# Boosting Vulnerability Detection of LLMs via Curriculum Preference Optimization with Synthetic Reasoning Data

**Xin-Cheng Wen**[1], **Yijun Yang**[1], **Cuiyun Gao** [2*], **Yang Xiao**[3], **Deheng Ye**[1]
[1] Tencent Inc., China
[2] The Chinese University of Hong Kong, China
[3] Chinese Academy of Sciences, China
xiamenwxc@foxmail.com, yijun.steven.yang@gmail.com, cygao@cse.cuhk.edu.hk

## Abstract

Large language models (LLMs) demonstrate considerable proficiency in numerous coding-related tasks; however, their capabilities in detecting software vulnerabilities remain limited. This limitation primarily stems from two factors: (1) the absence of reasoning data related to vulnerabilities, which hinders the models' ability to capture underlying vulnerability patterns; and (2) their focus on learning semantic representations rather than the reason behind them, thus failing to recognize semantically similar vulnerability samples. Furthermore, the development of LLMs specialized in vulnerability detection is challenging, particularly in environments characterized by the scarcity of high-quality datasets. In this paper, we propose a novel framework ReVD that excels at mining vulnerability patterns through reasoning data synthesizing and vulnerability-specific preference optimization. Specifically, we construct forward and backward reasoning processes for vulnerability and corresponding fixed code, ensuring the synthesis of high-quality reasoning data. Moreover, we design the triplet supervised fine-tuning followed by curriculum online preference optimization for enabling ReVD to better understand vulnerability patterns. The extensive experiments conducted on PrimeVul and SVEN datasets demonstrate that ReVD sets new state-of-the-art for LLM-based software vulnerability detection, e.g., 12.24%-22.77% improvement in the accuracy. The source code and data are available at `https://github.com/Xin-Cheng-Wen/PO4Vul`.

## 1 Introduction

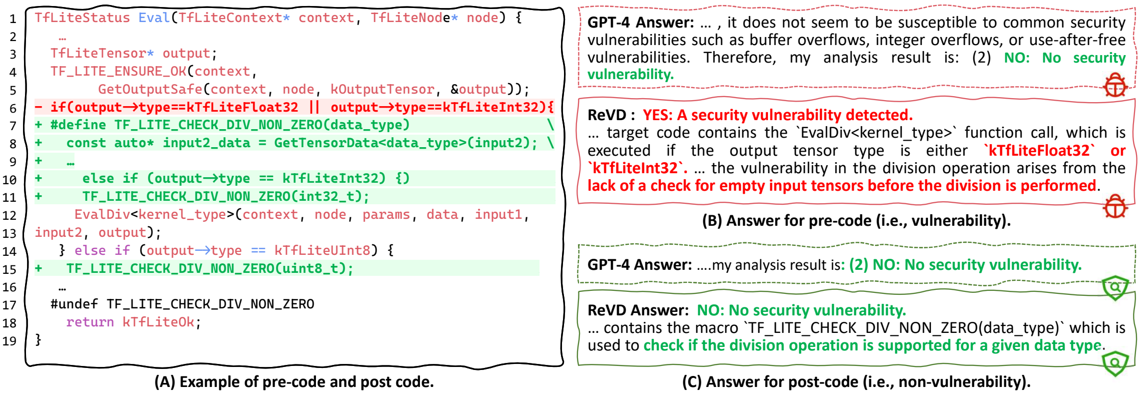
Software vulnerabilities, primarily due to insecure coding, pose significant risks as they can be exploited to compromise software systems, leading to severe security issues (Wen et al., 2023a, 2024). Thousands of software vulnerabilities are discovered per year (Statista, 2024), underscoring the critical importance of vulnerability detection methods (Zhou et al., 2019; Chakraborty et al., 2020; Cao et al., 2022). In recent years, Code Pre-Trained Models (CodePTMs) such as CodeBERT (Feng et al., 2020) and UniXcoder (Guo et al., 2022) have proven to be highly effective in identifying software vulnerabilities. They leverage extensive pre-training on diverse codebases to enhance their understanding and detection capabilities (Zheng et al., 2023). Following this paradigm, Large Language Models (LLMs) pre-trained on code data excel in comprehending and interpreting the semantics of both human and programming languages (Hou et al., 2023), offering superior intelligence and flexibility for coding-related tasks. Despite the substantial advancements of LLMs, their performance in vulnerability detection task is still limited mainly due to the following two aspects:

