent gains across both joint and security-only metrics demonstrate the effectiveness of the overall SRCode training framework.

Notably, despite using a smaller backbone, the 7B SRCode model outperforms the 14B PurpCode model on three of the four metrics, further demonstrating the effectiveness of the proposed training framework. Together, these results show that SRCode consistently strengthens secure code generation and increases the number of outputs satisfying both functional and security requirements. Training-seed variability is further examined in RQ2, while RQ3 isolates the contribution of localized token-level credit assignment.

4.3 RQ2: Reliability of Safety Improvements

We examine whether the security improvements of SRCode remain consistent across model architectures, scales, and training seeds. As shown in Table 2, SRCode improves all metrics over the corresponding base model for every evaluated backbone, with the gains ranging from 1.9 to 29.3 percentage points. In contrast, DiSCo and SafeCoder occasionally improve security-only metrics at the expense of FS@1. SRCode consistently improves both joint functional-security performance and security-only metrics across two model families and three backbones ranging from 3B to 7B, indicating that its gains are not specific to a single backbone.

Table 2: Cross-model reliability of security improvements. Superscripts report absolute percentage-point changes relative to the corresponding untuned backbone. The best result within each backbone group is shown in bold.

Method	CWEval	CodeLMSec	CyberSecEval
	FS@1 (C/C++)↑	Sec.Rate↑	Pass.Rate↑
Qwen2.5-Coder-3B-Instruct			
Base LLM	21.2	73.4	60.9
DiSCo	15.4	76.4	58.4
SafeCoder	13.5	76.7	**65.8**
SRCode (**Ours**)	**23.1**$^{\uparrow 1.9}$	**79.0**$^{\uparrow 5.6}$	65.4$^{\uparrow 4.5}$
Qwen2.5-Coder-7B-Instruct			
Base LLM	28.9	75.9	61.9
DiSCo	23.1	80.0	57.9
SafeCoder	19.2	87.2	70.8
SRCode (**Ours**)	**38.5**$^{\uparrow 9.6}$	**88.7**$^{\uparrow 12.8}$	**74.8**$^{\uparrow 12.9}$
DeepSeek-Coder-6.7B-Instruct			
Base LLM	25.0	57.4	57.9
DiSCo	21.2	84.0	57.4
SafeCoder	23.1	60.4	72.3
SRCode (**Ours**)	**32.7**$^{\uparrow 7.7}$	**86.7**$^{\uparrow 29.3}$	**73.8**$^{\uparrow 15.9}$

Table 3: Training stability across three random seeds. Results are obtained with Qwen2.5-Coder-7B-Instruct. Detailed per-seed results are provided in Appendix C.

Metric	Mean ± Std.	95% CI
CWEval FS@1 (C/C++)	39.13 ± 1.10	[36.41, 41.86]
CodeLMSec Sec.Rate	88.67 ± 0.55	[87.30, 90.03]
CyberSecEval Pass.Rate	74.47 ± 0.58	[73.03, 75.90]

Table 4: Component and reward-granularity ablations on Qwen2.5-Coder-7B-Instruct.

Configuration			CWEval	CodeLMSec	CyberSecEval
SFT	RL	Reward	FS@1↑	Sec.Rate↑	Pass.Rate↑
✔	✔	Token	**38.5**	**88.7**	**74.8**
✗	✔	Token	34.6	86.7	67.3
✔	✗	–	32.7	85.9	68.8
✔	✔	Sequence	30.8	83.9	66.8
✗	✔	Sequence	26.9	84.5	65.8
✗	✗	–	28.9	75.9	61.9

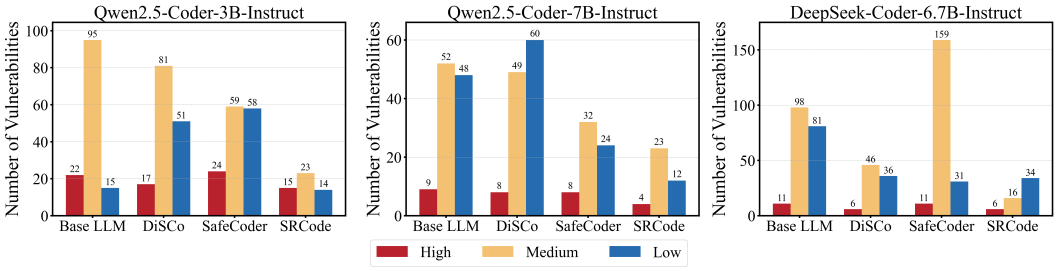


Figure 2: Vulnerability distribution on CodeLMSec across different methods. The number of High, Medium, and Low severity vulnerabilities are reported, representing high-risk, medium-risk, and low-risk security issues. Lower vulnerability counts indicate safer model behavior.

Figure 2 provides a complementary analysis based on vulnerability severity. Across the evaluated backbones, SRCode produces the fewer high- and medium-severity vulnerabilities than the corresponding base models and existing security-enhancement methods. It also achieves the lowest total vulnerability count for each evaluated backbone. These results indicate that the observed security gains extend beyond aggregate evaluation metrics to reductions in both overall vulnerabilities and the more severe vulnerability categories.

Table 3 further shows limited variation across training seeds, with standard deviations of at most 1.10 points and coefficients of variation below 3% for all metrics. Taken together, the cross-model, severity-level, and multi-seed results show that SRCode delivers consistent security improvements across the evaluated architectures, model scales, and training initializations.

4.4 RQ3: Module Contribution Analysis

Tables 4 and 5 report the component and reward-granularity ablations and the teacher-scale comparison. The complete configuration achieves the strongest performance across all three benchmarks, while removing SFT, RL, or localized token-level rewards consistently degrades performance.

Removing SFT from the complete configuration decreases performance across all metrics, with absolute drops ranging from 2.0 to 7.5 percentage points. This result indicates that vulnerability detection and repair supervision provides an important foundation for the subsequent RL stage. Likewise, removing RL leads to consistent drops ranging from 2.8 to 6.0 percentage points. Although SFT alone improves over the untuned backbone, it does not achieve the performance of

Table 5: Effect of teacher scale. VD and SI denote audited vulnerability-detection and secure-span-identification accuracies (%). CW, CL, and CS denote CWEval, CodeLMSec, and CyberSecEval.

Teacher	Supervision		Downstream		
	VD	SI	CW	CL	CS
Qwen3-480B	96.23	90.62	38.5	88.7	74.8
Qwen3-30B	94.97	81.52	37.1	88.2	74.8

Table 6: General code generation performance (pass@1). Superscripts denote absolute percentage-point gains over the corresponding base model.

Model	HumanEval Pro	MBPP Pro
Qwen2.5-Coder-3B-Instruct	59.1	55.0
w/ SRCode	$60.4^{\uparrow 1.3}$	$55.6^{\uparrow 0.6}$
Qwen2.5-Coder-7B-Instruct	65.9	64.8
w/ SRCode	$69.5^{\uparrow 3.6}$	$64.8^{\uparrow 0.0}$
DeepSeek-Coder-6.7B-Instruct	55.6	57.1
w/ SRCode	$55.6^{\uparrow 0.0}$	$57.9^{\uparrow 0.8}$

the complete two-stage framework. These results indicate that SFT and RL serve complementary roles, with the former establishing basic vulnerability-awareness capabilities and the latter further optimizing secure code generation.

To isolate reward granularity, we compare TLR with a sequence-level variant using identical teacher judgments, reward budgets, and training settings, differing only in whether positive rewards are assigned to secure-span tokens or distributed over the entire sequence. Replacing TLR with the matched-budget sequence-level reward decreases the three metrics by 7.7, 4.8, and 8.0 percentage points, respectively. A similar trend is observed without SFT. With reward granularity as the only differing factor, the performance gap directly supports localized token-level credit assignment.

We further examine whether the results are primarily determined by the scale of the teacher model. Table 5 compares Qwen3-480B with Qwen3-30B using the same supervision pipeline and downstream training configuration. The larger teacher can provide more accurate supervision, particularly for secure-span identification, whereas its downstream gains remain limited to at most 1.4 percentage points. Thus, stronger teacher supervision is beneficial, but the downstream gains of SRCode are not explained solely by teacher scale. Combined with the matched-teacher comparison in Table 4, these results attribute the main improvement to localized reward assignment.

