Learning to Generate Secure Code via Token-Level Rewards

Anonymous ACL submission

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 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 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 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 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 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 isolated, homogeneous tasks without a structured progression of difficulty, which constrains models from developing a principled understanding of 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 is widely regarded as a promising mechanism for improving model security, most existing methods rely on coarse, instance-level rewards that provide weak supervision for LLMs. In practice, many security vulnerabilities stem from localized and subtle code patterns, especially in system-level languages such as C/C++. Such coarse reward signals are insufficient for capturing fine-grained safety behaviors, which lead to security degradation when models are deployed in more natural, open-ended, and non-template-based settings.

To enable models to learn fine-grained secure code implementations while reducing the semantic gap between training and real-world deployment, we propose 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 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 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 the PrimeVul+ dataset, which introduce real developer interaction patterns across diverse contexts into training through implicitly guided prompt design and a curriculum of progressively challenging, hierarchical tasks. This design provides supervision for secure code generation that more closely reflects 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 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 and 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, 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 (Yao et al., 2025; Liu 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 (Li et al., 2025; Hasan 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

Despite the recent advances in LLM-based code generation, secure code generation remains a significant challenge. To tackle these issues, we propose a systematic solution, as illustrated in Figure 1.

3.1 Vul2Safe: Implicitly Prompted Secure Code Generation

To systematically construct high-quality security data for training, we propose the Vul2Safe framework, which converts real vulnerability samples into highly reliable security patch corpora and evaluation tasks. Specifically, we adopt DeepSeek-R1 as the base model and leverage its self-reflection mechanism to perform multi-round generative repairs on vulnerable code. Repair quality is ensured through CodeQL (GitHub, 2024) static analysis and manual sampling audits, producing semantically consistent $< code_{vul}, code_{repair} >$ pairs.

To better reflect how developers interact with LLMs in natural contexts, Vul2Safe generates diverse prompts for each sample. These prompts are implicitly guided by DeepSeek-R1 to cover different task contexts and usage scenarios. We incorporate two strategies in the implicit prompting: (i) vulnerability-inducing prompts based on functional requirements, which embed unsafe implementation patterns in instructions to guide the model toward vulnerability-related contexts; (ii) code-completion prompts based on benign code prefixes, which present potentially risky snippets to elicit the model's understanding of vulnerability patterns during subsequent code generation. Further details can be found in Appendix H. Ultimately, Vul2Safe produces $< Prompt, code_{vul}, code_{repair} >$ triplets, automatically transforming vulnerable code into structurally standardized, task-reproducible secure repair corpora, effectively addressing the scarcity of high-quality security data.

3.2 PrimeVul+: Curriculum-Based Safe Code Generation Dataset

To support training and evaluation of LLMs in real-world code security scenarios, we construct the PrimeVul+ dataset from high-quality open-source code. Following OWASP's latest LLM security report (OWASP Foundation, 2025), we select 3,289 code samples from PrimeVul (Ding et al., 2024) covering 14 common weakness enumeration (CWE) vulnerability categories. Each sample is systematically processed using the Vul2Safe framework. After that, we evaluate and organize the tasks according to vulnerability complexity and remediation difficulty, resulting in PrimeVul+, a structured and quality-controlled dataset closely aligned with real-world LLM usage. The dataset includes three progressively challenging task categories: vulnerable code detection, vulnerable code repair, and secure code generation, totaling 2,500 tasks.

Concretely, the first two categories are used for SFT to improve the model's basic capabilities in identifying and fixing vulnerabilities. The third category, i.e., secure code generation, is designed for RL, training the model to generate robust, best-practice-compliant security implementations under

