

Refinement-Integrated Reasoning Elicits Better Generalization of LLM-Based Code Vulnerability Detection

Xianda Du¹, Pei Ke^{1,2,*}, Enping Hu¹, Guannan Li¹, Jinyu Guo³, Ke Qin^{1,2}

¹Laboratory of Intelligent Collaborative Computing,

University of Electronic Science and Technology of China, Chengdu, China

²Ubiquitous Intelligence and Trusted Services Key Laboratory of Sichuan Province

³School of Information and Software Engineering,

University of Electronic Science and Technology of China, Chengdu, China

202321080231@std.uestc.edu.cn, kepei@uestc.edu.cn, qinke@uestc.edu.cn

Abstract

Existing code vulnerability detection (CVD) methods face severe challenges in terms of generalization. Most of these methods directly map the source code to a binary vulnerability label or a vulnerability explanation, struggling to capture the deep features within the complex relationship between them. Thus, these works fail to generalize to out-of-distribution data well. To address this issue, we propose a simple yet effective training framework called refinement-integrated reasoning (RIRE), leveraging refinements that indicate the patched code in the CVD task. Firstly, RIRE introduces the refinement as a guiding signal to help LLMs synthesize reasoning chains as explanations, which align with the vulnerability label, and uses these data for supervised fine-tuning. Secondly, refinement is utilized through a hierarchical preference optimization framework to help determine which reasoning-enhanced detection result is of higher quality, further mining essential features for fine-grained distinguishability and detection capability. Experimental results across five benchmark datasets demonstrate that our proposed RIRE framework achieves superior performance over strong baselines, especially being more generalizable to unseen datasets during training.

1 Introduction

Code vulnerability detection (CVD) aims to determine whether potential vulnerabilities exist in the source code (Du et al., 2024). As a long-standing task for securing software systems, it has attracted much attention recently due to the increasing number of vulnerabilities in the software especially developed with the assistance of large language models (LLMs) (Jiang et al., 2024b; Weyssow et al., 2025). Therefore, how to effectively detect code vulnerabilities has become an essential research

problem towards secure and reliable software systems in the era of LLMs.

With the rapid development of pre-trained language models (PLMs), code-specific PLMs (such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and UniXcoder (Guo et al., 2022)) and general large language models (LLMs) have demonstrated remarkable capabilities on the CVD task (Jiang et al., 2024a; Fang et al., 2024) via fine-tuning. However, a critical limitation for these models is the generalization capability especially when dealing with out-of-distribution data (Du et al., 2023, 2024). Since the labeled datasets in the CVD task are commonly of restricted scale, the models are required to learn in-depth features from limited training data, which can generalize to unseen data with different programming style and application contexts (Du et al., 2024). We argue that most of the existing approaches which map the source code to a binary classification label (i.e., secure / vulnerable) (Jiang et al., 2024b) only learn superficial relationship instead of deep vulnerability features, causing potential misguidance from irrelevant factors. Recent works augment vulnerability labels with reasoning chains with LLMs as explanations, the CVD model after training is still struggling to capture transferable features within the complex relationship between source code and explanations (Weyssow et al., 2025).

To address this issue, we propose a simple yet effective training framework called **Refinement-Integrated REasoning** (RIRE). Refinement indicates the patched code in the CVD task, which can offer a referenced solution to fixing code vulnerabilities and thus help to mine deep features that can be transferable to out-of-distribution data. Our main idea is to utilize refinement to not only synthesize high-quality reasoning chains as detection results, but also enhance the fine-grained distinguishability by contrasting these results with a refinement-oriented hierarchical preference opti-

*Corresponding author

mization framework. Specifically, RIRE incorporates refinement to guide the synthesis of reasoning chains as interpretable detection results with LLMs, which are then used for supervised fine-tuning. Then, refinement is then utilized during preference optimization to distinguish the quality of detection results in multiple levels including labels and explanations, helping the CVD model to capture in-depth vulnerability features that can be more generalizable to out-of-distribution data.

Our contributions are as follows¹:

- To the best of our knowledge, we are the first attempt to incorporate patched code to learn transferable features in the CVD task and enhance the generalization ability to out-of-distribution data.
- We propose a training framework called RIRE, which fully leverages refinement to distinguish the quality of detection results in a hierarchical manner for preference optimization.
- We conduct extensive experiments on five datasets, showing that RIRE can achieve remarkable performance especially on the unseen datasets, demonstrating better generalization capabilities.

2 Related Work

2.1 Code Vulnerability Detection

Code vulnerability detection is of great importance for the security of software systems (Hanif et al., 2021; Du et al., 2024). Early works utilize different neural network architectures to capture the features of source code, such as graph neural networks (Zhou et al., 2019; Cheng et al., 2021) and recurrent neural networks (Li et al., 2018, 2022). As pre-trained language models develop, researchers propose various code PLMs including CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and UniXcoder (Guo et al., 2022) that achieve remarkable performance on the CVD task. Recent works start to explore the use of LLMs for the CVD task. VulLLM (Du et al., 2024) adopts a multi-task learning approach that includes vulnerability localization, detection, and interpretation, to improve the generalization ability. LLM4CVD (Jiang et al., 2024b) evaluates four open-source

LLMs on five CVD benchmark datasets and investigates the effect of sample lengths on model performance.

Since most of these works solely focus on binary classification (i.e., whether a vulnerability exists), researchers tend to investigate the reasons / explanations for the prediction result of LLMs. They leverage prompt-based techniques to directly annotate vulnerability explanations and utilize them to train LLMs via supervised fine-tuning (Mao et al., 2024) and reinforcement learning (Weyssow et al., 2025).

Compared with existing works, our work is the first attempt to incorporate patched code as refinement to enhance the generalization ability of LLM-based code vulnerability detection models. We fully utilize refinement in the whole training process to capture transferable vulnerability features via supervised fine-tuning and hierarchical preference optimization.

2.2 Reasoning in Large Language Models

Recently, reasoning attracts much attention with the rapid development of large reasoning models such as OpenAI o1 (Jaech et al., 2024) and DeepSeek-R1 (Guo et al., 2025). There are mainly two lines of work on reasoning in large language models. The first line falls into the inference phase, which designs input prompts (Wei et al., 2022; Kojima et al., 2022) and decoding methods (Yao et al., 2023; Besta et al., 2024) to stimulate LLMs to generate high-quality reasoning results. Another line of work focuses on training methods that involve reasoning data construction (Magister et al., 2023; Cheng et al., 2024; Ye et al., 2025) and training objective design (Shao et al., 2024). These works are commonly validated in various reasoning tasks, such as mathematical, commonsense, and logical reasoning.

In comparison, our work focuses on the CVD task and incorporates reasoning into LLM-based CVD methods to acquire reliable explanations. Furthermore, we introduce refinement to learn deep mapping relationships between source code and detection explanations, which elicit generalization to out-of-distribution data.

3 Method

3.1 Task Definition and Method Overview

Given the source code x , our goal is to acquire the vulnerability detection result $y = (l, e)$, which in-

¹Our codes are available at <https://github.com/aNGeL4me/RIRE>.

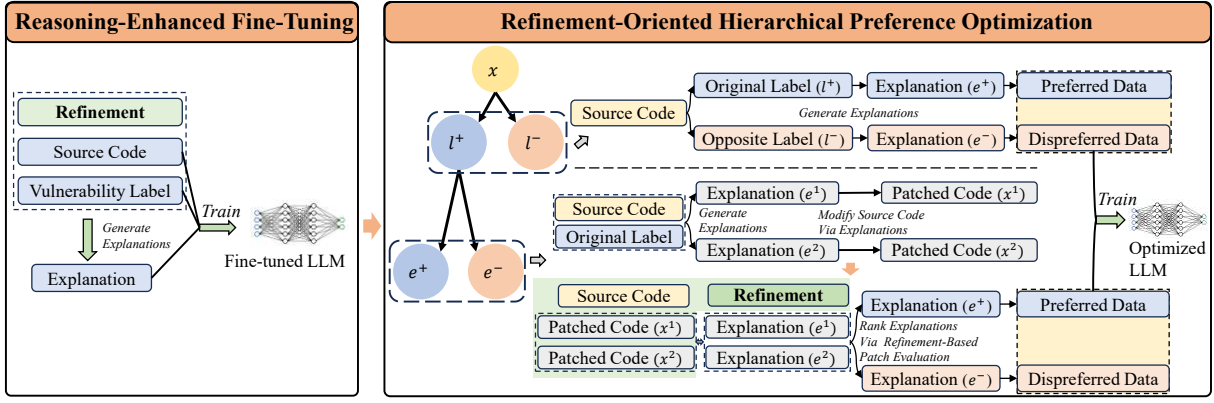


Figure 1: Overview of RIRE. **Left:** Our method synthesizes reasoning chains as explanations based on the source code, vulnerability label, and refinement. Then, our model is fine-tuned on the dataset augmented by explanations. **Right:** Our method subsequently conducts hierarchical preference optimization, where the preference dataset with contrastive labels or explanations is constructed with the assistance of refinement.

cludes a binary classification label l and a detailed explanation e to support the label. The label set is denoted as $l \in \{0, 1\}$ where 0 / 1 indicates secure / vulnerable code, respectively.