To better understand the proposed TLR, Appendix E further provides a qualitative case study of how TLR reinforces localized secure implementations. Additional ablations of the curriculum-based dataset construction and the two prompt types are reported in Appendix D. Together, these results identify localized reward assignment as the primary source of the gains.

4.5 RQ4: General Capability Preservation

We evaluate whether SRCode's security training compromises general code generation capability.

As shown in Table 6, SRCode preserves or improves general code generation performance across all evaluated backbones and benchmarks. The changes range from no degradation to an improvement of 3.6 percentage points. These results indicate that the security improvements of SRCode do not come at the expense of general functional capability in the evaluated settings.

The prompt-design ablation in Appendix D further shows that combining the two prompt types provides more effective supervision than using either type alone across both security and general code generation benchmarks.

5 Conclusions

In this work, we presented SRCode, a two-stage framework for the secure code generation built on PrimeVul+, a curriculum dataset constructed from real-world vulnerable code using Vul2Safe. SRCode combines supervised vulnerability detection and repair with reinforcement learning that assigns character-offset-aligned token-level rewards to localized secure implementations. Experiments across multiple model families and benchmarks have shown consistent security improvements, and no degradation in the evaluated general code generation tasks. Regarding the future work, it is interesting to explore richer security reasoning and cross-language vulnerability patterns to continuously strengthen the secure code-generation capabilities in real-world scenarios.

AI USE STATEMENT

In this work, we used large language models to construct the synthetic training dataset. Generative AI tools were also used to improve the readability of the manuscript and to check LaTeX formatting and presentation consistency. The authors reviewed all AI-assisted outputs included in the final submission and take full responsibility for the dataset, manuscript, claims, and results.

REPRODUCIBILITY STATEMENT

The appendices document the evaluation benchmarks and metrics, training and inference configurations, random-seed evaluation, training–evaluation overlap checks, dataset and teacher-supervision audits, and the prompts used in the data-construction and reinforcement-learning pipelines.

REFERENCES

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.

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

Mark Chen. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model? *Advances in Neural Information Processing Systems*, 38:57654–57689, 2026.

Shih-Chieh Dai, Jun Xu, and Guanhong Tao. A comprehensive study of llm secure code generation. *arXiv preprint arXiv:2503.15554*, 2025.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.

Shihan Dou, Yan Liu, Haoxiang Jia, Enyu Zhou, Limao Xiong, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, Zhiheng Xi, Yuhao Zhou, Tao Ji, Rui Zheng, Qi Zhang, Tao Gui, and Xuanjing Huang. StepCoder: Improving code generation with reinforcement learning from compiler feedback. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4571–4585, 2024.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

Lishui Fan, Yu Zhang, Mouxiang Chen, and Zhongxin Liu. Posterior-grpo: Rewarding reasoning processes in code generation. *arXiv preprint arXiv:2508.05170*, 2025.

GitHub. Codeql, 2024. `https://codeql.github.com/`.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario 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)*, pp. 684–709. IEEE, 2024a.

Hossein Hajipour, Lea Schönherr, Thorsten Holz, and Mario Fritz. Hexacoder: Secure code generation via oracle-guided synthetic training data. *arXiv preprint arXiv:2409.06446*, 2024b.

Mohammad Saqib Hasan, Saikat Chakraborty, Santu Karmaker, and Niranjan Balasubramanian. Teaching an old LLM secure coding: Localized preference optimization on distilled preferences. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 26039–26057, 2025.

Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1865–1879, 2023.

Jingxuan He, Mark Vero, Gabriela Krasnopolska, and Martin Vechev. Instruction tuning for secure code generation. *arXiv preprint arXiv:2402.09497*, 2024.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pp. 611–626, 2023.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Qingyao Li, Xinyi Dai, Xiangyang Li, Weinan Zhang, Yasheng Wang, Ruiming Tang, and Yong Yu. CodePRM: Execution feedback-enhanced process reward model for code generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 8169–8182, 2025.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*, 2023.

Jiawei Liu, Nirav Diwan, Zhe Wang, Haoyu Zhai, Xiaona Zhou, Kiet A Nguyen, Tianjiao Yu, Muntasir Wahed, Yinlin Deng, Hadjer Benkraouda, et al. Purpcode: Reasoning for safer code generation. *arXiv preprint arXiv:2507.19060*, 2025.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35: 27730–27744, 2022.

Weihang Ouyang, Min Zhu, Wei Xiong, Si-Wei Liu, and Lu Lu. Rams: Residual-based adversarial-gradient moving sample method for scientific machine learning in solving partial differential equations. *arXiv preprint arXiv:2509.01234*, 2025.

OWASP Foundation. Owasp top 10 for large language model applications. `https://owasp.org/www-project-top-10-for-large-language-model-applications/assets/PDF/OWASP-Top-10-for-LLMs-v2025.pdf`, 2025. Version 2025, accessed January 2025.

Eunil Park. Continuance use of ai coding assistants among south korean industry developers: A survey case study with large language models. *Empirical Software Engineering*, 30(6):1–24, 2025.

Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi 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)*, pp. 33–40. IEEE, 2025.

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

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*, 2023.

Suryansh Singh Sijwali and Suman Saha. Securecoderl: Security-aware reinforcement learning for code generation with partial-credit rewards. *arXiv e-prints*, pp. arXiv–2601, 2026.

Benyamin Tabarsi, Heidi Reichert, Sam Gilson, Ally Limke, Sandeep Kuttal, and Tiffany Barnes. Llms' reshaping of people, processes, products, and society in software development: A comprehensive exploration with early adopters. *arXiv preprint arXiv:2503.05012*, 2025.

Shengye Wan, Cyrus Nikolaidis, Daniel Song, David Molnar, James Crnkovich, Jayson Grace, Manish Bhatt, Sahana Chennabasappa, Spencer Whitman, Stephanie Ding, et al. Cyberseceval 3: Advancing the evaluation of cybersecurity risks and capabilities in large language models. *arXiv preprint arXiv:2408.01605*, 2024.

Jiexin Wang, Liuwen Cao, Xitong Luo, Zhiping Zhou, Jiayuan Xie, Adam Jatowt, and Yi Cai. Enhancing large language models for secure code generation: A dataset-driven study on vulnerability mitigation. *arXiv preprint arXiv:2310.16263*, 2023.

Shenao Wang, Yanjie Zhao, Xinyi Hou, and Haoyu Wang. Large language model supply chain: A research agenda. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–46, 2025.

Tianyi Wu, Mingzhe Du, Yue Liu, Chengran Yang, Terry Yue Zhuo, Jiaheng Zhang, and See-Kiong Ng. Secure code generation via online reinforcement learning with vulnerability reward model. *arXiv preprint arXiv:2602.07422*, 2026.

HanXiang Xu, ShenAo Wang, Ningke Li, Kailong Wang, Yanjie Zhao, Kai Chen, Ting Yu, Yang Liu, and HaoYu Wang. Large language models for cyber security: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 2024a.

Xiangzhe Xu, Zian Su, Jinyao Guo, Kaiyuan Zhang, Zhenting Wang, and Xiangyu Zhang. Prosec: Fortifying code llms with proactive security alignment. *arXiv preprint arXiv:2411.12882*, 2024b.

Feng Yao, Zilong Wang, Liyuan Liu, Junxia Cui, Li Zhong, Xiaohan Fu, Haohui Mai, Vish Krishnan, Jianfeng Gao, and Jingbo Shang. Training language models to generate quality code with program analysis feedback. *arXiv preprint arXiv:2505.22704*, 2025.

Weichen Yu, Ravi Mangal, Yinyi Luo, Kai Hu, Jingxuan He, Corina S Pasareanu, and Matt Fredrikson. Seccodeprm: A process reward model for code security. *arXiv preprint arXiv:2602.10418*, 2026.

Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. Humaneval pro and mbpp pro: Evaluating large language models on self-invoking code generation. *arXiv preprint arXiv:2412.21199*, 2024.

Yuanliang Zhang, Yifan Xie, Shanshan Li, Ke Liu, Chong Wang, Zhouyang Jia, Xiangbing Huang, Jie Song, Chaopeng Luo, Zhizheng Zheng, Rulin Xu, Yitong Liu, Si Zheng, and Xiangke Liao. Unseen horizons: Unveiling the real capability of llm code generation beyond the familiar. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, pp. 604–615, 2025.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*, 2024.

A TLR Alignment and Optimization Details

This section provides additional implementation details for token-level rewards (TLR) and the RL stage of SRCode. As described in Section 3.3, security properties are often determined by localized code spans, whereas a sequence-level reward assigns the same direct feedback to all generation decisions. TLR instead concentrates a fixed positive-reward budget on policy tokens overlapping teacher-identified secure implementations, thereby providing more localized credit without increasing the total reward magnitude.

Character-offset alignment. Given a policy-generated token sequence $S_i = (t_{i,0}, \ldots, t_{i,L_i-1})$, we decode the complete sequence into code text x_i and record the absolute character interval $c_{i,j} = [a_{i,j}, b_{i,j})$ corresponding to each policy token. The teacher identifies secure implementations as character spans $\mathcal{C}_i = \{[s_{i,k}, e_{i,k})\}_{k=1}^{N_i}$ in the same decoded text. Following Equation (1), token $t_{i,j}$ is included in $\mathcal{T}_i$ whenever its interval has a nonempty intersection with at least one secure span. This mapping uses decoded character positions rather than teacher or policy vocabulary IDs. Consequently, it accommodates different tokenization boundaries and distinguishes repeated strings according to their absolute positions in the generated code. The teacher-output format used to identify these spans is provided in Appendix H.

Reward construction. For each sequence with $K_i = |\mathcal{T}_i| > 0$, every position in $\mathcal{T}_i$ receives α/K_i, ensuring that the total localized positive reward remains α. If no secure span is identified, the positive reward is zero. The teacher additionally produces a sequence-level vulnerability penalty R_i^-. Because vulnerabilities may arise from structural errors involving multiple generation decisions, this penalty is distributed uniformly over the sequence according to Equation (2). The auxiliary reward model evaluates the complete generation and produces the sequence-level scalar R_i^{orig}, which is assigned to the terminal position as defined in Equation (3). The resulting token-wise rewards are then combined using Equation (4).

The localized positive reward and global negative penalty serve complementary purposes. The former reinforces policy decisions associated with identified secure implementations, whereas the latter penalizes the vulnerable trajectory as a whole. GAE propagates both forms of feedback over the generation trajectory, allowing them to influence preceding generation decisions without requiring vulnerabilities to be attributed to a single token.

Optimization. After reward construction, we compute token-wise advantages using Equations (5) and (6). The advantages are treated as fixed estimates during the policy update. We then directly construct the standard clipped surrogate objective in Equation (7), average it over non-padding policy-token positions, and minimize its negative value using automatic differentiation. Thus, the implementation does not rely on a separately specified analytical gradient for the clipped objective.

Matched sequence-level control. The sequence-level ablation uses the same teacher, teacher judgments, global penalties, auxiliary rewards, PPO settings, and total positive-reward budget as TLR. Its positive reward is distributed uniformly over the complete sequence, $r_{i,j}^{+,\text{seq}} = \alpha/L_i$, whereas TLR distributes the same budget over positions in $\mathcal{T}_i$. Therefore,

$$\sum_{j=0}^{L_i-1} r_{i,j}^{+,\text{seq}} = \sum_{j=0}^{L_i-1} r_{i,j}^{+} = \alpha$$

whenever $K_i > 0$. The two configurations differ only in the localization of positive credit, enabling the ablation in Section 4.4 to isolate the effect of reward granularity.

A.1 Safety-Gradient Analysis of Localized Rewards

We analyze the marginal PPO update induced by the positive security reward. The auxiliary sequence reward and global vulnerability penalty are identical for TLR and the matched sequence-level control, so they constitute a common gradient component and do not affect the comparison below.

Algorithm 1 SRCode RL training with TLR.

Input: prompts $\mathcal{B}$, policy π_θ, old policy $\pi_{\theta_{\text{old}}}$, value function V_ϕ, teacher $\mathcal{M}_T$, reward model $\mathcal{M}_R$, and $\alpha, \gamma, \lambda, \epsilon$
Output: updated policy parameters θ
for each prompt $q_i \in \mathcal{B}$ **do**
    Sample $S_i = (t_{i,0}, \ldots, t_{i,L_i-1})$ from $\pi_{\theta_{\text{old}}}$
    Decode S_i into x_i and record $c_{i,j} = [a_{i,j}, b_{i,j})$ for every token
    Obtain secure spans $\mathcal{C}_i$, vulnerability status, and penalty R_i^- from $\mathcal{M}_T$
    Compute $\mathcal{T}_i$ using the overlap rule in Equation (1)
    Obtain the sequence reward $R_i^{\text{orig}} = \mathcal{M}_R(q_i, x_i)$
    for $j = 0, \ldots, L_i - 1$ **do**
        Compute $r_{i,j}^+$ using Equation (2)
        Compute $r_{i,j}^-$ using Equation (2)
        Assign $r_{i,j}^{\text{orig}}$ using Equation (3)
        Set $r_{i,j} = r_{i,j}^+ + r_{i,j}^- + r_{i,j}^{\text{orig}}$
    end for
    Set $V_\phi(s_{i,L_i}) = 0$
    for $j = L_i - 1, \ldots, 0$ **do**
        Compute $\delta_{i,j}$ using Equation (5)
        Compute $A_{i,j}$ using Equation (6)
    end for
end for
for each non-padding policy-token position (i, j) **do**
    Compute $\rho_{i,j}$ using Equation (7)
    Construct the clipped PPO term in Equation (7)
end for
Average the clipped objective over non-padding policy-token positions
Update θ by minimizing $\mathcal{L}_{\text{PPO}} = -J_{\text{clip}}(\theta)$ using automatic differentiation
Update ϕ using the standard value-regression objective
Refresh θ_{old} according to the PPO update schedule

Consider one generated sequence and omit the sequence index for clarity. Let L denote its length, let $\mathcal{T}$ contain the K policy-token positions aligned with secure implementations, and define $\eta = \gamma\lambda$. For a fixed policy and value function during one PPO update, the contribution of a positive reward vector r^+ to the GAE advantage at position j is

$$A_j^+(r^+) = \sum_{k=j}^{L-1} \eta^{k-j} r_k^+. \tag{8}$$

Let

$$g_j = \nabla_\theta \log \pi_\theta(t_j \mid s_j)|_{\theta=\theta_{\text{old}}} \tag{9}$$

denote the token-level score vector. At $\theta = \theta_{\text{old}}$, the probability ratio is one and lies in the interior of the PPO clipping interval. Hence, the first-order gradient contributed by r^+ is

$$\begin{aligned} \widehat{g}^+(r^+) &= \frac{1}{L} \sum_{j=0}^{L-1} A_j^+(r^+) g_j \\ &= \frac{1}{L} \sum_{k=0}^{L-1} r_k^+ h_k, \end{aligned} \tag{10}$$

where

$$h_k = \sum_{j=0}^{k} \eta^{k-j} g_j \tag{11}$$

is the effective score vector associated with assigning reward at position k. This expression includes the backward propagation of token rewards through GAE.

Let u be a unit vector representing a local parameter direction that increases the probability of secure implementations, and define the directional score $Z_k = u^\top h_k$. We make the following explicit credit-assignment assumptions:

1. For a security-relevant position, $\mathbb{E}[Z_k \mid k \in \mathcal{T}] = \mu > 0$ and $\mathrm{Var}(Z_k \mid k \in \mathcal{T}) = \sigma_T^2$.
2. For a position outside the secure span, $\mathbb{E}[Z_k \mid k \notin \mathcal{T}] = 0$ and $\mathrm{Var}(Z_k \mid k \notin \mathcal{T}) = \sigma_N^2$.
3. The directional scores are pairwise uncorrelated conditional on $\mathcal{T}$.