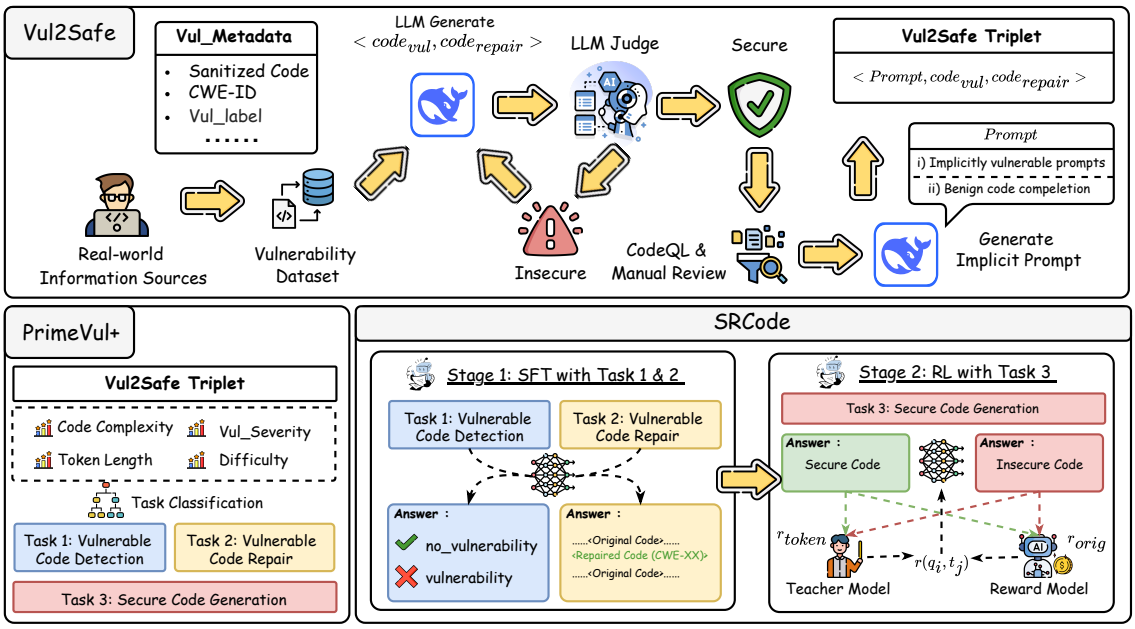

Figure 1: **Our Methodology.** (1) **Vul2Safe** transforms real-world vulnerable code into high-quality secure repair data with diverse implicit prompts. (2) **PrimeVul+** ranks the samples based on four metrics and classifies them into three progressively difficult, curriculum-style tasks. (3) **SRCode** first applies SFT on detection and repair tasks, and then performs RL on secure code generation with token-level rewards (TLR) for fine-grained safety optimization.

complex security contexts. PrimeVul+ thus provides a high-quality, scalable, and realistic foundation for training, evaluating, and benchmarking code security models.

3.3 TLR: Token-level Rewards for Safe Code Generation

In reinforcement learning (RL) driven secure code generation, traditional reward signals are typically evaluated at the code fragment level, which makes it difficult for policy models to accurately learn local secure implementations. To address this, we introduce a token-level rewards (TLR) mechanism that assigns fine-grained rewards to each token generated by the model. Specifically, for each training sample, the policy model receives an input prompt q_i and generates a code sequence $S_i = (t_0, t_1, ..., t_{L_i-1})$ of length L_i, token by token. After generating the sequence, the teacher model identifies secure implementations within the code and extracts all corresponding tokens into the secure token set T. Token-level positive rewards $r^+_{\text{token}}(q_i, t_j)$, for $j \in [0, L_i - 1]$, are then computed based on T, as defined in Equation (1).

$$r^+_{\text{token}}(q_i, t_j) = \begin{cases} 0, & t_j \notin T \\ \alpha, & t_j \in T \end{cases} \tag{1}$$

where $\alpha > 0$ represents the fixed reward for security tokens. The teacher model then analyzes whether the code contains CWE vulnerabilities. If they exist, $\text{Vul}(S_i) = True$ is set; a negative reward r_{global} is assigned based on the number and severity of vulnerabilities. This yields the negative reward $r^-_{\text{token}}(q_i, t_j)$, as depicted in Equation (2).

$$r^-_{\text{token}}(q_i, t_j) = \begin{cases} 0, & \neg\text{Vul}(S_i) \\ \frac{r_{\text{global}}}{L_i}, & \text{Vul}(S_i) \end{cases} \tag{2}$$

By combining the above process, the instantaneous reward $r(q_i, t_j)$ is obtained for each token, as shown in Equation (3), where $r_{\text{orig}}(q_i, t_j)$ represents the reward generated by reward model.

$$\begin{aligned} r(q_i, t_j) = r_{\text{orig}}(q_i, t_j) + r^-_{\text{token}}(q_i, t_j) \\ + r^+_{\text{token}}(q_i, t_j) \end{aligned} \tag{3}$$

Regarding RL policy optimization, the gradient follows the standard REINFORCE form (Williams, 1992), as described in Equation (4) below:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_j \nabla_\theta \log \pi_\theta(t_j \mid t_{<j}) A_j\right] \tag{4}$$

The advantages and further details (e.g., pseudo-code) of TLR are elaborated in Appendix A.

3.4 SRCode: A Two-Stage Training Framework Using the TLR

In this work, we propose SRCode, a two-stage training framework incorporating the TLR mechanism. During the SFT stage, we fine-tune baseline

models on the vulnerable code detection and vulnerable code repair tasks using the PrimeVul+ dataset, thereby enhancing their fundamental capabilities to detect and repair vulnerabilities. During the RL stage, the policy model generates each token in the code sequence to obtain immediate rewards as defined by Equation (3).

Upon receiving immediate rewards, the generalized advantage estimation (GAE) is employed to construct a token-level advantage function. To this end, we define the state during generation process as a prefix representation $s_{i,j} = (q_i, t_{<j})$ and denote the value network as $V_\phi(\cdot)$. The immediate reward for generating token t_j under state $s_{i,j}$ is denoted as $r_{i,j} = r(q_i, t_j)$. Based on above definitions, the one-step temporal difference residual becomes as:

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

where $\gamma \in [0,1]$ is the discount factor. After that, we employ GAE to perform weighted accumulation of residuals from future time steps, thus obtaining the advantage estimate $A_{\text{secure}}(q_i, t_j)$ for token t_j:

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

where $\lambda \in [0,1]$ controls the balance between bias and variance. At sequence termination states s_{i,L_i}, we set $V_\phi(s_{i,L_i}) = 0$ to ensure the advantage estimate converges naturally at the end. This definition provides a stable and variance-controlled estimation method for the advantage term in the subsequent PPO objective function, fully compatible with our introduced token-level fine-grained reward signal mechanism.