As shown in Figure 1, our main idea is to fully integrate refinement into the whole training process of LLM-based CVD models to capture the essential features with limited training data for better generalization to out-of-distribution datasets. Our proposed framework consists of three key stages:

- **Refinement Collection:** For each code sample x , we collect its patched version x^r as refinement via web-based development platform (such as GitHub) or LLM-based data synthesis, which serves as guidance for acquiring reasoning chains as explanations.
- **Reasoning-Enhanced Fine-Tuning:** We leverage a powerful LLM, i.e., DeepSeek-R1 (Guo et al., 2025), to generate reasoning chains as explanations based on the source code x , vulnerability label l , and collected refinement x^r . Then, we fine-tune our model on these reasoning-enhanced dataset.
- **Refinement-Oriented Hierarchical Preference Optimization:** We further conduct hierarchical preference optimization where the refinement is adopted to enhance two-level distinguishability between reasoning-enhanced detection results, leading to a high-quality preference dataset. Then, hierarchical preference optimization is utilized to further improve the performance and generalization of our model.

3.2 Refinement Collection

To collect patched code as refinement for training, we combine web crawling with LLM-based augmentation. Since most CVD datasets (e.g., DiverseVul and Devign) are sourced from GitHub, we trace the original repositories and retrieve commit histories via the GitHub API. For each sample in the CVD dataset $D = \{(x_i, l_i)\}_{i=1}^{|D|}$, we extract the corresponding commit and obtain the code changes and commit message. We then augment the original dataset with the patched code as refinement x^r and the associated commit message x^{rm} for reasoning data construction and model training. For samples whose repositories are inaccessible, we resort to LLMs to directly generate patched code based on the source code and vulnerability label.

To assess patch quality, we conducted a human evaluation on 100 randomly sampled model-generated patches. The evaluation yields a correctness rate of 85%, indicating high-quality patch generation and the effectiveness of our method. Detailed analysis is provided in Appendix A.

3.3 Reasoning-Enhanced Fine-Tuning

To equip our model with the ability to generate explanations which can provide detailed information to support the vulnerability label, we first design the form of explanations and make it include three core parts, i.e., vulnerable lines, causes, and summaries. Specifically, the lines indicate the position of potential vulnerabilities in the source code, while the causes denote the description and analysis of vulnerabilities in the corresponding lines. The summaries provide an overview of the above

information to directly support the vulnerability label.

To construct high-quality explanation data, we employ a powerful LLM (i.e., DeepSeek-R1) to generate a three-step reasoning chain as the explanation e_i based on the source code x_i , vulnerability label l_i , refinement x_i^r , and its message x_i^{rm} :

$$e_i = \text{LLM}(x_i, l_i, x_i^r, x_i^{rm}) \quad (1)$$

The prompt design for explanation generation is shown in the Appendix D. The refinement serves as a reference to help the LLM localize and analyze vulnerabilities based on the source code. Note that for the samples that are labeled as secure, the modifications from source code to refinement are typically functional and unrelated to security. Therefore, we guide the LLM to analyze the secure mechanisms in these samples via prompt design.

After acquiring the reasoning-enhanced dataset $D_r = \{(x_i, l_i, e_i)\}_{i=1}^{|D_r|}$, we perform supervised fine-tuning on the LLM P_θ to equip it with the preliminary ability to generate interpretable detection results including both vulnerability labels and explanations:

$$\mathcal{L}_{FT} = - \sum_{i=1}^{|D_r|} \log P_\theta(l_i, e_i | x_i) \quad (2)$$

3.4 Refinement-Oriented Hierarchical Preference Optimization

To further enhance the model’s generalization ability, we employ hierarchical preference optimization and incorporate refinement to help construct the two-level preference dataset. Specifically, for each sample (x_i, l_i) , $1 \leq i \leq |D_r|$. We construct preference pairs in a hierarchical manner:

- **Paired Data with Opposite Labels:** To make our model generate more accurate detection results, we construct preference pairs to enhance its distinguishability between accurate and wrong results. Specifically, we adopt DeepSeek-R1 to generate explanations with opposite labels:

$$e_i^+ = \text{LLM}(x_i, l_i) \quad (3)$$

$$e_i^- = \text{LLM}(x_i, 1 - l_i) \quad (4)$$

Thus, the original label with its explanation can be regarded as preferred data $y_i^+ = (l_i, e_i^+)$, while the opposite label $1 - l_i$ with the

corresponding explanation e_i^- is recognized as dispreferred data $y_i^- = (1 - l_i, e_i^-)$.

- **Paired Data with Consistent (Correct) Labels:** To further equip our model with the ability to generate more reliable explanations, we construct preference pairs to enhance its distinguishability between the explanations of different qualities that correspond to correct label. Similarly, we first utilize DeepSeek-R1 to generate two explanations conditioned on the correct label:

$$e_i^1, e_i^2 = \text{LLM}(x_i, l_i) \quad (5)$$

Existing works (Sun et al., 2024) have shown that the quality of detection results (especially explanations) may not be easy to distinguish directly via LLMs. Thus, inspired by Yu et al. (2025), we use DeepSeek-R1 to modify the source code x_i via these two explanations e_i^1, e_i^2 , respectively, to acquire the patched code x_i^1, x_i^2 :

$$x_i^1 = \text{LLM}(x_i, l_i, e_i^1) \quad (6)$$

$$x_i^2 = \text{LLM}(x_i, l_i, e_i^2) \quad (7)$$

Then, we incorporate the refinement x_i^r to judge which patched code is better, and adopt this result to determine the quality of e_i^1 and e_i^2 indirectly:

$$e_i^+, e_i^- = \text{LLM}(x_i, x_i^r, e_i^1, e_i^2) \quad (8)$$

where $\text{LLM}(x_i, x_i^r, e_i^1, e_i^2)$ outputs two explanations in the descending order of qualities. Therefore, the preferred and dispreferred data are denoted as $y_i^+ = (l_i, e_i^+)$, $y_i^- = (l_i, e_i^-)$, respectively.

After preference data construction, we utilize direct preference optimization (DPO) (Rafailov et al., 2023) on the collected dataset $D_p = \{(x_i, y_i^+, y_i^-)\}_{i=1}^{|D_p|}$ to further train P_θ :

$$\mathcal{L}_{PO} = - \sum_{i=1}^{|D_p|} \log \sigma(\beta \log \frac{P_\theta(y_i^+ | x_i)}{P_{ref}(y_i^+ | x_i)} - \beta \log \frac{P_\theta(y_i^- | x_i)}{P_{ref}(y_i^- | x_i)}) \quad (9)$$

where $P_{ref}(\cdot)$ denotes the reference model initialized with the model after reasoning-enhanced fine-tuning, and $\sigma(\cdot)$ indicates the sigmoid function. β

is a controlling factor in the DPO algorithm. Note that our method mainly utilizes refinement to improve the training process. During inference, our model does not need refinement as extra input and adopts the same settings as common CVD models.

4 Experiment

4.1 Dataset

Following Du et al. (2024), we select five widely-used vulnerability detection datasets to measure the performance of different models: DiverseVul (Chen et al., 2023), Devign (Zhou et al., 2019), BigVul (Fan et al., 2020), ReVeal (Chakraborty et al., 2021), Juliet (Boland and Black, 2012).