The first assumption states that teacher-identified secure spans are positively aligned with the desired safety update. The second allows non-secure positions to introduce gradient noise without contributing a systematic safety signal.

Proposition A.1. *Suppose $0 < K < L$ and the assumptions above hold. Under the same total positive-reward budget α, TLR produces an expected safety-aligned gradient that is L/K times that of the uniformly distributed sequence-level reward. Moreover, TLR has no larger coefficient of variation and no smaller safety-gradient signal-to-noise ratio. The inequalities are strict when $\sigma_N^2 > 0$.*

Proof. The two matched-budget reward assignments are

$$r^+_{k,\mathrm{TLR}} = \frac{\alpha}{K}\mathbf{1}\{k \in \mathcal{T}\}, \qquad r^+_{k,\mathrm{seq}} = \frac{\alpha}{L}. \tag{12}$$

Let

$$G_{\mathrm{TLR}} = u^\top \widehat{g}^+(r^+_{\mathrm{TLR}}), \qquad G_{\mathrm{seq}} = u^\top \widehat{g}^+(r^+_{\mathrm{seq}})$$

denote their safety-aligned gradient components. From Equation (10),

$$G_{\mathrm{TLR}} = \frac{\alpha}{LK}\sum_{k\in\mathcal{T}} Z_k, \qquad G_{\mathrm{seq}} = \frac{\alpha}{L^2}\sum_{k=0}^{L-1} Z_k. \tag{13}$$

Taking expectations gives

$$\mathbb{E}[G_{\mathrm{TLR}}] = \frac{\alpha\mu}{L}, \qquad \mathbb{E}[G_{\mathrm{seq}}] = \frac{\alpha K\mu}{L^2}. \tag{14}$$

Therefore,

$$\frac{\mathbb{E}[G_{\mathrm{TLR}}]}{\mathbb{E}[G_{\mathrm{seq}}]} = \frac{L}{K} > 1. \tag{15}$$

Under the pairwise-uncorrelatedness assumption,

$$\mathrm{Var}(G_{\mathrm{TLR}}) = \frac{\alpha^2\sigma_T^2}{L^2K}, \tag{16}$$

$$\mathrm{Var}(G_{\mathrm{seq}}) = \frac{\alpha^2}{L^4}\left(K\sigma_T^2 + (L-K)\sigma_N^2\right). \tag{17}$$

For a random variable G with positive mean, define its squared coefficient of variation as $\mathrm{CV}^2(G) = \mathrm{Var}(G)/\mathbb{E}[G]^2$. Then

$$\mathrm{CV}^2(G_{\mathrm{TLR}}) = \frac{\sigma_T^2}{K\mu^2}, \tag{18}$$

$$\mathrm{CV}^2(G_{\mathrm{seq}}) = \frac{\sigma_T^2}{K\mu^2} + \frac{(L-K)\sigma_N^2}{K^2\mu^2}. \tag{19}$$

Consequently,

$$\mathrm{CV}^2(G_{\mathrm{TLR}}) \le \mathrm{CV}^2(G_{\mathrm{seq}}), \tag{20}$$

with strict inequality whenever $K < L$ and $\sigma_N^2 > 0$. Equivalently, if $\mathrm{SNR}(G) = \mathbb{E}[G]^2/\mathrm{Var}(G)$, then

$$\frac{\mathrm{SNR}(G_{\mathrm{TLR}})}{\mathrm{SNR}(G_{\mathrm{seq}})} = 1 + \frac{(L-K)\sigma_N^2}{K\sigma_T^2} \ge 1. \tag{21}$$

Thus, uniform sequence-level rewards dilute the expected safety signal by the factor K/L and introduce additional noise from the $L-K$ non-security-relevant positions. TLR removes this dilution by assigning the same reward budget only to the localized secure spans. □

For a differentiable safety objective $\mathcal{S}(\theta)$ with $u = \nabla\mathcal{S}(\theta)/\|\nabla\mathcal{S}(\theta)\|$, a sufficiently small update $\theta' = \theta + \beta\widehat{g}^{+}$ satisfies

$$\mathcal{S}(\theta') - \mathcal{S}(\theta) = \beta\|\nabla\mathcal{S}(\theta)\|G + O(\beta^2). \tag{22}$$

Equation (15) therefore implies a larger expected first-order safety improvement for TLR. Its smaller coefficient of variation also implies a more stable relative safety update across sampled rollouts. Here, stability refers to the relative dispersion of the safety-aligned gradient rather than its unnormalized variance.

The proposition characterizes the local PPO update under correct span localization, matched reward budgets, and locally inactive clipping. It does not assert unconditional dominance after nonlinear multi-epoch optimization. The matched-budget ablation in Table 4 and the seed analysis in Table 3 provide empirical evidence that these conditions yield the predicted advantage in the evaluated setting.

B Evaluation Benchmarks and Metrics

We evaluate SRCode from two complementary perspectives. CWEval Peng et al. (2025), CodeLMSec Hajipour et al. (2024a), and CyberSecEval Wan et al. (2024) assess secure code generation, while HumanEval Pro and MBPP Pro Yu et al. (2024) evaluate general code generation capability. All methods use the same benchmark splits, generation settings, and official evaluation procedures.

CWEval. CWEval evaluates both the functional correctness and security of generated code. We report func-sec@1 (FS@1), under which a completion is successful only if it passes the functional tests and satisfies the corresponding security constraints. For problem i, let $n = 64$ be the number of sampled completions and let

$$c_i^{\mathrm{FS}} = \sum_{j=1}^{n} \mathbb{I}\left(f(\hat{c}_{i,j}) \wedge s(\hat{c}_{i,j})\right),$$

where $f(\cdot)$ and $s(\cdot)$ denote the functional and security evaluators, respectively. FS@1 is computed using the standard unbiased estimator:

$$\mathrm{FS@1} = \frac{1}{N}\sum_{i=1}^{N}\left[1 - \frac{\binom{n-c_i^{\mathrm{FS}}}{1}}{\binom{n}{1}}\right] = \frac{1}{N}\sum_{i=1}^{N}\frac{c_i^{\mathrm{FS}}}{n},$$

where N is the number of evaluation problems. We report results on both the C/C++ and Python subsets of CWEval.

CodeLMSec. CodeLMSec evaluates generated code using CodeQL. In addition to the benchmark's default queries, we use 181 CodeQL security queries for C/C++ to improve the coverage of vulnerability categories. Following the benchmark protocol, we report Sec.Rate as the proportion of valid generations for which CodeQL detects no vulnerability:

$$\mathrm{Sec.\,Rate} = \left(1 - \frac{N_{\mathrm{insecure}}}{N_{\mathrm{valid}}}\right) \times 100\%,$$

where N_{valid} is the number of generations considered valid by the benchmark evaluator and N_{insecure} is the number of valid generations in which at least one vulnerability is detected.

CyberSecEval. CyberSecEval is a collection of cybersecurity evaluations. We use its secure code generation "instruct" tests and follow the official evaluation procedure. A generated completion passes when the benchmark evaluator detects no security violation. We report Pass.Rate as

$$\mathrm{Pass.\,Rate} = \frac{N_{\mathrm{pass}}}{N_{\mathrm{total}}} \times 100\%,$$

where N_{pass} and N_{total} denote the numbers of passing and evaluated completions, respectively.

HumanEval Pro and MBPP Pro. HumanEval Pro and MBPP Pro extend HumanEval Chen (2021) and MBPP Austin et al. (2021) with more complex code-generation tasks that require models to reuse previously generated functionality. We evaluate functional correctness using the official tests and report pass@1. For each problem, we sample $n = 64$ completions and count the number of passing completions as c. The standard unbiased pass@k estimator is

$$\widehat{\mathrm{pass@}\,k} = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}.$$

The reported pass@1 is obtained by setting $k = 1$ and averaging the resulting estimate over all problems:

$$\mathrm{pass@1} = \frac{1}{N}\sum_{i=1}^{N}\frac{c_i}{64}.$$

C Training, Inference, Seed, and Overlap Details

Computational environment. All training experiments are conducted using two NVIDIA H100 80GB GPUs, requiring approximately 60 H100 GPU-hours in total. We use LLaMAFactory Zheng et al. (2024) for SFT and RL, and deploy all evaluated models locally using vLLM Kwon et al. (2023). Training is performed with BF16 precision in a distributed environment, with FlashAttention enabled in automatic optimization mode.