Based on this, we use the aforementioned advantage function $A_{\text{secure}}(q_i, t_j)$ as the core input to the PPO clip objective, with the basic form of the objective function given by Equation (7).

$$\begin{aligned} J_{\text{PPO}}^{\text{secure}} = \sum_{i=1}^{M} \sum_{j=0}^{L_i - 1} \min\big(&\rho_{i,j} \times A_{\text{secure}}(q_i, t_j), \\ &\text{clip}\left(\rho_{i,j}, 1-\varepsilon, 1+\varepsilon\right) \times A_{\text{secure}}(q_i, t_j)\big) \end{aligned} \tag{7}$$

where $\rho_{i,j}$ represents the probability ratio between the current policy and the old policy at token t_j, which corresponds to the importance sampling coefficient in the PPO framework. To prevent excessive updates, we apply clipping to this coefficient. The specific formula for the importance sampling coefficient is as follows:

$$\rho_{i,j} = \frac{P_\theta(t_j \mid s_{i,j})}{P_{\theta'}(t_j \mid s_{i,j})}, \quad s_{i,j} = (q_i, t_{<j}) \tag{8}$$

In the actual optimization process, the PPO loss function takes the negative value of the aforementioned objective function. The gradient formula (9) demonstrates that the log probability of each token is multiplied by the pruned advantage function, thereby reinforcing safe tokens and suppressing vulnerability patterns during updates.

$$\begin{aligned} \nabla_\theta \mathcal{L}_{\text{PPO}}^{\text{secure}} = -\sum_{i=1}^{M} \sum_{j=0}^{L_i - 1} \Big[&\min\Big(\rho_{i,j} \times A_{\text{secure}}(q_i, t_j), \\ &\text{clip}(\rho_{i,j}, 1-\varepsilon, 1+\varepsilon) \times A_{\text{secure}}(q_i, t_j)\Big) \\ &\times \nabla_\theta \log P_\theta(t_j \mid s_{i,j}) \Big] \end{aligned} \tag{9}$$

4 Experiments

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

RQ1: Effectiveness Improvement Analysis. *Compared to baseline methods, do our PrimeVul+ dataset and SRCode approach substantially enhance the safety of code generated by LLMs?*

RQ2: Reliability of Safety Improvements. *Can SRCode consistently deliver stable performance across models of varying scales and architectures, while uniformly reducing the number of vulnerabilities in the generated code?*

RQ3: Module Contribution Analysis. *What contribution does each module of SRCode make? Which designs are most critical to improve safety?*

RQ4: Generalization Analysis and Sampling Efficiency. *How does SRCode affect general code generation performance and sampling efficiency?*

4.1 Experienmental Setup

Base Models and Benchmarks. Three open-source large language series models: CodeLlama (Roziere et al., 2023), Qwen-Coder (Hui et al., 2024), and DeepSeek-Coder (Guo et al., 2024) are adopted for experiments. During evaluation, we mainly focus on the effectiveness of LLMs in enhancing code safety within C/C++ scenarios. Specifically, we employ three high-quality security evaluation benchmarks, i.e., CWEval (Peng et al., 2025), CodeLMSec (Hajipour et al., 2024a), and CyberSecEval (Wan et al., 2024), to ensure our results accurately and comprehensively reflect the models' security capabilities. Besides, to further gauge the model's general code generation capabilities, we employ the HumanEval Pro and MBPP

Method	Size	CWEval		CodeLMSec	CyberSecEval
		FS@1(C/C++)	FS@1(Python)	Sec.Rate	Pass.Rate
w/o 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	<u>32.7</u>$^{\uparrow 3.8\%}$	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$^{\uparrow 15.8\%}$	<u>73.8</u>$^{\uparrow 10.9\%}$
RL Models (Using Qwen2.5-Coder-Instruct as the backbone)					
Vanilla PPO	7B	26.9	45.8	84.5	65.8
Purpcode	14B	25.0	<u>52.0</u>	**89.2**	69.8
SRCode (ours)	7B	**38.5**$^{\uparrow 9.6\%}$	**54.2**$^{\uparrow 9.2\%}$	<u>88.7</u>$^{\uparrow 12.8\%}$	**74.8**$^{\uparrow 12.9\%}$

Table 1: **Main Results.** We present the results under the experimental setup described in Section 4.1. **FS@1**, i.e., func-sec@1, evaluates both the correctness and security of the code; **Sec.Rate** and **Pass.Rate** both represent the proportion of securely implemented code out of the total samples; the higher is better. [PrimeVul+] indicates the model trained using our dataset; [SafeCoder] indicates the model provided by the authors; [SafeCoder*] indicates the model trained by us using the authors' dataset and setting. We highlight the **best** and <u>second-best</u> results, and calculate the improvements of our method compared to the baselines without security optimizations.

Pro benchmarks (Yu et al., 2024). Detailed experimental settings are provided in Appendix C.

Baselines. To systematically evaluate the performance of proposed SRCode framework, we select the representative open-source instruction-fine-tuned large models, including Qwen2.5-Coder-Instruct, DeepSeek-Coder-Instruct, and CodeLlama-Instruct. In comparative experiments, we further introduce SafeCoder (He et al., 2024) and DiSCo (Hasan et al., 2025) as additional references. Regarding RL optimization methods, we choose the classic PPO approach and the recent high-quality work Purpcode (Liu et al., 2025) as baselines. These were compared with the proposed SRCode to verify whether it outperforms existing security fine-tuning methods.

Evaluation. We comprehensively evaluate the model-generated code via a combination of static analysis, execution testing, and general-purpose benchmarks to ensure the reliability of our analysis. For pass@1, we report the results using 64 independent samples. Further details of benchmarks and metrics can be found in Appendix B.