**(1) The absence of reasoning data behind vulnerability-fixing processes hinders the model from understanding and aligning with established vulnerability patterns.** The current approaches (Fu et al., 2023; Fu and Tantithamthavorn, 2022) solely utilize extensive pre-training across diverse codebases to enhance their vulnerability detection capabilities, yet acquiring specific reasoning knowledge about vulnerability-fixing processes remains challenging. These methods do not explicitly generate step-by-step reasoning data related to vulnerabilities for model training, and the thought processes of the individuals who fix vulnerabilities are not usually reflected in the dataset annotations. As illustrated in Figure 1 (A), we present an example of a Divide By Zero vulnerability (CWE, [n.d.]), where the code attempts to "`output->type == kTfLiteInt32`". However, this may inadvertently lead to a division by zero within the division operations in "`EvalDiv`". Therefore, the subsequent post-code employs the "`TF_LITE_CHECK_DIV_NON_ZERO`". This macro effectively checks for and prevents division by

---

[*] Corresponding author.

*Findings of the Association for Computational Linguistics: ACL 2025*, pages 8935–8949
July 27 - August 1, 2025 ©2025 Association for Computational Linguistics



```
TfLiteStatus Eval(TfLiteContext* context, TfLiteNode* node) {
   …
  TfLiteTensor* output;
  TF_LITE_ENSURE_OK(context,
        GetOutputSafe(context, node, kOutputTensor, &output));
- if(output→type==kTfLiteFloat32 || output→type==kTfLiteInt32){
+ #define TF_LITE_CHECK_DIV_NON_ZERO(data_type)             \
+   const auto* input2_data = GetTensorData<data_type>(input2); \
+   …
+     else if (output→type == kTfLiteInt32) {)
+     TF_LITE_CHECK_DIV_NON_ZERO(int32_t);
      EvalDiv<kernel_type>(context, node, params, data, input1,
input2, output);
   } else if (output→type == kTfLiteUInt8) {
+   TF_LITE_CHECK_DIV_NON_ZERO(uint8_t);
   …
  #undef TF_LITE_CHECK_DIV_NON_ZERO
    return kTfLiteOk;
}
```

**(A) Example of pre-code and post code.**

**GPT-4 Answer:** … , it does not seem to be susceptible to common security vulnerabilities such as buffer overflows, integer overflows, or use-after-free vulnerabilities. Therefore, my analysis result is: (2) **NO: No security vulnerability.**

**ReVD : YES: A security vulnerability detected.**
… target code contains the `EvalDiv<kernel_type>` function call, which is executed if the output tensor type is either **`kTfLiteFloat32` or `kTfLiteInt32`.** … the vulnerability in the division operation arises from the **lack of a check for empty input tensors before the division is performed**.

**(B) Answer for pre-code (i.e., vulnerability).**

**GPT-4 Answer:** ….my analysis result is**: (2) NO: No security vulnerability.**

**ReVD Answer: NO: No security vulnerability.**
… contains the macro `TF_LITE_CHECK_DIV_NON_ZERO(data_type)` which is used to **check if the division operation is supported for a given data type**.

**(C) Answer for post-code (i.e., non-vulnerability).**

Figure 1: **The vulnerability example of "Divide By Zero" (i.e., (CVE-2021-37683, 2021)).** *Figure (A):* The pre-code (i.e., labeled as vulnerable) and post-code (i.e., labeled as non-vulnerable). The code shaded in red and green denote the vulnerable and fixed code, respectively. *Figure (B):* GPT-4 and ReVD's answer for pre-code. *Figure (C):* GPT-4 and ReVD's answer for post-code.

zero errors in division operations, thereby also averting potential divide by zero vulnerability. It is evident that existing vulnerability data often includes pre- and post-code snippets and lacks comprehensive reasoning data necessary to understand the triggers of these vulnerabilities. Hence, GPT-4 fails to detect this vulnerability in Figure 1 (B). **(2) Existing methods tend to focus on learning semantic representations rather than vulnerability patterns, which fail to distinguish the vulnerability from the fixed one due to the highly semantic similarity of the code.** Vulnerability patches often involve subtle code changes such as adjusting buffer sizes, correcting data types, or adding security checks (Luo et al., 2024). These modifications typically result in a post-code (i.e., non-vulnerable) that is semantically similar to their pre-code (i.e., vulnerable). For instance, as shown in Figure 1 (B) and (C), GPT-4 does not discern the subtle differences between the vulnerability and the corresponding fixed code and generates the same answer for them. More specifically, GPT-4 fails to distinguish 78.62% pairs of vulnerabilities and corresponding fixed code in realistic vulnerability detection scenarios from the PrimeVul (Ding et al., 2024) dataset.