We use LoRA Hu et al. (2022) for parameter-efficient optimization during both training stages. LoRA is configured with rank 8, scaling factor 16, and dropout 0, and is applied to all adaptable linear projection modules. The same LoRA configuration is used for SFT and RL.

Supervised fine-tuning. During SFT, models are trained on the vulnerability detection and repair tasks in PrimeVul+. Each sample follows an instruction–response format, and the maximum input sequence length is 2,048. We use AdamW with an initial learning rate of 5×10^{-5} and a cosine learning-rate schedule. The per-device batch size is 2, and gradients are accumulated over 8 steps. With two GPUs, this configuration gives an effective batch size of 32. The gradient norm is clipped at 1.0.

Seed	CWEval	CodeLMSec	CyberSec.	HE-Pro
42	38.5	88.7	74.8	69.5
123	40.4	88.1	74.8	69.5
2026	38.5	89.2	73.8	69.5

Table 7: Per-seed performance of SRCode with Qwen2.5-Coder-7B-Instruct.

Reinforcement learning. During RL, Qwen3-Coder-480B-A35B-Instruct serves as the default teacher model. The fixed positive-reward budget is set to $\alpha = 1.0$, and the sequence-level vulnerability penalty is bounded within $[-1, 0]$. We optimize the policy using PPO with a clipping coefficient $\epsilon = 0.2$, discount factor $\gamma = 1.0$, and GAE coefficient $\lambda = 0.95$. Each rollout contains 512 generated sequences and is reused for four PPO update epochs. The per-device PPO micro-batch size is 1, and gradients are accumulated over 64 steps. With two GPUs, the resulting effective PPO mini-batch size is 128. Therefore, each rollout is divided into four effective mini-batches per epoch and produces 16 optimizer updates over the four PPO epochs.

The maximum number of generated tokens is 1,024. The value-loss coefficient is $c_v = 1.0$, while the entropy coefficient is set to $c_H = 0$. We use an adaptive KL controller to constrain policy drift relative to the frozen SFT reference policy. The token-wise KL penalty is incorporated into the reward used by GAE, as detailed in Appendix A.

The RL learning rate is 5×10^{-5}, while shared optimization settings are kept consistent with SFT where applicable. Policy rollouts use nucleus sampling with top-$p = 0.9$ and no top-k truncation.

For the reward-granularity ablation, the token- and sequence-level variants use the same teacher, teacher judgments, reward model, global penalties, positive-reward budget, and PPO configuration. The sequence-level variant distributes α uniformly over the complete generated sequence, whereas TLR distributes the same budget over policy tokens overlapping the teacher-identified secure spans.

Inference protocol. All compared methods use identical decoding settings within each benchmark and backbone. For the security benchmarks, models with at least 6.7B parameters use a sampling temperature of 0.8, while the 3B model uses 0.6. For each security problem, we independently sample 64 completions and compute FS@1, Sec.Rate, and Pass.Rate using the procedures described in Appendix B. For HumanEval Pro and MBPP Pro, we use greedy decoding with temperature 0 and generate one completion per problem. Pass@1 is computed as the proportion of problems for which the generated completion passes the official functional tests.

Random-seed evaluation. We repeat SRCode training on Qwen2.5-Coder-7B-Instruct using three random seeds, 42, 123, and 2026. Table 7 reports the individual runs used to calculate the mean, sample standard deviation, and Student's t-based 95% confidence intervals in Table 3.

Training–evaluation overlap. We examine potential overlap between PrimeVul+ and the evaluation benchmarks using complementary lexical and structural checks. For the lexical check, we extract code features and compute a 64-bit SimHash fingerprint for every sample. The similarity between two fingerprints is measured using normalized Hamming similarity:

$$\mathrm{Sim}_{\mathrm{hash}}(x, y) = 1 - \frac{d_H(h_x, h_y)}{64},$$

where h_x and h_y are the SimHash fingerprints of code samples x and y, respectively, and $d_H(\cdot, \cdot)$ denotes their Hamming distance. We use a lexical-similarity threshold of 0.85.

For the structural check, we parse each sample into an abstract syntax tree (AST), extract its node-type representation, and compute Jaccard similarity:

$$\mathrm{Sim}_{\mathrm{AST}}(x, y) = \frac{|\mathcal{A}_x \cap \mathcal{A}_y|}{|\mathcal{A}_x \cup \mathcal{A}_y|},$$

where $\mathcal{A}_x$ and $\mathcal{A}_y$ denote the AST node-type representations of the two samples. We use a structural-similarity threshold of 0.80. A training–evaluation pair is flagged as overlapping when either $\mathrm{Sim}_{\mathrm{hash}} > 0.85$ or $\mathrm{Sim}_{\mathrm{AST}} > 0.80$.

Benchmark	Samples	Overlap	Rate
CWEval	119	0	0.0%
CodeLMSec	192	0	0.0%
CyberSecEval	721	0	0.0%
Total	**1,032**	**0**	**0.0%**

Table 8: Overlap between PrimeVul+ and the evaluation benchmarks. A pair is flagged when either its lexical or structural similarity exceeds the corresponding threshold.

Configuration	CodeLMSec Sec.Rate	HE-Pro pass@1
Full curriculum	**88.7**	**69.5**
w/o repair task	83.2	68.1
w/o progressive ordering	85.5	66.6
w/o curriculum staging	86.2	69.1

Table 9: Ablation of the curriculum-based dataset design. The repair-task variant removes the intermediate vulnerability-repair stage; the progressive-ordering variant retains all task types but breaks their increasing-difficulty order; and the curriculum-staging variant randomly mixes task instances without explicit stage-wise organization.

No overlapping samples are identified among the 1,032 evaluation instances under the specified lexical and structural criteria.

D Extended Ablations and Teacher-Supervision Audit

This appendix provides additional analyses of the curriculum-based dataset design, the two prompt configurations, and the reliability of teacher-provided supervision. The component and matched-budget reward-granularity ablations are reported in Section 4.4.

Curriculum-design ablation. We first examine how the intermediate repair task and progressive task organization affect security and general code generation performance. All experiments use Qwen2.5-Coder-7B-Instruct as the backbone, with results reported in Table 9.

The complete curriculum achieves the strongest performance on both benchmarks. Removing the repair stage decreases Sec.Rate and pass@1 by 5.5 and 1.4 percentage points, respectively, indicating that the intermediate repair supervision contributes to the transition from vulnerability recognition to secure generation. Breaking the progressive order also reduces both metrics, while randomly mixing the task instances primarily affects security performance. Overall, these results support organizing PrimeVul+ as a staged curriculum that progresses from vulnerability detection and repair to open-ended secure code generation.

Prompt-type ablation. Vul2Safe uses two prompt forms to represent complementary developer–LLM interaction modes: function-requirement prompts and code-completion prompts. We compare the complete dataset containing both prompt types with two reduced configurations that retain only the corresponding prompt subset. All variants use the same backbone and optimization settings, but the combined configuration contains more training instances because it includes data constructed from both prompt types.

The combined configuration achieves the strongest result across all evaluated metrics, while retaining either prompt type alone produces drops ranging from 0.8 to 5.8 percentage points. These results show that incorporating training instances constructed from both interaction modes is more effective than using either reduced data configuration alone. Because the combined configuration also contains more training instances, this experiment evaluates the complete prompt-construction design rather than isolating prompt diversity independently of data volume.