4.2 RQ1: Effectiveness Improvement Analysis

To address RQ1, we conduct a systematic evaluation of various models under identical experimental conditions. The evaluation aims to quantify the direct security benefits of the proposed SRCode method while simultaneously examining its potential impact on code functional correctness. Overall, the results, as shown in Table 1, clearly demonstrate that SRCode exhibits significant advantages across all key security metrics.

In particular, these results empirically validate the effectiveness and generalizability of the proposed PrimeVul+ dataset in guiding models to learn secure coding behaviors (see the upper part of Table 1). Specifically, without RL, significant variations in security metrics are observed across different architectures and model families. However, after fine-tuning on PrimeVul+, all models achieve consistent and substantial gains in code security, with an average increase of 9.39%. Among them, the Codellama model achieves the most pronounced average improvement of 12.20%, while the DeepSeek model demonstrates an even higher improvement of 25.1% on CodeLMSec.

After introducing RL, the methods with RL demonstrate an overall positive impact on the safety performance of baseline models (see the lower part of Table 1). However, traditional PPO exhibits a 2.0% decline in the FS@1 metric compared to the baseline, indicating that without explicit safety objective constraints, traditional PPO tends to prioritize optimizing the overall reward signal. This

tendency leads to capability degradation in complex safety scenarios. The further comparison with Purpcode reveals that its GRPO-based optimization strategy achieves commendable results across multiple metrics. Nevertheless, even against this highly competitive baseline model, SRCode demonstrates sustained and significant advantages. Notably, the 7B model trained with SRCode even outperforms the Purpcode's 14B model on nearly all security metrics. This observation implies that the performance gains primarily stem from the method's effective guidance of the model, while also enabling the model to learn security coding conventions with strong transferability.

4.3 RQ2: Reliability of Safety Improvements

In security code generation tasks, the focus is not on the model's rote memorization of specific vulnerability patterns, but rather on its ability to learn security coding behaviors with generalization capabilities. Consequently, we analyze and discuss RQ2. Under a unified training setup, we evaluate multiple models spanning different architectures and scales, with the results reported in Table 2.

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**	**79.0**	65.4
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**	**88.7**	**74.8**
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**	**86.7**	**73.8**

Table 2: **Comparison of security-enhanced code generation methods.** The best results are in bold.

The overall results show that existing security enhancement methods fail to deliver the expected security improvements in certain scenarios, even exhibiting performance degradation across multiple benchmarks. Despite incorporating substantial security examples during training, these approaches fundamentally rely on pattern-level memorization rather than semantic-level reasoning. In contrast, SRCode achieves stable and significant improvements across all models and benchmarks.

Taking the Qwen2.5-Coder-7B-Instruct as an example, it achieves improvements of 9.6%, 12.8%, and 12.9% across three metrics, representing the best results within its group. This further demonstrates that the SRCode training strategy enhances transferable and generalizable structured security reasoning capabilities, whose effectiveness is independent of specific model scale or architecture.

The in-depth analysis of vulnerability counts at different severity levels (high, medium, low) is provided in Appendix E, which confirms that SRCode delivers stable and reliable safety improvements across models of varying scales and architectures.

4.4 RQ3: Module Contribution Analysis

Table 3 presents the ablation study results. Overall, removing each module individually leads to varying degrees of performance degradation in SRCode. First, the model without SFT shows a $4 \sim 5\%$ decline across all metrics, particularly on tasks involving multi-step pointer operations, memory copying, and boundary-sensitive logic. Without this stage, models trained solely through RL generated code that is more prone to vulnerabilities in CWEval dynamic testing. This indicates that lacking SFT weakens the model's foundational understanding of secure coding conventions and significantly impacts the effectiveness of RL training.

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

Table 3: **Ablation study** on Qwen2.5-Coder-7B-Instruct. SFT refers to supervised fine-tuning on the PrimeVul+, RL refers to reinforcement learning using PPO, and TLR refers to the token-level rewards.

Second, when the RL phase is removed, the FS@1 metric of models shows a significant decline, with the synergistic performance of functionality and security markedly deteriorating, reflecting a classic alignment tax phenomenon. Performance also drops by 2.8% and 6.0% on CodeLMSec and CyberSecEval, respectively. The model first learns fundamental functional correctness on the SFT foundation while acquiring preliminary code security implementation. Crucially, it should progressively learn vulnerability-fixing strategies

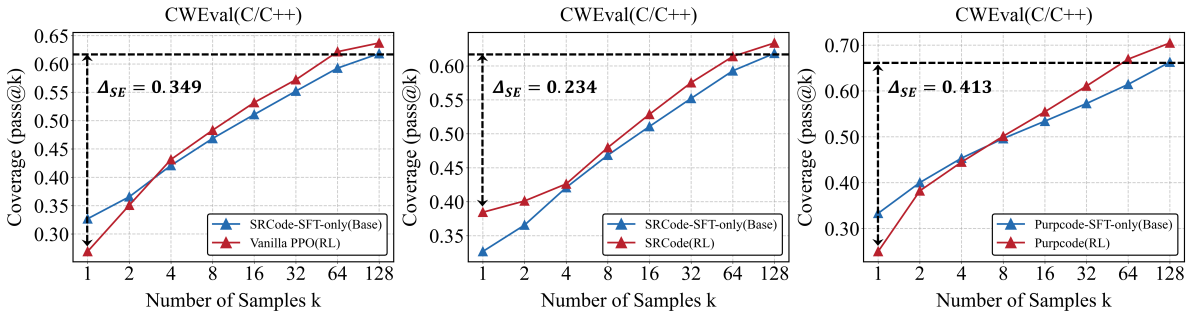

Figure 2: **Comparison of Sampling Efficiency Gaps (Δ_{SE}).** A smaller Δ_{SE} indicates that the RL-trained model is closer to the theoretical performance upper bound of the base model under large-k sampling.

during the RL phase to achieve a reliable balance between functional correctness and security.