We adopt the same setting as Du et al. (2024), which combines the training / validation sets of DiverseVul and Devign for model training. Then, we assess the model performance on all the five datasets, respectively. The amount of samples for training / validation is 22,837 / 2,864, respectively. The test set of DiverseVul / Devign / BigVul / ReVeal / Juliet contains 1,532 / 1,873 / 1,170 / 2,028 / 3,152 examples, respectively. In this way, the performance on unseen datasets (i.e., ReVeal, BigVul, and Juliet) can reflect the generalization ability besides effectiveness.

4.2 Implementation Detail

We choose CodeLlama-7B and Qwen2.5-Coder-7B as the base model, which are both commonly-used LLMs for code-related tasks. The experiments are conducted on 4 NVIDIA A800 GPUs. Considering the limitation of computational resources, we perform low-rank adaption (LoRA) (Hu et al., 2022) during reasoning-enhanced fine-tuning. The rank of LoRA is set to 80. We use AdamW (Loshchilov and Hutter, 2019) optimizer with the weight decay of 0.01. The learning rate is $1e-4$. We set the context length to 1,024 and the batch size to 32. The number of training epochs is 10.

During refinement-oriented hierarchical preference optimization, we also adopt LoRA and set the rank to 80. The same optimizer is utilized with the learning rate of $2e-6$ and the warmup steps of 50. The batch size is 32, while the number of training epochs is 3. We set the hyper-parameter β to 0.1. Greedy decoding is employed in our experiment.

As for the data preparation of refinement-oriented hierarchical preference optimization, we sampled 7,940 instances with a 1:1 ratio of vulnerable to secure data from the overall training set.

Among these, 5,292 samples are used to generate paired data with both opposite labels and consistent labels, while the remaining 2,648 samples were used to generate paired data with only opposite labels. In total, 13,232 preference pairs are used for DPO training.

4.3 Baseline

We consider state-of-the-art non-LLM-based and LLM-based models as our baselines.

Non-LLM-based Models: We follow Du et al. (2024) to first select strong baselines that are not built based on LLMs, including Devign (Zhou et al., 2019), ReVeal (Chakraborty et al., 2021), CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), UniXcoder (Guo et al., 2022), ReGVD (Nguyen et al., 2022), and EPVD (Zhang et al., 2023). For these baselines, we directly reprint the experimental results from Du et al. (2024).

LLM-based Models: We consider two categories of LLM-based models as baselines. Firstly, we involve two state-of-the-art closed-source LLMs, i.e., GPT-4o and DeepSeek-R1 (Guo et al., 2025), which show promising results on general tasks. Secondly, we include several training methods that adopt the same dataset as RIRE:

- **SFT** (Jiang et al., 2024b): A standard supervised fine-tuning method that maps each code sample directly to a binary classification label.
- **SFT-CoT** (Ho et al., 2023): A variant of supervised fine-tuning that trains the model on chain-of-thought (CoT) reasoning chains directly generated based on DeepSeek-R1 without the use of external information.
- **VulLLM** (Du et al., 2024): A multi-task learning method with data augmentation. It introduces auxiliary tasks such as vulnerability localization and explanation to improve the performance of binary vulnerability detection.

4.4 Main Result

Following Du et al. (2024), we adopt the F1 score as our primary evaluation metric. The F1 score is widely used in CVD tasks (Du et al., 2024; Jiang et al., 2024b), for its ability to balance precision and recall. To further improve the robustness of the evaluation, we additionally report the Matthews Correlation Coefficient (MCC), which is particularly suitable for assessing binary classifiers on imbalanced datasets. As shown in Table

Method	Seen Dataset				Unseen Dataset						Average					
	DiverseVul		Devign		BigVul		ReVeal		Juliet		Seen		Unseen		All	
	F1	MCC	F1	MCC	F1	MCC	F1	MCC	F1	MCC	F1	MCC	F1	MCC	F1	MCC
<i>Non-LLM-Based Method</i>																
Devign	60.36	/	58.93	/	53.14	/	59.43	/	57.71	/	59.65	/	56.76	/	57.91	/
ReVeal	57.04	/	53.84	/	53.92	/	56.67	/	57.86	/	55.44	/	56.15	/	55.87	/
CodeBERT	65.45	/	<u>66.81</u>	/	55.19	/	46.48	/	28.73	/	66.13	/	43.47	/	52.53	/
GraphCodeBERT	66.00	/	66.62	/	54.70	/	55.61	/	31.22	/	66.31	/	47.18	/	54.83	/
UniXcoder	67.27	/	66.05	/	56.06	/	59.57	/	61.36	/	66.66	/	59.00	/	62.06	/
ReGVD	65.86	/	61.14	/	58.55	/	54.94	/	36.87	/	63.50	/	50.12	/	55.47	/
EPVD	66.85	/	66.76	/	52.89	/	48.95	/	25.86	/	<u>66.81</u>	/	42.57	/	52.26	/
<i>LLM-Based Method</i>																
<i>Untrained Model based on Closed-Source APIs</i>																
GPT-4o	52.30	1.18	53.69	6.74	55.81	6.37	52.26	5.03	74.21	41.69	53.00	3.96	60.76	17.70	57.65	12.20
DeepSeek-R1	52.05	5.22	49.08	4.85	52.97	3.76	48.54	0.62	72.56	40.43	50.57	5.04	58.02	14.94	55.04	10.98
<i>Trained Model based on CodeLlama</i>																
SFT	58.77	1.98	58.05	-4.95	46.56	-0.86	59.71	0.67	49.80	4.72	58.41	-1.49	52.02	1.51	54.58	0.31
SFT-CoT	59.32	26.71	58.53	12.06	52.17	20.13	55.04	8.98	62.80	3.08	58.93	19.39	56.67	10.73	57.57	14.19
VulLLM	67.46	35.38	62.78	27.13	62.92	22.33	65.70	23.54	58.25	7.61	65.12	31.26	62.29	17.83	63.42	23.20
RIRE (Ours)	68.39	28.12	62.93	15.43	66.92	25.63	65.07	15.70	79.77	62.19	65.66	21.78	70.59	34.51	68.62	29.41

Table 1: F1 and MCC scores on the five datasets. The best results of trained models based on CodeLlama are highlighted in bold, while the best results across all the models are underlined.

ID	FT		PO		Seen Dataset		Unseen Dataset			Average
	CoT	Ref.	DPO	Ref.	DiverseVul	Devign	BigVul	ReVeal	Juliet	
①	✗	✗	✗	✗	58.77	58.05	46.56	59.71	49.80	54.58
②	✓	✗	✗	✗	59.32	58.53	52.17	55.04	62.80	57.57
③	✓	✓	✗	✗	67.64	62.02	65.93	64.19	77.24	67.40
④	✓	✓	✓	✗	68.11	62.34	66.36	64.86	79.32	68.20
⑤	✓	✓	✓	✓	68.39	62.93	66.92	65.07	79.77	68.62

Table 2: F1 scores of different ablation models built on CodeLlama. FT / PO indicates the training process of supervised fine-tuning / preference optimization, respectively. CoT means whether the training data contains reasoning chains in addition to vulnerability labels. DPO represents whether DPO is conducted on the model. Ref. means whether refinement is incorporated into supervised fine-tuning / preference optimization.

1, RIRE achieves superior performance on most of the datasets over the trained baselines, which shows the effectiveness and generalization ability across different datasets.

We further examine the F1–MCC discrepancy in our results. RIRE exhibits relatively higher false positive rates and recall-oriented behavior compared to some baselines. This likely stems from the explanation-enhanced supervision in our reasoning data: for vulnerable samples, the refinement provides concrete evidence that enriches explanations with explicit vulnerability cues, whereas safe samples lack such strong signals and yield more general explanations. This asymmetry encourages recall-oriented predictions, increasing false positives and lowering MCC despite competitive F1 scores. Importantly, on unseen benchmarks like

BigVul and Juliet, RIRE still achieves higher MCC than VulLLM, confirming its generalization advantage beyond F1 gains.

Notably, RIRE outperforms VulLLM in terms of the average performance on CodeLlama, which is a strong baseline that only outputs the vulnerability label. This indicates that RIRE can successfully improve the detection performance while providing interpretable reasoning chains as explanations. This also highlights the potential of incorporating reasoning into code understanding tasks.