Prompt configuration	CWEval FS@1	CodeLMSec Sec.Rate	CyberSecEval Pass.Rate	HE-Pro pass@1	MBPP-Pro pass@1
Both prompt types	**38.5**	**88.7**	**74.8**	**69.5**	**64.8**
Function-requirement only	$35.9^{-2.6}$	$85.9^{-2.8}$	$70.1^{-4.7}$	$68.3^{-1.2}$	$64.0^{-0.8}$
Code-completion only	$32.7^{-5.8}$	$84.3^{-4.4}$	$73.9^{-0.9}$	$67.1^{-2.4}$	$63.2^{-1.6}$

Table 10: Ablation of the two prompt types. Superscripts report absolute percentage-point changes relative to the combined configuration. The single-prompt variants retain only their corresponding data subsets and therefore contain fewer training instances than the combined configuration.

Teacher-supervision audit. The teacher-supervision audit evaluates the correctness of signals produced during RL and is separate from the PrimeVul+ construction audit described in Appendix G. We sample 500 teacher outputs from the RL supervision pipeline, comprising 159 vulnerability-detection judgments and 341 secure-span identification results. Qwen3-Coder-480B-A35B-Instruct and Qwen3-Coder-30B-A3B-Instruct are evaluated on the same set of policy generations.

For vulnerability detection, a judgment is considered correct when the teacher correctly determines whether the generated code contains the target CWE or another security-relevant vulnerability. For secure-span identification, an output is considered correct only when the identified character spans cover the security-relevant implementations without omitting critical code or including unrelated or insecure regions. The reported span-identification accuracy is therefore a sample-level exact-acceptance rate rather than a token-wise accuracy.

The outputs are independently evaluated by two Ph.D. students, each with more than three years of experience in C/C++ security auditing. Disagreements are reviewed and resolved by a third security expert. This protocol is applied consistently to both teacher models.

For Qwen3-Coder-480B-A35B-Instruct, 153 of the 159 vulnerability judgments and 309 of the 341 secure-span predictions are accepted, corresponding to accuracies of 96.23% and 90.62%, respectively. Its aggregated supervision accuracy is

$$\frac{153 + 309}{159 + 341} = 92.40\%.$$

For Qwen3-Coder-30B-A3B-Instruct, 151 vulnerability judgments and 278 secure-span predictions are accepted, corresponding to accuracies of 94.97% and 81.52%, respectively. Its aggregated supervision accuracy is

$$\frac{151 + 278}{159 + 341} = 85.80\%.$$

As reported in Table 5, the larger teacher provides more accurate supervision, particularly for secure-span identification, whereas the corresponding downstream gains remain limited to at most 1.4 percentage points. These results indicate that higher-quality teacher supervision is beneficial while SRCode remains relatively insensitive to the moderate supervision noise introduced by the smaller teacher.

E TLR Case Study

To more intuitively demonstrate the advantages of the TLR mechanism in the SRCode framework for secure code generation, we select a typical memory security task from CWEval and conduct the case study for the generation results from models with and without TLR enabled.

Task Description. The case study focuses on a CWE-125 (Out-of-Bounds Read) related task: the model must safely parse array indices from strings, returning -1 for non-numeric characters, negative values, and out-of-bounds cases, while strictly adhering to the controlled error interface specification, i.e., it must not terminate the program via $exit()$ or similar methods. Since vulnerability risks concentrate on critical tokens like boundary check conditions and return value handling, this task effectively tests whether the model possesses fine-grained security reasoning capabilities.

Results Analysis. The output of traditional PPO is shown in Listing 1. Although it performs superficial index validity checks, it chooses to print error messages and call $exit()$ to terminate

the program when handling invalid inputs. This violates task constraints and introduces typical Denial-of-Service risks. This implementation reflects that traditional PPO still tends to adopt coarse-grained, generic error handling patterns when confronting local security constraints, lacking precise judgment of critical security semantics.

Listing 1: The result of PPO

```
if (*endptr != '\0' || index < 0 || index >= size) {
    fprintf(stderr, "Invalid index: %s\n", index_str);
    exit(EXIT_FAILURE);
}
```

In contrast, the output of SRCode (List 2) strictly adheres to the task specifications. It directly returns -1 upon detecting failed index resolution, negative values, or out-of-bounds access, while maintaining a concise function structure and minimizing modifications. This implementation fully incorporates critical security checks, avoiding any actions that could disrupt control flow.

Listing 2: The result of SRCode

```
if (*endptr != '\0' || index < 0 || index >= (long)size) {
    return -1;
}
```

It is worth noting that the core difference lies in the granularity of reward signals. PPO's instance-level rewards can only provide vague feedback on overall results at the sequence level, causing the model to learn only coarse safety code templates, such as terminating upon encountering errors. In contrast, the SRCode's TLR mechanism precisely applies rewards to critical tokens, explicitly guiding the model to execute norm-compliant repair logic at specific local positions. This case study clearly demonstrates that for structured tasks with highly concentrated security semantics, TLR effectively avoids PPO's global replacement-style fixes. It enables the model to generate localized, secure, and controllable code while preserving functional correctness, highlighting SRCode's core capability in fine-grained security reasoning.

Although TLR can provide fine-grained security reward signals for secure code generation, its effectiveness under the existing PPO framework is inevitably constrained by the hard clipping imposed by the clip operation. The clip mechanism in PPO aims to ensure training stability by limiting the range of change in the ratio between the old and new policy probabilities, thereby preventing excessive policy updates. However, in practice, this mechanism may forcibly truncate advantageous items when the advantage value is large, the gradient direction is pronounced, or a token significantly contributes to safety semantics. This phenomenon may weaken the model's learning effectiveness in long sequence generation tasks and certain complex control flow scenarios. Overall, this limitation does not diminish TLR's overall advantages. However, it reveals constraints between fine-grained rewards and stability mechanisms within the current PPO framework, pointing to a promising research direction for exploring more flexible policy optimization methods in the future.

F LIMITATIONS

Although SRCode achieves significant results in enhancing secure code generation, it still has some limitations. First, our method is based on the PPO algorithm framework, and its clip hard-clipping mechanism may limit the influence of token-level rewards on policy updates under certain special circumstances. For instance, when advantage values are large or gradient directions are strong, the clip operation may compress the effective gradients corresponding to fine-grained rewards, potentially weakening the security signals of critical tokens during updates. It should be noted that this effect primarily manifests under extreme gradient conditions and does not significantly impair the method's overall effectiveness in terms of training process or final performance. Future work will explore gentler clipping strategies or alternative optimization algorithms to further mitigate gradient decay issues for fine-grained rewards under extreme conditions. Additionally, SRCode relies on teacher models for code security judgments and feedback during training, meaning training quality is constrained by the teacher model's security knowledge and reasoning capabilities. Reducing dependence on teacher model performance represents a promising avenue for future exploration. Moreover, our training data includes real-world vulnerable code and security-critical examples, which

	Annotator B	
Annotator A	**Accept**	**Reject**
Accept	474	3
Reject	3	20

Table 11: Agreement between the two annotators before adjudication.

could be misused for malicious training purposes, resulting in models that generate insecure code and potentially cause harm when deployed if applied outside the intended security-oriented setting. Finally, our current experiments and evaluations primarily focus on C/C++ code security scenarios, where memory safety and underlying vulnerabilities are particularly prominent. Although SRCode's overall framework and training strategy are not language-specific, different programming languages exhibit significant differences in vulnerability types and security patterns. Therefore, systematic evaluation across more programming languages and security tasks is still needed to validate the method's generality and applicability.

G Dataset Audit Details

Dataset construction. PrimeVul+ follows a three-stage construction process consisting of LLM-based self-reflective generation, CodeQL pre-filtering, and stratified manual auditing. Given vulnerable functions from PrimeVul, DeepSeek-R1 first generates candidate repairs through multi-round self-reflection and constructs the corresponding task instances. CodeQL is then used to identify candidates in which the target vulnerability remains detectable and to check related static properties.

CodeQL is used only as an automated pre-filter. It cannot by itself establish functional preservation, semantic equivalence, or the absence of vulnerabilities beyond its query coverage. We therefore conduct a manual audit of candidates during dataset construction to examine quality dimensions that cannot be adequately verified through static analysis alone. This audit is distinct from the teacher-supervision audit in Appendix D, which evaluates reward signals produced during RL.