Finally, with TLR removed, all metrics exhibit a clear decline, particularly in the model's fine-grained performance for complex security scenarios. This degradation is most pronounced in tasks involving ambiguous trigger conditions and vulnerabilities caused by the single-character errors. Appendix G further presents a case study to intuitively illustrate the advantages of TLR. Besides, more detailed analysis and the ablation results on PrimeVul+ dataset are provided in Appendix D.

4.5 RQ4: Generalization Analysis and Sampling Efficiency

To verify whether improving code security comes at the cost of utility, we evaluate SRCode over HumanEval Pro and MBPP Pro, two more comprehensive datasets for testing general code generation capabilities with self-calling and code reuse requirements, as shown in Table 4.

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
DeepSeek-Coder-6.7B-Instruct	55.6	57.1
w/ SRCode	55.6	57.9$^{\uparrow 0.8\%}$

Table 4: Evaluation of **general code generation capability** for the models with SRCode (i.e., w/SRCode).

It can be observed from Table 4 that across three base models of varying scales and architectures, SRCode does not cause any performance degradation, maintaining stable overall performance with the improvements in some cases. Concretely, Qwen2.5-Coder-7B-Instruct achieves a significant 3.6% increase on HumanEval Pro, while others also show modest gains across various benchmarks. This demonstrates that SRCode enhances model safety without compromising general capabilities, instead further strengthening generalization and robustness. Particularly for small-to-medium-sized models, SRCode's structured safety training framework delivers additional performance gains, enabling models to maintain high-quality code generation across a broader range of tasks.

To better reflect realistic user scenarios with limited sampling budgets, we further examine the sampling efficiency of SRCode. As shown in Figure 2, SRCode consistently exhibits superior sampling efficiency especially under the low-sample settings (better reflecting real-world user experience), enabling models to produce more secure and higher-quality code with fewer candidates. The detailed analysis is supplemented in Appendix F.

5 Conclusions

In this work, we first designed the Vul2Safe secure code generation framework, and then constructed the PrimeVul+ dataset for building a highly trustworthy code security corpus from real-world vulnerable code to strengthen the model's security perception at the data level. Upon this foundation, we further introduced the SRCode training framework, which pioneers the incorporation of the token-level rewards into reinforcement learning for secure code generation, effectively enhancing the model's fine-grained security reasoning capabilities.

Experiments demonstrated that our constructed data and proposed training framework significantly improved the security performance across multiple evaluation benchmarks while enhancing the overall code quality. Future work will explore richer forms of security reasoning and cross-language vulnerability patterns to continuously strengthen the security robustness in real-world scenarios.

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 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.

Acknowledgments

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, and 1 others. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

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

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

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

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. 2024. 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)*, pages 4571–4585.

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

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

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

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

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

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

Mohammad Saqib Hasan, Saikat Chakraborty, Santu Karmaker, and Niranjan Balasubramanian. 2025. 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)*, pages 26039–26057.

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

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

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

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

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*.

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

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

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

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

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

Jiawei Liu, Nirav Diwan, Zhe Wang, Haoyu Zhai, Xiaona Zhou, Kiet A Nguyen, Tianjiao Yu, Muntasir Wahed, Yinlin Deng, Hadjer Benkraouda, and 1 others. 2025. Purpcode: Reasoning for safer code generation. *arXiv preprint arXiv:2507.19060*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, and 1 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

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

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

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

OWASP Foundation. 2025. 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. Version 2025, accessed January 2025.

Eunil Park. 2025. 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.

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

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

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

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

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

Shengye Wan, Cyrus Nikolaidis, Daniel Song, David Molnar, James Crnkovich, Jayson Grace, Manish Bhatt, Sahana Chennabasappa, Spencer Whitman, Stephanie Ding, and 1 others. 2024. Cyberseceval 3: Advancing the evaluation of cybersecurity risks and capabilities in large language models. *arXiv preprint arXiv:2408.01605*.

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

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

Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256.

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

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

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

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

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. 2025. 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*, page 604–615.

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

A Methodology Details and Algorithm

In this section, we first supplement the analysis of advantages of the token-level rewards (TLR) mechanism. The necessary background and preliminaries have been introduced in Section 3.3. When $|T| \ll L_i$ (where T is the set of secure tokens and L_i is the length of generated code sequence S_i) and only sentence-level rewards are employed, all tokens share the same sequence-level reward. As a result, safety-critical tokens contribute only a negligible portion of the reward signal when computing the advantage A_j (the advantage of token t_j in sequence S_i). Their gradients are therefore significantly diluted, making it difficult to effectively reinforce safety structures within code segments. In contrast, token-level positive reinforcement can provide fine-grained, strongly directive learning signals that significantly outperform sentence-level rewards in learning local security patterns.

In secure code generation tasks, vulnerabilities are rarely caused directly by a single token but rather stem from structural errors such as missing code logic, insufficient condition checks, or improper resource usage. Such errors are generally difficult to accurately identify through local attribution to individual tokens. Therefore, when the teacher model determines that sequence S_i contains a vulnerability, we employ a negative reward r^{-}_{token} as shown in Equation (2). This design ensures that each token's contribution to the policy gradient shifts toward reducing the overall probability of the vulnerable sequence. After incorporating this reward into the advantage term, gradient updates still follow Equation (9). If vulnerabilities exist in the code sequence, the uniform distribution of negative rewards across all tokens ensures the advantage function maintains consistent sign and magnitude, thereby avoiding gradient noise and instability potentially caused by single-token negative rewards.