Additionally, we can observe that RIRE can beat state-of-the-art closed-source LLMs like GPT-4o and DeepSeek-R1 on nearly all the datasets with a much smaller scale. This demonstrates that closed-source LLMs still struggle with the CVD task possibly due to the wide range of potential vulnerabil-

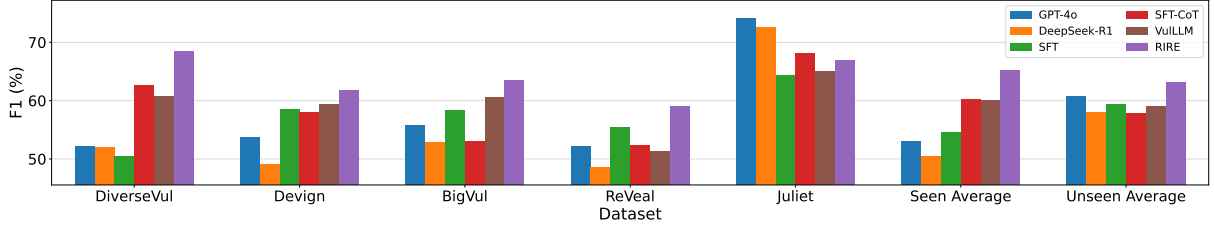


Figure 2: F1 scores across five datasets and the Seen/Unseen averages using Qwen2.5-Coder 7B as the base model.

ities. For comparison, our model equipped with supervised fine-tuning and hierarchical preference optimization on refinement-integrated reasoning data can achieve better performance even with a much smaller scale.

We additionally analyze the scalability of RIRE across different model sizes. Experiments on Qwen2.5-Coder with 1.5B, 3B, and 7B parameters indicate that RIRE consistently achieves superior performance over the baselines and benefits from larger model scales, highlighting its strong adaptability and scalability. Detailed results are provided in Appendix C.

4.5 Ablation Study

To investigate the impact of each component in our proposed RIRE method, we conduct a detailed ablation study. As shown in Table 2, we remove each part during fine-tuning and hierarchical preference optimization respectively and construct five variants to measure the individual contribution. The findings from the ablation study are as follows:

Refinement works well for both fine-tuning and hierarchical preference optimization. Comparing ② and ③, we can observe that refinement brings a relatively large performance improvement to supervised fine-tuning. This indicates that refinement-integrated synthesis can remarkably enhance the quality of reasoning chains, which contributes to the detection performance. From ④ and ⑤, we can find that refinement also works well for hierarchical preference optimization, though the improvement is smaller than that in the process of fine-tuning.

Reasoning improves the model performance on both seen and unseen data. The comparison between ① and ② demonstrates that enriching the output with reasoning chains not only provides explanations that can be understood by researchers, but also benefits the detection performance. This lays the foundation for our proposed refinement-

integrated reasoning method to further improve the performance and generalization.

4.6 Analysis on Qwen2.5-Coder

Using the Qwen2.5-Coder 7B model, we evaluate our method against baseline approaches on five benchmark datasets, consisting of seen datasets (DiverseVul and Devign) and unseen datasets (BigVul, ReVeal, and Juliet). We further report the average performance on the seen and unseen datasets. As illustrated in Figure 2, our method consistently outperforms the baselines across all datasets except Juliet.

4.7 Analysis on Human Evaluation

To verify whether our method can provide reliable explanations to interpret the reasons for detection results, we conduct human evaluation on the generated explanations. We follow the existing works (Prasad et al., 2023; Lyu et al., 2023) to select *correctness* and *faithfulness* as our evaluation criteria. Correctness indicates whether the explanations are factually and logically correct, while faithfulness means whether the explanations are faithful to the predicted vulnerability label.

We randomly select 100 samples from all the five datasets (i.e., 20 from each dataset) and collect the generated results from RIRE (based on CodeLlama) and three baselines that can generate explanations (i.e., GPT-4o, DeepSeek-R1, and SFT-CoT). We adopt the pairwise comparison between our model and the above baselines. For each pair of generated results (one from RIRE and the other from the corresponding baseline, given the same input source code), three experts in the field of security are hired to label which is better (i.e., win, lose, or tie) in terms of the metrics above. The two metrics are evaluated independently.

The results in Table 3 show that RIRE can significantly surpass the three baselines on both correctness and faithfulness, demonstrating that our method can provide high-quality explanations that

Method	Correctness				Faithfulness			
	Win (%)	Lose (%)	Tie (%)	κ (%)	Win (%)	Lose (%)	Tie (%)	κ (%)
RIRE vs. GPT-4o	50.8**	12.5	36.7	97.18	50.8**	15.8	33.4	34.67
RIRE vs. DeepSeek-R1	51.7**	12.5	35.8	97.08	47.5*	20	32.5	49.95
RIRE vs. SFT-CoT	26.7*	10	63.3	74.25	49.2	23.3	27.5	38.61

(a) Seen datasets

Method	Correctness				Faithfulness			
	Win (%)	Lose (%)	Tie (%)	κ (%)	Win (%)	Lose (%)	Tie (%)	κ (%)
RIRE vs. GPT-4o	34.4**	19.4	46.2	54.30	43.3*	22.8	33.9	39.75
RIRE vs. DeepSeek-R1	33.3	22.8	43.9	55.17	50.6*	22.2	27.2	49.72
RIRE vs. SFT-CoT	33.9	21.7	44.4	54.04	41.1	27.8	31.1	57.67

(b) Unseen datasets

Table 3: Human evaluation results of correctness and faithfulness on seen and unseen datasets. The scores indicate the percentages of *win*, *lose*, and *tie* when RIRE is compared with baselines. κ denotes Fleiss’ Kappa that reflects the agreement of different annotators (all values indicate at least moderate agreement). The scores marked with * mean $p < 0.05$ while ** means $p < 0.01$ in the sign test.

Source Code <pre>static int asf_read_ext_content(AVFormatContext *s, const GUIDParseTable *g) { ASFContext *asf = s->priv_data; AVIOContext *pb = s->pb; uint64_t size = avio_rl64(pb); uint16_t nb_desc = avio_rl16(pb); int i, ret; for (i = 0; i < nb_desc; i++) { uint16_t name_len, type, val_len; uint8_t *name = NULL; name_len = avio_rl16(pb); if (!name_len) return AVERROR_INVALIDDATA; name = av_malloc(name_len); if (!name) return AVERROR(ENOMEM); avio_get_str16le(pb, name_len, name, name_len); type = avio_rl16(pb); val_len = avio_rl16(pb); if ((ret = process_metadata(s, name, name_len, val_len, type, &s- >metadata)) < 0) return ret; align_position(pb, asf->offset, size); return 0; } }</pre> Vulnerability Label Vulnerable	GPT-4o The code snippet shows memory allocation and string operation, with a mitigating check verifying that the string length is non-zero before processing. Furthermore, dynamic memory allocation ('av_malloc') is checked for success, preventing null pointer dereference issues..... [result_start] Secure [result_end] SFT-CoT Result: [Vulnerable] Vulnerable line: align_position(pb, asf->offset, size); Cause: None Summary: The function reads a 64-bit 'size' value from the input stream using 'avio_rl64(pb)' but does not validate it. If 'size' is excessively large..... RIRE Result: [Vulnerable] Vulnerable line: if ((ret = process_metadata(s, name, name_len, val_len, type, &s->metadata)) < 0) Cause: The original code does not free the 'name' buffer allocated via 'av_malloc' if 'process_metadata' returns an error. This leaves the dynamically allocated memory unreleased, causing a memory leak. The 'name' pointer is only freed after 'process_metadata' is called, but if 'process_metadata' fails, the error path skips freeing 'name', leaving the allocated memory permanently unreleased. Summary: Memory leak due to failure to free 'name' buffer before returning on error in 'process_metadata' call.	Source Code <pre>SPL_METHOD (SplFileObject , key) { spl_filesystem_object * intern = (spl_filesystem_object *) zend_object_store_get_object (getThis () TSRMLS_CC); if (zend_parse_parameters_none () == FAILURE) { return ; } RETURN_LONG (intern -> u.file.current_line_num); }</pre> Vulnerability Label Vulnerable DeepSeek-R1 The code includes a check (for the absence of parameters) that ensures correct usage. However, the code does not perform any inherently risky operations (like buffer copies or pointer storage) that would require security mitigation. The parameter check is for correctness, not security. Since there is no vulnerability pattern present and the code is straightforward, it is secure..... [result_start] Secure [result_end]	GPT-4o The code snippet involves accessing a 'spl_filesystem_object' and returning a value from a member field without any apparent pattern related to a buffer overflow or use-after-free design flaw..... [result_start] Secure [result_end] SFT-CoT Result: [Secure] Vulnerable line: None Cause: None Summary: The function securely retrieves the current line number without vulnerabilities. It uses 'zend_parse_parameters_none' to enforce no input parameters are accepted, preventing malicious input..... RIRE Result: [Vulnerable] Vulnerable line: RETURN_LONG (intern -> u.file . current_line_num); Cause: The code directly accesses 'intern->u.file' without validating if the 'file' member is valid. If the object was created via a different constructor (e.g., 'SplFileInfo'), 'file' might be uninitialized or invalid, leading to undefined behavior or memory corruption when dereferencing it. Summary: The code assumes 'intern->u.file' is always valid, but objects created via different constructors (e.g., 'SplFileInfo') do not initialize 'file', causing unsafe memory access.
--	---	---	---