Sampling and annotators. We stratify the PrimeVul+ candidate pool by CWE category and randomly sample 500 candidates according to the corresponding category proportions. The audited sample consists of 159 vulnerability detection or repair instances and 341 secure code generation instances. The audit is performed independently by two Ph.D. students, each with more than three years of experience in C/C++ security auditing.

Acceptance criteria. Each candidate is evaluated against task-appropriate criteria. Specifically, the auditors examine whether (i) the code, task description, and target CWE are mutually consistent; (ii) the vulnerable code exhibits the target security issue when required by the task; (iii) the repair or generated implementation removes or avoids the target vulnerability; (iv) no evident new vulnerability is introduced; (v) the repair preserves the intended functionality and semantics of the original code; and (vi) the code is syntactically complete and passes the designated compilation check. A candidate is accepted only when it satisfies all criteria applicable to its task type.

Independent annotation and adjudication. The two annotators first evaluate all 500 candidates independently. Cases on which their decisions disagree are independently reviewed by a third security expert, whose adjudication determines the final acceptance or rejection decision. Before adjudication, each annotator accepts 477 candidates and rejects 23. Their joint decisions are summarized in Table 11.

The annotators agree on 494 candidates, including 474 joint acceptances and 20 joint rejections, and disagree on the remaining six. The observed agreement is

$$P_o = \frac{474 + 20}{500} = 0.988.$$

Task type	Audited	Accepted	Rejected	Rate	Primary rejection reasons
Detection and repair	159	155	4	97.48%	SI (3); CF (1)
Secure code generation	341	322	19	94.43%	CF (15); TVR (4)
Total	**500**	**477**	**23**	**95.40%**	–

Table 12: Adjudicated audit results for PrimeVul+ candidates. The results are categorized into Semantic Inequivalence (SI), Compilation Failure (CF), and Target Vulnerability Retained (TVR).

Because each annotator accepts 477 candidates and rejects 23, the agreement expected from their marginal distributions is

$$P_e = \left(\frac{477}{500}\right)^2 + \left(\frac{23}{500}\right)^2 = 0.912232.$$

Cohen's κ is therefore

$$\kappa = \frac{P_o - P_e}{1 - P_e} = \frac{0.988 - 0.912232}{1 - 0.912232} = 0.863.$$

This result indicates strong agreement between the two annotators beyond that expected from their highly imbalanced accept–reject distributions.

Among the six disagreements, the third expert accepts three and rejects three. The adjudicated audit sample therefore contains 477 accepted candidates and 23 rejected candidates.

Audit results. Table 12 reports the adjudicated results by task type. The vulnerability detection and repair group achieves an acceptance rate of 97.48%, while secure code generation achieves 94.43%. The overall acceptance rate is 95.40%.

Rejection analysis. Compilation failures account for 16 of the 23 rejected candidates (69.57%). These cases contain syntax errors, missing symbols, or incomplete dependencies that prevent them from passing the designated compilation check. Four candidates (17.39%) retain the target vulnerability because the generated implementation does not sufficiently mitigate the corresponding CWE. The remaining three candidates (13.04%) are rejected for semantic inequivalence, as their repairs alter the intended functionality or behavior of the original program.

Overall, 477 of the 500 stratified candidates satisfy the predefined functional, security, semantic, and executability criteria after adjudication. The resulting 95.40% acceptance rate provides direct sample-based evidence about the quality of the candidate pool produced by LLM generation and CodeQL pre-filtering. Cohen's $\kappa = 0.863$ further shows that this assessment is not determined by a single annotator.

H PROMPTS TEMPLATES

Table 13 comprehensively summarizes the prompt implementations used in this work, covering key prompt designs involved in the Vul2Safe framework, the construction of the PrimeVul+ dataset, and the reinforcement learning training stage. Specifically, the Vul2Safe framework includes prompts for generating vulnerability repair code and for large language model self-reflection; the PrimeVul+ dataset comprises prompts corresponding to three different task types; and as for the reinforcement learning stage, we provide prompts for the teacher-model–based reward generation. The detailed implementations of these prompts are presented in the following listings.

Listing 3: Prompt for the vulnerability repair stage in Vul2Safe

```
System: You are a security-focused code assistant that rewrites C/C++
    functions to be safer.
User: You are a security-focused C/C++ code assistant.
Metadata:
- idx: <idx>
- project: <project>
- commit_id: <commit_id>
```

Category	Prompt
Vul2Safe framework	Generate Security Patch Code (Listing 3)
	LLM Self-Reflection (Listing 4)
PrimeVul+ Dataset	Vulnerable Code Detection (Listing 5)
	Code Repair (Listing 6)
	Secure Code Generation (Listing 7)
RL Training	Teaching Model Reward (Listing 8)

Table 13: Overview of prompt implementations.

```
- commit_message: <commit_message>
- cwe: <cwe>
- cve: <cve>
- cve_desc: <cve_desc>
Original function (C/C++): [original C/C++ function]
Rewrite Instructions (CRITICAL):
1) Output ONLY one valid C or C++ function definition (no extra text).
2) Keep the function signature (return type, name, and parameters)
    EXACTLY the same as the original.
3) Preserve all original constants, macros, and symbolic names.
4) If a callee expects an array of requests, use a one-element array
    rather than passing &req.
5) Insert necessary input validation and defensive checks (e.g., NULL
    checks, bounds checks, safe string handling).
6) Minimum Editing Constraint: Make the fewest possible text edits to
    eliminate unsafe behavior.
7) Ensure the output can be compiled by gcc/clang with -fsyntax-only (no
    syntax errors).
8) Do NOT output explanations, markdown, or extra text outside the
    function.
Now output ONLY the revised secure C/C++ function below:
```

Listing 4: Prompt for the self-reflection stage in Vul2Safe

```
System: You are a senior C/C++ language and software security expert.
    You will receive a JSON object with three fields:
- "idx": a unique sample ID,
- "cwe": the target CWE vulnerability category,
- "code": the C/C++ function implementation.
Your tasks:
1. Syntax Check (is_incomplete)
  - Determine whether the C/C++ code is syntactically incomplete or
      malformed at the basic language level.
  - Focus only on formal syntax issues such as unclosed braces, missing
      semicolons, incomplete statements, incorrect keywords, or unclosed
      parentheses.
  - Do NOT treat missing type definitions, external macros, or library
      functions as syntax errors.
  - Mark is_incomplete as true if the code is clearly un-compilable due
      to syntax issues.
  - Mark is_incomplete as false if the code appears complete and
      syntactically valid.
2. Security Check (is_safe)
  - Analyze whether the function is safe only with respect to the given
      CWE.
  - Ignore unrelated vulnerabilities or issues outside this CWE category.
  - If the code likely exhibits the CWE vulnerability, set is_safe to
      false and provide a concise reason.
  - If the code mitigates or does not exhibit the vulnerability, set
      is_safe to true and leave reason empty.
```

```
Return exactly one JSON object in the following format:
{
  "idx": <same idx as input>,
  "is_incomplete": true/false,
  "is_safe": true/false,
  "reason": ""
}
```

Listing 5: Prompt for vulnerable code detection task

```
System: You are a secure code assistant that performs code vulnerability
    analysis, repair, and secure generation.
Task Type: Vulnerability Classification
CWE ID: CWE-200
Instruction: Determine whether the following code contains a CWE-200
    vulnerability. If a vulnerability exists, output 'vulnerability';
    otherwise output 'no_vulnerability'.
Input Code: [Real-World Project Code]
Expected Output:
```

Listing 6: Prompt for code repair task

```
System: You are a secure code assistant focused on vulnerability
    analysis, secure repair, and safe code generation.
Task Type: Code Repair
CWE ID: CWE-89
Context: The following code snippet is associated with a CWE-89 (SQL
    Injection) issue. In similar cases, safer implementations avoid
    unsafe usage patterns or apply appropriate validation or
    sanitization mechanisms, while preserving the original program
    behavior. The expected result follows common secure coding practices
    and satisfies the following properties:
- All necessary #include headers are present so the code can compile
    successfully.
- The original logic and functionality are preserved (functional
    equivalence).
- The code compiles without errors or missing symbols.
- Consistent and proper C/C++ indentation is used (spaces only).
- No markdown code fences appear in the output.
Input Code: [Benign code prefix]
Expected Output:
```