Based on this analysis, token-level positive rewards and sentence-level negative rewards form a complementary relationship in the optimization objective: the former aims to strengthen the correct generation of local secure code patterns, while the latter aims to globally penalize structural errors that lead to vulnerabilities. When a token belongs to a secure pattern, the advantage function is primarily determined by r^{+}_{token}, and the policy update gradient moves toward reinforcing secure tokens; When the code sequence contains vulnerabilities, the advantage function updates the gradient toward

reducing the generation probability of the entire code segment. Since the two reward types act on different levels of the code structure, this mechanism simultaneously ensures learning of local code security implementation and punishment of global vulnerability patterns. From the perspective of optimization objectives, the above mechanism can be formalized as an expectation-based optimization objective, as shown in Equation (10) below:

$$J_{\text{PPO}}^{\text{secure}} = \mathbb{E}_{(q_i,t)\sim\pi_{\phi'}}\Bigg[\sum_{j=0}^{L_i-1}\min\big(\rho_{i,j}\times A_{\text{secure}}(q_i,t_j), \text{clip}\big(\rho_{i,j},1-\varepsilon,1+\varepsilon\big)\times A_{\text{secure}}(q_i,t_j)\big)\Bigg] \tag{10}$$

In practical training, since the objective is difficult to compute directly, we approximate its optimization by unfolding the sampled sequence and accumulating the objective at the token level. To align with secure code generation scenarios, we unfold each generated sequence within a batch and sum the objective functions across all tokens, as shown in Equation (8). Algorithm 1 illustrates the training details of SRCode's RL phase.

Algorithm 1 SRCode RL Training.

Input: Training samples q_i, policy model π_θ, teacher model, value network V_ϕ
Output: Updated policy model θ
Token Generation
for each query q_i **do**
  Generate code sequence $S_i = (t_0, \ldots, t_{L_i-1})$ using π_θ
  Reward Construction
  Teacher extracts secure token set T and vulnerability status $\text{Vul}(S_i)$
  for each token t_j **do**
    Assign positive reward if $t_j \in T$
    Assign negative penalty if $\text{Vul}(S_i) = \text{True}$
    Compute final reward $r_{i,j}$ using Eq. (3)
  end for
  Advantage Computation (GAE)
  for each token t_j **do**
    Compute TD residual $\delta_{i,j}$ using Eq. (5)
    Compute advantage $A_{\text{secure}}(q_i,t_j)$ using Eq.(6)
  end for
  PPO Objective
  for each token t_j **do**
    Compute importance ratio $\rho_{i,j}$ using Eq. (8)
    Compute clipped PPO term
    Compute token-level objective $J_{i,j}$ using Eq. (7)
  end for
end for
Policy Update
Update policy parameters θ using Eq.(9)

B Evaluation Details

Our evaluation consists of two main components, designed to measure the model's performance in code generation safety and code generation general capability, respectively. For code safety evaluation, three representative security benchmarks are employed: CWEval (Peng et al., 2025), CodeLMSec (Hajipour et al., 2024a), and CyberSecEval (Wan et al., 2024). For general code generation capability assessment, we utilize two extended and enhanced classic code generation benchmarks: HumanEval Pro and MBPP Pro (Yu et al., 2024).

CWEval. We employ the FS@1 (func-sec@1) metric to simultaneously evaluate generated code across functional correctness and security dimensions. A code instance is counted as successful only when it passes both functional tests and satisfies security constraints. The specific formula of FS@1 is as follows:

$$\text{FS@1} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\left(f(\hat{c}_i)\wedge s(\hat{c}_i)\right)$$

where N denotes the total number of evaluation samples, and $\hat{c}_i$ represents the first code sample generated by the model for the ith input. The function $f(\cdot)$ is the functional correctness evaluation function, returning "True" when the generated code passes the corresponding functional test, and "False" otherwise. Function $s(\cdot)$ denotes the safety verification function, returning "True" when the generated code satisfies security constraints and contains no known vulnerabilities or insecure implementations, and "False" otherwise. The indicator function $\mathbb{I}(\cdot)$ takes the value "1" when its condition holds, and "0" otherwise. Additionally, building upon the C/C++ task, we supplement our evaluation using its provided Python subset to observe the model's generalization performance in cross-language security modes.

CodeLMSec. This evaluation benchmark uses the CodeQL analyzer to inspect model-generated code. Analysis of its built-in query file collection reveals limited coverage of CWE vulnerabilities. To address this, we supplement the analysis with an additional query set provided by CodeQL. This set contains 181 security query files tailored for C/C++ scenarios, thereby enhancing vulnerability detection coverage and result reliability. The Sec.Rate metric is employed to quantitatively evaluate the security of model-generated code. Specifically, we perform multiple repeated samples of tasks from the evaluation set and use CodeQL to detect vulnerabilities in each sample. The Sec.Rate metric is defined as the proportion of valid generated samples that remain undetected for vulnerabilities, cal-

culated as follows:

$$\text{Sec.Rate} = \left(1 - \frac{\text{Insecure Generations}}{\text{Valid Generations}}\right) \times 100\%$$

CyberSecEval. As CyberSecEval is a collection of security evaluation benchmarks, we focus on the tasks within this collection that emphasize secure code generation. These tasks are referred to as "instruct tests" in the original benchmark. We adopt the official Pass.Rate metric for security evaluation, which measures the proportion of samples passing security evaluation within a category. To be more specific, it is defined as the percentage of evaluation items in which no security vulnerabilities are detected among all evaluated items. Similar to the Sec.Rate metric, Pass.Rate also measures the proportion of generated results passing security checks, but it aggregates statistics at the task level rather than the sample level.