Figure 3: Generated results on Devign. The problematic part is highlighted in blue, while the correct core part is in red.

can be utilized to interpret how the models predict the final label. We observe a larger improvement margin on the seen datasets than on the unseen ones, as vulnerability detection on unseen datasets is inherently more challenging. Nevertheless, the model demonstrates strong generalization ability, showing effective transfer to unseen datasets even when trained solely on seen data.

4.8 Case Study

To intuitively show the generation quality of our model, we present two qualitative examples in Figure 3 and 4. The case shown in Figure 3 belongs to the Devign dataset, both GPT-4o and DeepSeek-R1 predict incorrect vulnerability label. SFT-CoT correctly predicts the vulnerability label; however,

Figure 4: Generated results on Reveal. The problematic part is highlighted in blue, while the correct core part is in red.

it fails to accurately localize the vulnerable line and provides an incorrect explanation of the vulnerability cause. In contrast, only RIRE correctly identifies the vulnerability label, precisely localizes the vulnerable line, and provides an accurate explanation of the underlying cause.

The case shown in Figure 4 belongs to the Reveal dataset. All baseline methods, including GPT-4o, DeepSeek-R1, and SFT-CoT, predict incorrect vulnerability label. In contrast, only RIRE correctly identifies the vulnerability label, precisely localizes the vulnerable line, and provides an accurate explanation of the vulnerability cause.

In addition, while RIRE generally achieves more accurate vulnerability detection, a small number of failure cases still exist. Our analysis shows that, in many such cases, baseline methods produce en-

tirely incorrect predictions, whereas RIRE, despite making errors, still yields comparatively better results. Details are provided in Appendix B.

5 Conclusion

We propose a training framework for the CVD task called RIRE, which integrates refinement into the whole training process to enhance the performance and generalization. This framework first utilizes the patched code as refinement to guide the generation of reasoning chains for supervised fine-tuning. Subsequently, the refinement is employed to distinguish the quality of reasoning-enhanced detection results for hierarchical preference optimization. Experimental results show that RIRE outperforms strong baselines on several benchmark datasets and produces reliable explanations, thereby largely enhancing the generalization.

Limitations

In this study, we proposed a refinement-integrated reasoning training framework, RIRE, which is designed to help models capture deep features from source code to reasoning chains as explanations. However, it also faces several limitations: (1) **Limited language scope:** Our experiments are conducted on C/C++ benchmarks (DiverseVul and Devign), consistent with prior CVD studies. While extending the evaluation to other programming languages (e.g., Java, Python) would provide additional insights into cross-language generalization, this direction is beyond the scope of this work. We leave it for future exploration. (2) **Imperfect evaluation metrics for code vulnerability detection explanations:** Existing evaluation metrics in the vulnerability detection field are relatively limited and primarily focus on vulnerability label prediction. In our setting, the detection results not only include vulnerability labels but also involve reasoning chains about vulnerable line and underlying causes of vulnerabilities. Even when the vulnerability labels are correctly predicted, there is a strong need for systematic evaluation of the generated explanations to more comprehensively assess a model’s vulnerability detection capability. However, reliable and unified metrics for evaluating such explanations are currently lacking. As future work, we plan to explore trustworthy and standardized evaluation metrics and consider constructing a dedicated benchmark for explanation-aware vulnerability detection.

Acknowledgments

This work was supported by Noncommunicable Chronic Diseases-National Science and Technology Major Project (2023ZD0501806), Sichuan Science and Technology Program (2025ZNS-FSC1488), Fundamental Research Funds for the Central Universities (ZYGX2025XJ041), and CIPS-SMP-Zhipu Large Model Fund (CIPS-SMP20250314).

References

- Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefer. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Thirty-Eighth AAAI Conference on Artificial Intelligence*, pages 17682–17690.
- Tim Boland and Paul E. Black. 2012. Juliet 1.1 C/C++ and java test suite. *Computer*, pages 88–90.
- Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296.
- Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David A. Wagner. 2023. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 654–668.
- Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Trans. Softw. Eng. Methodol.*, 30(3):38:1–38:33.
- Xiaoxue Cheng, Junyi Li, Wayne Xin Zhao, and Ji-Rong Wen. 2024. Chainlm: Empowering large language models with improved chain-of-thought prompting. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics*, pages 2969–2983.
- Qianjin Du, Shiji Zhou, Xiaohui Kuang, Gang Zhao, and Jidong Zhai. 2023. Joint geometrical and statistical domain adaptation for cross-domain code vulnerability detection. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 12791–12800.
- Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. 2024. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. In *Findings of the Association for Computational Linguistics, ACL 2024*, pages 10507–10521.

- Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *17th International Conference on Mining Software Repositories*, pages 508–512.
- Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2024. Large language models for code analysis: Do llms really do their job? In *33rd USENIX Security Symposium*.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.
- Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225.
- Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations*.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR*, abs/2501.12948.
- Hazim Hanif, Mohd Hairul Nizam Bin Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. 2021. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *J. Netw. Comput. Appl.*, 179:103009.
- Namgyu Ho, Laura Schmid, and Se-Young Yun. 2023. Large language models are reasoning teachers. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14852–14882.
- Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations*.
- Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helgar, Aleksander Madry, Alex Beutel, Alex Carney, et al. 2024. Openai o1 system card. *CoRR*, abs/2412.16720.
- Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024a. A survey on large language models for code generation. *CoRR*, abs/2406.00515.
- Xuefeng Jiang, Lvhua Wu, Sheng Sun, Jia Li, Jingjing Xue, Yuwei Wang, Tingting Wu, and Min Liu. 2024b. Investigating large language models for code vulnerability detection: An experimental study. *CoRR*, abs/2412.18260.
- Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*.
- Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Trans. Dependable Secur. Comput.*, 19(4):2244–2258.
- Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. In *25th Annual Network and Distributed System Security Symposium*.
- Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *7th International Conference on Learning Representations*.
- Qing Lyu, Shreya Havaladar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pages 305–329.
- Lucie Charlotte Magister, Jonathan Mallinson, Jakub Adámek, Eric Malmi, and Aliaksei Severyn. 2023. Teaching small language models to reason. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1773–1781.
- Qiheng Mao, Zhenhao Li, Xing Hu, Kui Liu, Xin Xia, and Jianling Sun. 2024. Towards effectively detecting and explaining vulnerabilities using large language models. *CoRR*, abs/2406.09701.
- Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. Regvd: Revisiting graph neural networks for vulnerability detection. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings*, pages 178–182.
- Archiki Prasad, Swarnadeep Saha, Xiang Zhou, and Mohit Bansal. 2023. Receval: Evaluating reasoning chains via correctness and informativeness. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 10066–10086.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300.

Shichao Sun, Junlong Li, Weizhe Yuan, Ruifeng Yuan, Wenjie Li, and Pengfei Liu. 2024. The critique of critique. In *Findings of the Association for Computational Linguistics, ACL 2024*, pages 9077–9096.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

Martin Weyssow, Chengran Yang, Junkai Chen, Yikun Li, Huihui Huang, Ratnadira Widyasari, Han Wei Ang, Frank Liauw, Eng Lieh Ouh, Lwin Khin Shar, and David Lo. 2025. R2vul: Learning to reason about software vulnerabilities with reinforcement learning and structured reasoning distillation. *CoRR*, abs/2504.04699.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*.

Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. 2025. LIMO: less is more for reasoning. *CoRR*, abs/2502.03387.