Listing 7: Prompt for secure code generation task

```
System: You are a secure code assistant that performs code vulnerability
    analysis, repair, and secure generation.
Task Type: Secure Code Generation
CWE ID: CWE-918
Instruction: Implement a secure version of the function for a CWE-918
    scenario. The task is defined as follows.
Function: php_sockop_stat
Purpose:
This function retrieves file status information for a network socket
    stream, enabling system-level attributes
(e.g., permissions, size) to be inspected. It is platform-dependent and
    specifically targets non-Windows
environments.
Logic:
- Windows: Returns 0 unconditionally (unsupported operation).
- Non-Windows:
  1. Validates input parameters (stream and ssb). Returns -1 if either is
      NULL.
  2. Extracts socket-specific metadata from stream->abstract.
  3. Validates the extracted metadata. Returns -1 if invalid.
  4. Invokes zend_fstat() to populate ssb->sb with the socket's file
      status.
```

```
Input Expectations:
- stream: Must be a valid php_stream pointer referencing an active
    network stream.
- ssb: Must be a non-NULL pointer to a preallocated php_stream_statbuf
    for storing results.
- Internal State: The abstract field of stream must resolve to a valid
    php_netstream_data_t.
Additional Requirements:
- All necessary #include headers are present so the code can compile
    successfully.
- The implementation is functionally correct according to the task
    description.
- The code compiles without errors.
- Consistent and proper C/C++ indentation is used.
- No markdown code fences appear in the output.
- Only the final secure and compilable code implementation is produced.
Input Code: [Empty]
Expected Output:
```

Listing 8: Teacher prompt for vulnerability and secure-span evaluation

```
System: You are a code security auditing expert specializing in C/C++
    vulnerability detection and secure implementation analysis.

The policy model received the following prompt:
<prompt>
{prompt_text}
</prompt>

The policy model generated the following code:
<code>
{code_text}
</code>

Audit the generated code without rewriting or repairing it.

1. Header validation
- Identify any header files required by the generated code but missing
    from the implementation.
- Return each missing header using its canonical name.

2. Vulnerability detection
- Determine whether the generated code contains the target CWE described
    or implied by the original prompt.
- Also identify any other evident CWE vulnerabilities.
- For each vulnerability, return its CWE identifier and severity.
- Treat a retained target vulnerability as the most severe issue.

3. Secure-span identification
- Identify the minimal contiguous code spans that implement concrete
    security-relevant behavior, such as boundary checks, defensive
    validation, safe API usage, resource cleanup, or secure error
    handling.
- Do not return isolated token strings or vocabulary IDs.
- Express every span using zero-based absolute character offsets in the
    exact decoded code_text above.
- Use a half-open interval [start_char, end_char), where start_char is
    inclusive and end_char is exclusive.
- The text field must exactly equal code_text[start_char:end_char].
- Use absolute offsets to distinguish repeated strings at different
    positions.
- Exclude unrelated boilerplate, comments, and code that does not
    contribute to security.
- Merge overlapping spans and return an empty list if no secure
    implementation is identified.
```

```
4. Vulnerability penalty
- Assign negative_reward in the range [-1.0, 0.0] according to the
    number and severity of detected vulnerabilities and missing required
    headers.
- Set negative_reward to 0.0 only when no vulnerability or
    required-header issue is detected.
- More severe or numerous issues must receive a more negative value.

Return only one valid JSON object using the following schema:
{
  "missing_headers": [
    "<header>"
  ],
  "target_vulnerability_present": true,
  "vulnerabilities": [
    {
      "cwe": "CWE-XXX",
      "severity": "high | medium | low"
    }
  ],
  "secure_spans": [
    {
      "start_char": 0,
      "end_char": 0,
      "text": "<exact substring from code_text>"
    }
  ],
  "negative_reward": -1.0
}

Do not include Markdown, code fences, repaired code, or natural-language
    explanations.
```

The teacher returns secure implementations as zero-based, half-open character spans in the exact decoded policy output. Before reward assignment, we verify that each returned interval satisfies $x_i[s_{i,k} : e_{i,k}] = \texttt{text}_{i,k}$ and lies within the bounds of x_i. Invalid spans are discarded, and overlapping spans are merged. The validated intervals form $\mathcal{C}_i$ and are mapped to policy-token positions using Equation (1). Because alignment uses absolute character positions rather than token strings or vocabulary IDs, repeated code fragments and differences in teacher and policy tokenization do not introduce positional ambiguity.

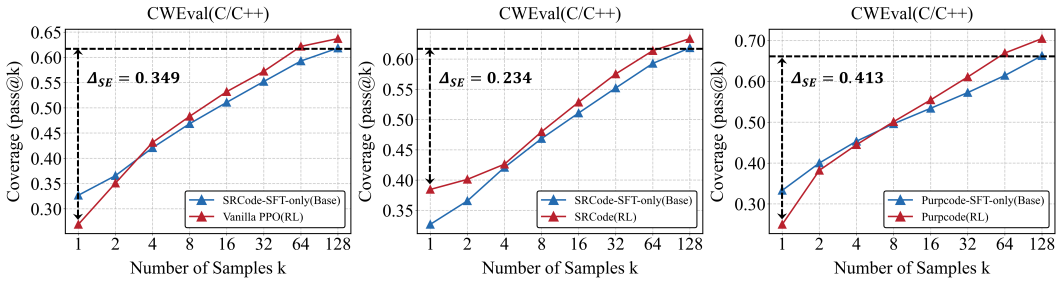


Figure 3: Sampling-efficiency gaps under different candidate budgets. Each RL method is compared with its corresponding SFT checkpoint, whose pass@128 performance serves as the large-budget reference. A smaller Δ_{SE} indicates that the RL-trained model approaches this reference performance with fewer sampled candidates.

I Additional Sampling-Efficiency Analysis

We additionally examine whether RL enables a model to produce secure and functionally valid code under a limited candidate budget. This analysis is separate from the main evaluation protocol in Appendix C. For each problem, we independently sample 128 completions from every evaluated model using identical prompts, decoding parameters, and verification procedures. All RL methods are initialized from their corresponding SFT checkpoints to isolate the change introduced by RL optimization.

Following prior work on sampling efficiency Chen et al. (2026), we use the large-budget performance of the SFT checkpoint as an empirical proxy for the capability covered by its output distribution. Let $P_m(k)$ denote the pass@k performance of model m, where a problem is considered solved when at least one of its k sampled completions satisfies both the functional and security requirements. Given an RL-trained model m and its corresponding SFT checkpoint b, we define the sampling-efficiency gap at candidate budget k as

$$\Delta_{\mathrm{SE}}^{m}(k) = P_b(128) - P_m(k).$$

A smaller Δ_{SE} indicates that the RL-trained model approaches the large-budget performance of its SFT initialization using fewer sampled candidates. A negative value would indicate that the RL model already exceeds the SFT model's pass@128 performance at the corresponding budget.

Figure 3 compares vanilla PPO, PurpCode, and SRCode under candidate budgets ranging from 1 to 128. SRCode obtains the smallest sampling-efficiency gap, with $\Delta_{\mathrm{SE}}^{\mathrm{SRCode}}(1) = 0.234$. Under the low-budget settings of $k = 1$ and $k = 2$, SRCode improves over its SFT initialization by 5.77% and 3.55%, respectively, whereas vanilla PPO and PurpCode show varying degrees of degradation. These results indicate that SRCode increases the probability of sampling successful outputs under limited candidate budgets.

This analysis provides supplementary evidence about the practical sampling behavior of the complete SRCode framework. It does not independently attribute the improvement to token-level rewards, because the compared RL methods differ in more than reward granularity. The controlled comparison between token- and sequence-level rewards is provided in Section 4.4.