HumanEval Pro and MBPP Pro. HumanEval Pro and MBPP Pro are enhanced versions built upon the classic code generation benchmarks HumanEval (Chen, 2021) and MBPP (Austin et al., 2021), designed to evaluate models' comprehensive capabilities in self-calling code generation scenarios. Unlike the original benchmarks that focus solely on single-function generation, these two Pro versions require models to correctly reuse their own code after generating a base solution to complete more complex derivative tasks. This simultaneously evaluates models' code generation capabilities, progressive reasoning abilities, and consistency in code reuse and invocation. Regarding evaluation metrics, we adopt the unbiased estimation method for pass@k proposed in prior work to assess model performance (Chen, 2021). Specifically, for each problem, we first generate n code samples and count the number of samples that pass the test as c. Subsequently, pass@k is defined as the probability that at least one correct sample is included when randomly selecting k samples from these n samples. Its mathematical form is as follows:

$$\text{pass@}k = \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

where $(n-c)$ denotes the number of code samples that failed testing. The term $\frac{\binom{n-c}{k}}{\binom{n}{k}}$ represents the probability that all k randomly selected samples are erroneous code. This unbiased estimation method effectively reduces evaluation variance under finite sampling conditions and is widely used in code generation benchmark assessments.

C Setup Details

All training and evaluation are conducted on two NVIDIA H100 80G GPUs, amounting to approximately 60 hours of total GPU time. Training and optimization for all models are performed using the LLaMAFactory framework (Zheng et al., 2024), employing a unified training workflow and hyperparameter settings. The low-rank adaptation (LoRA) (Hu et al., 2022) is introduced during training to achieve efficient parameter fine-tuning. LoRA is configured with a rank of 8 and scaling factor of 16, without dropout, and applied to all adaptable layers in the model. The LoRA settings remain consistent across both supervised fine-tuning and reinforcement learning phases. For evaluation benchmarks, all models are locally deployed via the vLLM framework (Kwon et al., 2023) on the same GPU environment. For the code safety evaluation benchmark, we uniformly set the sampling temperature to 0.8 for models with parameter sizes of at least 6.7B, and to 0.6 for all other models. For the code generation general capability benchmark, we uniformly set the temperature to 0. For all pass@1 metrics, we estimate results using 64 independent samples to ensure experimental reliability.

During supervised fine-tuning, models are trained on instruction-response format data with a maximum input sequence length of 2048. Training employs the AdamW optimizer with a learning rate of 5×10^{-5} and a cosine learning rate scheduling strategy. Each GPU employed a batch size of 2 during training, achieving an equivalent batch size of 16 via 8-step gradient accumulation. The gradient clipping threshold is set to 1.0. All training is conducted at BF16 precision to balance computational efficiency and numerical stability.

During the reinforcement learning phase, LoRA is also employed for efficient parameter training. A reward model fine-tuned using LoRA is loaded for return computation. Furthermore, Qwen3-Coder-480B-A35B-Instruct is selected as the teacher model, with both the positive reward α and the upper bound of negative rewards for token-level rewards set to 0.2. The learning rate for reinforcement learning remains consistent with the supervised fine-tuning phase, and all hyperparameters are identical. During policy generation, we employ a top-p sampling strategy set to 0.9 without top-k

truncation to enhance diversity in generated outputs. At the implementation level, all experiments utilize Flash Attention's auto-optimization mode and run in a distributed training environment.

D Extended Ablation Study for SRCode

In this section, we first conduct an in-depth analysis of the results presented in Table 3. Notably, we observe an intriguing and insightful phenomenon on the CodeLMSec benchmark: models trained solely with RL perform marginally better than the SFT + RL variant. Reasonably inferring the cause, we find that CodeLMSec prompts are reverse-engineered by the model, exhibiting strong adversarial properties and weak semantic constraints. RL models without TLR directly update parameters based on instance-level security rewards during training, leading them to adopt more conservative output strategies when encountering such inputs. This training-evaluation consistency yields a slight score improvement but does not imply that such models possess higher security or generalization capabilities in real-world code generation tasks. A similar pattern emerges in Table 1 when comparing our proposed SRCode with Purpcode.

Second, to further validate the effectiveness of the hierarchical data design, we conduct additional ablation experiments on the security evaluation benchmark CodeLMSec and the general code generation benchmark HumanEval Pro, with the results as shown in Table 5.

Method	CodeLMSec	HumanEval Pro
	Sec.Rate	pass@1
SRCode	**88.7**	**69.5**
w/o repair task	83.2	68.1
w/o progressive order	85.5	66.6
w/o explicit order	86.2	69.1

Table 5: **Ablation study for the hierarchical data design.** We evaluate the effectiveness of curriculum data design on a security benchmark and a general code generation benchmark. ***w/o repair task*** denotes removing the intermediate-difficulty vulnerability repair task; ***w/o progressive order*** indicates breaking the progressive difficulty ordering of tasks; ***w/o explicit order*** refers to randomly mixing tasks of different difficulty levels without an explicit curriculum structure.