Tianshu Yu, Chao Xiang, Mingchuan Yang, Pei Ke, Bosi Wen, Cunxiang Wang, Jiale Cheng, Li Zhang, Xinyu Mu, Chuxiong Sun, and Minlie Huang. 2025. Training language model to critique for better refinement. In *Findings of the Association for Computational Linguistics, ACL 2025*, pages 26760–26804.

Junwei Zhang, Zhongxin Liu, Xing Hu, Xin Xia, and Shanping Li. 2023. Vulnerability detection by learning from syntax-based execution paths of code. *IEEE Trans. Software Eng.*, 49(8):4196–4212.

Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, pages 10197–10207.

A Analysis Of Generated Patched Code

To ensure the quality of the generated patches, we conducted a human evaluation of the model-generated patches. Specifically, we randomly sampled 100 patches generated by the model, with an

Source Code <pre>static void rtas_get_time_of_day(PowerPCCPU *cpu, SPAPREnvironment *spapr, uint32_t token, uint32_t nargs, target_ulong args, uint32_t nret, target_ulong rets) { struct tm tm; if (nret != 8) { rtas_st(rets, 0, RTAS_OUT_PARAM_ERROR); return; } qemu_get_timedate(6tm, spapr- >rtc_offset); rtas_st(rets, 0, RTAS_OUT_SUCCESS); rtas_st(rets, 1, tm.tm_year + 1900); rtas_st(rets, 2, tm.tm_mon + 1); rtas_st(rets, 3, tm.tm_mday); rtas_st(rets, 4, tm.tm_hour); rtas_st(rets, 5, tm.tm_min); rtas_st(rets, 6, tm.tm_sec); rtas_st(rets, 7, 0); /* we don't do nanoseconds */ }</pre>	GPT-4o The presence of the 'nret' validation provides a direct mitigating factor against overflows or misuse of the 'rets' buffer. While further assumptions about external code (e.g., 'rtas_st') cannot be fully ruled out, the snippet..... [result_start] Secure [result_end] SFT-CoT Result: [Secure] Vulnerable line: None Cause: None Summary: The function demonstrates secure practices through robust input validation and safe memory access patterns. It validates the expected number of return values (nret) before processing..... RIRE Result: [Vulnerable] Vulnerable line: if (nret != 8) { Cause: The original code checks 'nret' (number of return values) instead of 'nargs' (number of arguments) to validate input. If 'nret' is incorrectly treated as the number of return values instead of the number of arguments, the check fails to enforce that the caller provides exactly 8 arguments. This allows the function to write to 'rets' (return buffer) beyond the allocated size when 'nret' is not 8, leading to a buffer overflow. Summary: Incorrect parameter count validation (using 'nret' instead of 'nargs') permits out-of-bounds memory access.
Vulnerability Label Vulnerable	

Figure 5: Representative failure cases. The problematic parts are highlighted in blue, whereas the correct parts are highlighted in red.

equal number of samples labeled as 1 and 0. For samples with label 1, we examined whether the generated patch successfully fixes the vulnerability in the original code without introducing new vulnerabilities. For samples with label 0, we assessed whether the code modifications introduced any vulnerabilities. After careful evaluation, we found that the correctness rate of the generated patches is 85%. This result demonstrates that the generated patches are of sufficiently high quality and supports the effectiveness of our method built upon the refinement process.

B Error Analysis

Among all failed cases, we identify two representative examples. As shown in Figure 5, the three baseline methods all fail to correctly classify the vulnerability, resulting in incorrect label predictions. By comparison, RIRE achieves better performance by correctly identifying the vulnerability label and precisely localizing the vulnerable lines. Nevertheless, its explanation of the vulnerability cause remains incomplete. Specifically, the actual root cause is insufficient parameter validation, as both nret and nargs should be validated instead of checking only nret.

As illustrated in Figure 6, the three baseline methods all produce incorrect vulnerability label predictions for this case. By contrast, RIRE correctly identifies the vulnerability label, demonstrating better performance than the baselines. Nevertheless, RIRE fails to accurately locate the vulnerable line and to identify the true root

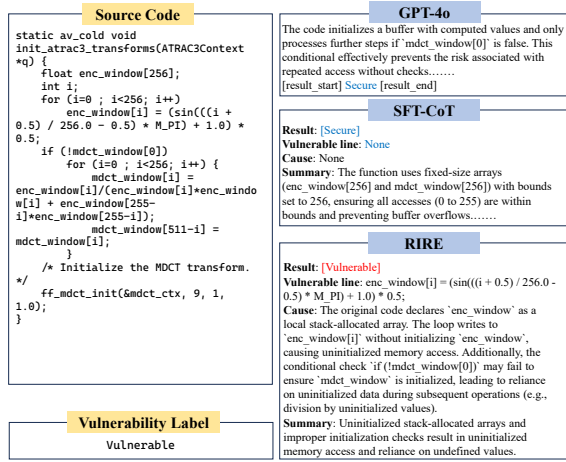


Figure 6: Representative failure cases. The problematic parts are highlighted in blue, whereas the correct part are highlighted in red.

cause. Specifically, the vulnerability occurs at `ff_mdct_init(&mdct_ctx, 9, 1, 1.0)`, and is caused by an improper assignment of the first argument to `ff_mdct_init`, where `&q->mdct_ctx` should be used instead of `&mdct_ctx`, ultimately resulting in a memory leak.

C Analysis on Model Scale

To investigate whether RIRE can be effectively applied to the base models with different scales, we conduct experiments using three versions of Qwen2.5-Coder with varying parameter sizes, i.e., 1.5B, 3B, and 7B. For comparison, we report the average performance of SFT, SFT-CoT, and RIRE across five datasets. From Figure 7, we observe that RIRE achieves the best performance on all the settings, demonstrating its adaptability to the models with different scales. Also, the performance of RIRE consistently improves as the model scale increases, showing notable scalability and the potential of generalizing to larger models.

D Prompt Template Design

This section presents all the prompt templates used in our method.

D.1 Refinement Collection

For refinement collection, some samples may not successfully have patched code due to the inaccessibility of original repositories. Thus, we resort to LLMs to directly generate patched code based on the source code and the corresponding vulnerability label as refinement. The prompt is shown in

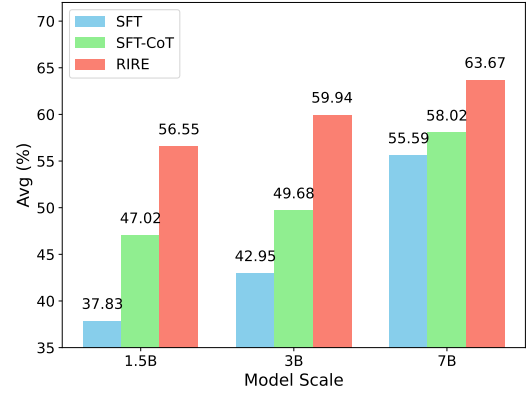


Figure 7: Average F1 scores of Qwen2.5-Coder with different scales across five benchmark datasets.

Figure 8.

D.2 Reasoning-Enhanced Fine-Tuning

For reasoning-enhanced fine-tuning, we first employ DeepSeek-R1 to generate a three-step reasoning chain as the explanation. The prompt is shown in Figure 9. Then we conduct supervised fine-tuning on the reasoning-enhanced dataset, where the prompts for model training on CodeLlama and Qwen2.5-Coder are shown in Figure 10 and 11, respectively.

D.3 Refinement-Oriented Hierarchical Preference Optimization

For refinement-oriented hierarchical preference optimization, we construct the preference dataset by generating paired data with opposite or consistent labels. The prompts for paired data generation with opposite labels are shown in Figure 12 and 13. As for paired data generation with consistent labels, we first utilize DeepSeek-R1 to generate two explanations conditioned on the correct label, where the prompts are shown in Figure 14 and 15. Then, we modify the source code via these two explanations, respectively, to acquire the patched code, where the prompts are shown in Figure 16 and 17. Finally, we judge the quality of two patched codes based on the refinement and output them in descending order of the quality. The corresponding prompts are shown in Figure 18 and 19.

Please modify the given code snippet and output the modified version. The output format should refer to the style of GitHub commit pages. Specific requirements will be described below. There are three types of code modifications: deletion only, addition only, and both deletion and addition. Note that when presenting the modified code, you must also include the context lines before and after the changes.