The overall results show that the complete hierarchical dataset achieves optimal performance on both benchmarks, indicating that organizing data in ascending order of difficulty can simultaneously enhance both the model's code security and its general code generation capabilities. When medium-difficulty vulnerability remediation tasks are removed, the model's performance on the security evaluation benchmark declines significantly. This demonstrates that vulnerability remediation tasks serve as a crucial intermediate stage, bridging the transition from vulnerability identification to secure code generation.

Regarding the configurations that disrupt the difficulty progression or randomly mixed tasks of varying difficulty, the model exhibits varying degrees of performance drop. This observation demonstrates that the task sequence itself could significantly impact the model's ability to progressively learn complex code generation. Note that, in the randomly mixed task setting, the model's performance on the general capability evaluation approaches that of the complete dataset configuration. However, it still shows a gap in security metrics, further indicating that the explicit hierarchical data structure is more effective in consistently enhancing secure code generation capabilities.

E Vulnerability Distribution Analysis

Figure 3 shows that compared to various baseline models, SRCode achieves a significant reduction in both high-risk and medium-risk vulnerabilities, demonstrating greater robustness and reliability in mitigating the most security-threatening severe vulnerabilities. Simultaneously, compared to existing security enhancement methods, SRCode exhibits an overwhelming advantage in reducing the overall number of vulnerabilities. Regardless of model size or structure, SRCode consistently achieves the lowest vulnerability count among all methods.

These observations further support RQ2, showing that SRCode provides reliable safety improvements across models with different scales and architectures. The consistent reduction in vulnerability counts suggests that these gains are stable and not limited to specific model settings.

F Sampling Efficiency Analysis

The experimental results on sampling efficiency gaps have been illustrated in Figure 2 (in the main paper). Note that, lower sampling counts better reflect real-world user experience, as practical usage typically involves only a small number of candidate generations. To enable a fair and precise comparison of RL-induced improvements, all base models

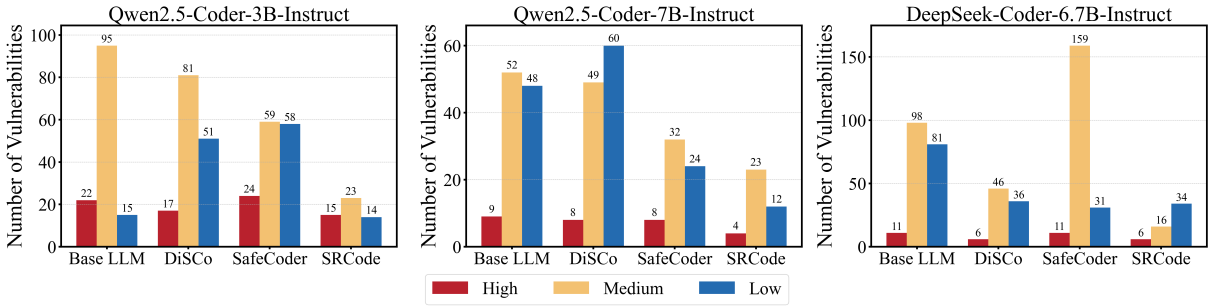

Figure 3: **Vulnerability distribution on CodeLMSec across different methods.** The number of High, Medium, and Low severity vulnerabilities are reported, representing high-risk, medium-risk, and low-risk security issues, respectively. Each subplot is for a specific base model. Lower vulnerability counts indicate safer model behavior.

are uniformly set to their SFT-only versions.

We observe from Figure 2 that SRCode achieves the optimal performance on this metric, with a value of only 0.234, significantly outperforming traditional PPO and Purpcode. Furthermore, under the $k = 1$ and $k = 2$ settings that more closely resemble real-world user scenarios, the sampling efficiency of SRCode is improved by 5.77% and 3.55% respectively compared to the baseline model, while the other two methods exhibit varying degrees of decline. This phenomenon indicates that under the low-sample constraints, the fine-grained token-level rewards mechanism introduced by SRCode more effectively guides the model toward strategy optimization. This enables it to generate higher-quality, more secure code without requiring extensive candidate sampling.

G Case Study and Discussions

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.

H Prompts

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 6: Overview of prompt implementations.

Table 6 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>
- 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: Prompt for teacher model evaluation for TLR

```
System: You are a code security auditing expert
    specializing in C/C++ vulnerability
    detection and secure hardening.
Original prompt received by the policy model:
    {prompt_text}
Code generated by the policy model: {code_text}
Please complete the task according to the
    following steps:
1. Required header file check:
- Determine whether the code is missing any
    header files required to complete the task.
```

```
- Do NOT provide the repaired code.
2. CWE vulnerability detection:
- Determine whether the code contains any CWE
    vulnerabilities, especially those described
    in the original prompt.
3. Secure implementation identification:
- Mark tokens in the code that correspond to
    secure implementations
  (e.g., use of safe functions, correct header
      files, defensive checks).
- Output must be a list in the following format:
  "correct_tokens": ["token1", "token2", ...]
- Do NOT output any natural language
    explanations. Only output tokens.
4. Minimal repair assessment and negative
    reward calculation:
- Assign a negative reward based on
    vulnerability severity and missing header
    files.
- If the vulnerability described in the prompt
    exists, it is considered the most severe.
- The reward range is [min, 0], where more
    severe issues receive larger negative
    values.
- If the code contains no vulnerabilities, then
    negative_reward = 0.
```