****Case 1: Deletion Only****
If one or more lines of code are deleted, add a "-" at the beginning of each deleted line. Also include one line before and one line after the deleted section as context. Both the deleted lines and the context lines must preserve the exact indentation of the original source code.

****Case 2: Addition Only****
If one or more lines of code are added, add a "+" at the beginning of each added line. To show where the addition occurs, you must include the line before and the line after the addition from the original code as context. The added lines and the surrounding context must retain the exact indentation as in the original source code.

****Case 3: Both Deletion and Addition****
If lines are both deleted and added, prepend deleted lines with "-" and added lines with "+". If the deletion and addition represent a replacement (i.e., the added code is a revised version of the deleted code), include the line before and the line after the change as context, and present the diff with consistent indentation from the source code. If the deletion and addition are unrelated (i.e., not a replacement), present them separately:
- The deleted section must include the deleted lines and their surrounding context, following Case 1.
- The added section must include the added lines and their surrounding context, following Case 2.

The output for all three types of changes should consist of 5 parts:

1. The header line: "@@ @@"
2. The function definition line from the original code
3. The context line before the modification
4. The modified line(s)
5. The context line after the modification

Below are examples for the three cases:

****Deletion Only Example:****

```
@@ -414,7 +414,6 @@ static int swf_write_trailer(AVFormatContext *s)\n        url_fseek(pb, swf->duration_pos,\n        SEEK_SET);\n        put_le16(pb, video_enc->frame_number);\n    }\n    av_free(swf);\n    return 0;\n}"
```

****Addition Only Example:****

```
"@@ -36,6 +36,7 @@ static int raw_read_packet(AVFormatContext *s, AVPacket *pkt)\n    ret= av_get_packet(s->pb, pkt,\n    size);\n    pkt->flags &= ~AV_PKT_FLAG_CORRUPT;\n    pkt->stream_index = 0;\n    if (ret < 0)\n        return ret;"
```

****Replacement Example:****

```
"@@ -242,15 +242,15 @@ static int v4l2_send_frame(AVCodecContext *avctx, const AVFrame *frame)\n    {\n        V4L2m2mContext *s = ((V4L2m2mPriv*)avctx->priv_data)->context;\n        V4L2Context *const output =\n        &s->output;"
```

Figure 8: The prompt template for generating patched code as refinement for the samples where the commit data cannot be acquired from GitHub.

```

Analyze the security of this function (label={label}):
{func} Code modification context: 1. Changes: {changes or 'No changes'} 2. Commit message: {message}
Task requirements:
{"Explain why the code CONTAINS vulnerabilities" if label == "1" else "Explain why this code is SECURE"} by:
{"Analyze the cause of the vulnerability based on the vulnerability-fix(if changes exist) and the description (by
comparing with the fixed code, focusing on the issues present in the original code). "
"Check whether the code lines mentioned in the commit message exist in the function. If the message refers to code that
does not appear in the function, ignore those parts. If the message describes a vulnerability pattern that is
semantically related to the current code logic, you may use the message as supporting information. Do not label code
lines as vulnerable if they are mentioned in the message but not present in the function. "
"Required elements:"
"  - Describe vulnerable line: if there are multiple vulnerable lines(from the vulnerability function), select a
representative line as the 'vulnerable line' rather than listing all lines"
"  - Describe the vulnerability cause, which should focus on describing the issues in the vulnerable code, not on how
the fix resolves them"
"  - Summarize the cause of the vulnerability"
if label == "1" else
"Code analysis (STRICTLY follow these rules):"
"  - If modification context is security-related (e.g., fixes vulnerabilities):"
"    * IGNORE modification context completely(Do NOT discuss modification context, removed/added lines, or commit
messages when explaining security properties.)"
"    * Analyze original code's security mechanisms ONLY"
"  - If modification is non-security:"
"    * Use modification context to confirm original code's security properties"
"    * DO NOT mention modification details in analysis"
"    * Cite original code's specific lines/structures as security evidence"
"  - Identify security mechanisms in the ORIGINAL code, analyze why the code does not contain security issues, consider
the following aspects: integer overflow, buffer overflow (out-of-bounds access, stack overflow), out-of-bounds
read/write, illegal memory access, memory leak, command injection, use-after-free, improper input validation, resource
leakage (sensitive information disclosure), null pointer dereference, input validation and error handling (improper
checks or handling of exceptional conditions), resource management errors, and logical defects."
"Required elements:"
"  - Highlight 1-2 secure code patterns: Explain why the code is safe, providing reasoning and referencing relevant
parts of the code. For instance, if the function involves pointer operations but performs null checks before usage,
ensuring null pointer dereference is avoided, mention this explicitly and refer to the relevant code section."}
Output format (MUST follow):
Vulnerable line: [vulnerable line / None]
Cause: [vulnerability cause / None]
Summary: [vulnerability cause summary / secure code patterns]
Result: {"[Vulnerable]" if label == "1" else "[Secure]"}

Examples:
Label=1 Example:
Vulnerable line: xxx\n Cause: xxx\n Summary: xxx\n Result: [Vulnerable]

Label=0 Example:
Vulnerable line: None\n Cause: None\n Summary: xxx\n Result: [Secure]

```

Figure 9: The prompt template for generating a three-step reasoning chain as explanations with DeepSeek-R1, including the vulnerable line, cause, and summary.

```

Below is an instruction that describes a task, paired with an input that provides further context.
Write a response that appropriately completes the request.
### Instruction:
Act as a security analyst and determine whether the following code is [Vulnerable] or [Secure]. Start
your response by immediately outputting the detection result as either [Vulnerable] or [Secure], using
the following strict format:
Result: [Vulnerable] or [Secure]
Vulnerable line: <the specific line that is vulnerable, or 'None' if secure>
Cause: <brieif explanation of the cause if vulnerable, or 'None' if secure>
Summary: <concise summary>
Result: [Vulnerable] or [Secure]
### Input:{input}
### Response:

```

Figure 10: The prompt template for reasoning-enhanced fine-tuning on CodeLlama.

```

< |im_start| > user
Below is an instruction that describes a task, paired with an input that provides further context.
Write a response that appropriately completes the request.
Instruction: Act as a security analyst and determine whether the following code is [Vulnerable] or
[Secure]. Start your response by immediately outputting the detection result as either [Vulnerable] or
[Secure], using the following strict format:
Result: [Vulnerable] or [Secure]
Vulnerable line: <the specific line that is vulnerable, or 'None' if secure>
Cause: <brief explanation of the cause if vulnerable, or 'None' if secure>
Summary: <concise summary>
Result: [Vulnerable] or [Secure]
Input: {input}
< |im_end| >
< |im_start| > assistant

```

Figure 11: The prompt template for reasoning-enhanced fine-tuning on Qwen2.5-Coder.

```

Given the following code {func}, it is known to be safe and does not contain any vulnerabilities (you do
not need to determine whether the code is safe-even if you think it is unsafe, treat it as safe). Please
identify the security mechanisms present in the original code and explain why the code does not pose any
security issues. Your analysis should consider the following aspects: integer overflow, buffer overflow
(out-of-bounds access, stack overflow), out-of-bounds read/write, illegal memory access, memory leak,
command injection, use-after-free, improper input validation, resource leaks (sensitive information
disclosure), null pointer dereference, improper input validation and error handling (failure to properly
check or handle exceptional conditions), resource management errors, and logic flaws. These aspects cover
nearly all areas of security. You only need to analyze 1-2 secure coding patterns and explain why the
code is secure. Provide a reasonable explanation and reference relevant parts of the code to support your
explanation (the relevant code should follow your explanation in the same sentence, with no line break).
For example: If a function involves pointer operations but includes a null pointer check before use,
effectively preventing null pointer dereference issues, clearly state this and indicate the corresponding
code. Your final output must follow the format below (do not use Markdown bold syntax such as **):
Vulnerable line: None
Cause: None
Summary: xxx
Result: [Secure]
The summary should include analysis of only one or two aspects and be limited to 70-90 words. Only one
analysis result is required.

```

Figure 12: The prompt template for generating a label-inconsistent reasoning chain as an explanation for vulnerable samples during the refinement-oriented hierarchical preference optimization stage.

Given the following code {func}, it is known to be unsafe and contains vulnerabilities (you do not need to determine whether the code is secure—even if you think it is safe, you must treat it as unsafe). Please explain why the code is unsafe. Consider the following aspects in your analysis: integer overflow, buffer overflow (out-of-bounds access, stack overflow), out-of-bounds read/write, illegal memory access, memory leak, command injection, use-after-free, improper input validation, resource leaks (sensitive information disclosure), null pointer dereference, improper input validation and error handling (failure to properly check or handle exceptional conditions), resource management errors, and logic flaws. These aspects cover nearly all areas of security. Focus on analyzing one or two of them and explain why the code is unsafe based on these aspects. Organize your output as follows: first, print the vulnerable line of code; then print the cause of the vulnerability (i.e., what security issue exists in that line); then provide a summary of the cause; and finally print the result. The cause and summary should each be limited to 60–80 words. Your final output must follow the format below:

Vulnerable line: xxx
Cause: xxx
Summary: xxx
Result: [Vulnerable]

Only one analysis result is required. Wrap the code in the Vulnerable line field using a single backtick `, and do not use Markdown bold syntax (e.g., **).

Figure 13: The prompt template for generating a label-inconsistent reasoning chain as an explanation for secure samples during the refinement-oriented hierarchical preference optimization stage.

Given the following code {func}, it is known to be unsafe and contains vulnerabilities (you do not need to determine whether the code is secure—even if you think it is safe, you must treat it as unsafe). Please explain why the code is unsafe. In your analysis, consider the following aspects: integer overflow, buffer overflow (out-of-bounds access, stack overflow), out-of-bounds read/write, illegal memory access, memory leak, command injection, use-after-free, improper input validation, resource leaks (sensitive information disclosure), null pointer dereference, improper input validation and error handling (failure to properly check or handle exceptional conditions), resource management errors, and logic flaws. These aspects nearly cover all areas of security. You only need to focus on analyzing one or two of them and explain why the code is unsafe in relation to these aspects. Organize your output as follows: first, print the vulnerable line of code; then print the cause of the vulnerability (i.e., what security issue exists in that line); then provide a summary of the cause; and finally print the result. The content of the cause and the summary should each be limited to 60–80 words. Your final output must follow the format below:

Vulnerable line: xxx
Cause: xxx
Summary: xxx
Result: [Vulnerable]

I need you to generate two different analysis results, each analyzing a different security issue. Wrap the code in the Vulnerable line field using a single backtick `, and do not use Markdown bold syntax (e.g., **).

Figure 14: The prompt template for generating two label-consistent reasoning chains as explanations for vulnerable samples during the refinement-oriented preference optimization stage.

Given the following code {func}, it is known to be safe and does not contain any vulnerabilities (you do not need to determine whether the code is safe-even if you think it is unsafe, treat it as safe). Please identify the security mechanisms present in the original code and explain why the code does not pose any security issues. Your analysis should consider the following aspects: integer overflow, buffer overflow (out-of-bounds access, stack overflow), out-of-bounds read/write, illegal memory access, memory leak, command injection, use-after-free, improper input validation, resource leaks (sensitive information disclosure), null pointer dereference, improper input validation and error handling (failure to properly check or handle exceptional conditions), resource management errors, and logic flaws. These aspects cover nearly all areas of security. You only need to analyze 1-2 secure coding patterns and explain why the code is secure. Provide a reasonable explanation and reference relevant parts of the code to support your explanation (the relevant code should follow your explanation in the same sentence, with no line break). For example: If a function involves pointer operations but includes a null pointer check before use, effectively preventing null pointer dereference issues, clearly state this and indicate the corresponding code. Your final output must follow the format below (do not use Markdown bold syntax such as ******):

Vulnerable line: None
Cause: None
Summary: xxx
Result: [Secure]

I need you to generate two different analysis results (in English), each with a different summary focusing on different secure coding aspects (each summary should analyze two aspects and be limited to 70-90 words, maintaining content coherence).

Figure 15: The prompt template for generating two label-consistent reasoning chains as explanations for secure samples during the refinement-oriented hierarchical preference optimization stage.

Given the following code:\n{func_code}\n, it is unsafe and contains a vulnerability. The vulnerability analysis consists of four parts: vulnerable line, root cause, summary, and result. The detailed content is as follows:
{explanation}

Based on the three parts-"vulnerable line", "root cause", and "summary"-please repair the given function-level code to ensure it becomes safe.

Note:

- The final output does not need to include the complete modified code snippet (the final output should be in English);
- Only provide the modified lines and necessary context. For example:
Delete 'int temp_flag = 0;' after the line 'else if (action == BUS_NOTIFY_ADD_DEVICE)'; (if the modification is code deletion)
Or add 'host_features |= (1 << VIRTIO_NET_F_MAC);' after the line 'int i, config_size = 0;'; (if the modification is code insertion)
Or replace 'switch (open_flags)' with 'switch (fmode)'; (if the modification is code replacement)
- Do not include any explanation;
- Do not use Markdown bold syntax (e.g., ******);

Figure 16: The prompt template for generating patched code based on the explanation for vulnerable samples during the refinement-oriented hierarchical preference optimization stage.

```

Given the following code:\n{func_code}\n, it is safe and contains no vulnerabilities. The following
analysis explains why this function-level code is safe (consisting of four parts: vulnerable line, root
cause, summary, and result):
{analysis}

Please focus on the "Summary" section and make modifications that do not affect the code's security,
ensuring the modified code remains safe.

Note:
- The final output does not need to include the complete modified code snippet (the final output should be
in English);
- Only provide the modified lines and their context. For example:
    Delete `int temp_flag = 0;` after the line `else if (action == BUS_NOTIFY_ADD_DEVICE)`; (if the
modification is code deletion)
    Or add `host_features |= (1 << VIRTIO_NET_F_MAC);` after the line `int i, config_size = 0;`; (if the
modification is code insertion)
    Or replace `switch (open_flags)` with `switch (fmode)`; (if the modification is code replacement)
- Do not include any explanation;
- Do not use Markdown bold syntax (e.g., **);

```

Figure 17: The prompt template for generating patched code based on the explanation for secure samples during the refinement-oriented hierarchical preference optimization stage.

```

Given this code:\n{func_code}\n, which is unsafe and vulnerable, I have two repair proposals for
how to fix the vulnerability in the current code:\n
Proposal 1: {vulnerable1}\n
Proposal 2: {vulnerable2}\n
Please score these two repair proposals and evaluate which one is better.
Additionally, I have a ground truth repair proposal as follows:\n{changes}\nPlease refer to the
ground truth repair proposal (first compare if the repair starting points are the same as the
ground truth; if one repair's starting point matches the ground truth and the other does not, the
one matching the ground truth is better; if both have the same starting point as the ground truth,
then compare the semantic similarity of the code implementations).
Please evaluate these two repair proposals. The final output format should be:
[evaluation_start] 1>2 [evaluation_end] (meaning the first repair proposal is better)
or [evaluation_start] 2>1 [evaluation_end] (meaning the second repair proposal is better)
or [evaluation_start] 1=2 [evaluation_end] (meaning the two repair proposals are equal, only if
they are exactly the same)
Note: Only output the evaluation result without any explanation or additional text.

```

Figure 18: The prompt template for quality judgment of two patched codes for vulnerable samples based on the refinement.

Given this code:\n{func_code}\n, which is safe and has no vulnerabilities, although there are no security vulnerabilities, there exist security-irrelevant modifications (such as feature modifications, syntax changes, adding comments, etc.). Please score the following two modification proposals and evaluate which one is better.

The two modification proposals are:\n

Proposal 1: {secure1}\n

Proposal 2: {secure2}\n

Please score and evaluate which proposal is better.

Additionally, I have a ground truth modification proposal as follows:\n{changes}\nPlease refer to the ground truth modification proposal (first compare if the modification starting points are the same as the ground truth; if one modification's starting point matches the ground truth and the other does not, the one matching the ground truth is better; if both have the same starting point as the ground truth, then compare the semantic similarity of the code implementations).

Please evaluate these two modification proposals. The final output format should be:

[evaluation_start] 1>2 [evaluation_end] (meaning the first modification proposal is better)

or [evaluation_start] 2>1 [evaluation_end] (meaning the second modification proposal is better)

or [evaluation_start] 1=2 [evaluation_end] (meaning the two modification proposals are equal, only if they are exactly the same)

Note: Only output the evaluation result without any explanation or additional text.

Figure 19: The prompt template for quality judgment of two patched codes for secure samples based on the refinement.