To mitigate the above issues, in this paper, we propose a novel framework ReVD that excels at mining vulnerability patterns through vulnerability-specific preference optimization with reasoning-centric data synthesis. Specifically, we purposefully design three modules: **(1) Bi-directional Vulnerability Data Generation (BVD).** It consists of forward and backward reasoning processes for vulnerability and corresponding fixed code and utilizes the vulnerability information to generate a high-quality vulnerability reasoning dataset with 28k samples. **(2) Triplet Supervised Fine-Tuning (T-SFT).** We propose a novel triplet loss function by analyzing the relationships among pre-code, post-code, and code-diff. Fine-tuning LLMs with such a loss enhances their ability to capture vulnerability patterns as it enables consistency checks between their forward and backward reasoning. **(3) Curriculum Online Preference Optimization (COPO).** The LLM focuses on learning the vulnerability patterns on which it underperforms via iterative preference optimization with instance- and task-level curricula, continuously enhancing its effectiveness in real-world complex scenarios.

We evaluate ReVD and compare it with nine representative CodePTMs and LLM-based vulnerability detection baselines. The extensive evaluations highlight that ReVD outperforms all baselines and sets new state-of-the-art across two widely-used datasets PrimeVul (Ding et al., 2024) and SVEN (He and Vechev, 2023), with improvements of 9.08%, 10.33%, and 18.15% in terms of accuracy, F1-Score, and VP-Score, respectively. More importantly, we also release the first vulnerability reasoning dataset and the corresponding preference dataset, shedding novel insights on training more generalizable and versatile LLM experts for vulnerability detection in real-world scenarios. **The major contributions of this paper are summarized below:**

**Reasoning Data:** We draw inspiration from the surge of large reasoning models (Guo et al., 2025; Cai et al., 2024, 2025), find the substantial benefits of reasoning data for vulnerability detection, and propose the first fully automated pipeline to synthesize vulnerability reasoning data.

**Method:** We develop a novel two-stage framework ReVD on our synthetic reasoning data, which can turn open-source LLMs into strong vulnerability detectors by aligning their coding knowledge with vulnerability semantics, patterns, and types.

**Results:** The results demonstrate that ReVD sets new state-of-the-art for LLM-based software vulnerability detection, e.g., 12.24%-22.77% improvement in the accuracy.

## 2 Related Work

**Software Vulnerability Detection:** CodePLMs (Fu and Tantithamthavorn, 2022; Feng et al., 2020; Zhang et al., 2023a) and LLMs (OpenAI, 2023; ChatGPT, 2022) are widely used in vulnerability detection and can be classified into prompt-based and fine-tuning-based methods. The prompt-based method (Fu et al., 2023; Steenhoek et al., 2024; Zhou et al., 2024) has demonstrated effective performance on datasets specific to certain types of vulnerabilities. For instance, Li et al. (2024) propose Lift, which integrates static analysis with LLMs and focuses particularly on the detection of use-before-initialization vulnerabilities in the Linux kernel. Similarly, Sun et al. propose LLM4Vuln (Sun et al., 2024) and only target the smart contract vulnerabilities. Additionally, Ding et al. (Ding et al., 2024) explored the use of chain-of-thought prompts, which have shown promising results in enhancing detection capabilities. The fine-tuning-based (Wen et al., 2024; Fu and Tantithamthavorn, 2022) approaches add a randomly initialized binary classification head to the language model and jointly optimize all weights based on ground-truth labels. With the development of the Transformer architecture, encoder-only Transformers, such as CodeBERT (Feng et al., 2020), and encoder-decoder Transformers, such as UniXcoder (Guo et al., 2022), serve as foundational backbones that are further fine-tuned to enhance their efficacy in identifying software vulnerabilities. Additionally, recent studies have explored the vulnerability capabilities of larger LMs (Du et al., 2024), such as code-llama (Rozière et al., 2023) and llama (Touvron et al., 2023). These approaches mainly learn representations from the semantic perspective (Cheng et al., 2022). In this work, we propose the first fully automated pipeline to synthesize vulnerability reasoning data. Furthermore, we develop a novel two-stage framework ReVD on our synthetic reasoning data.

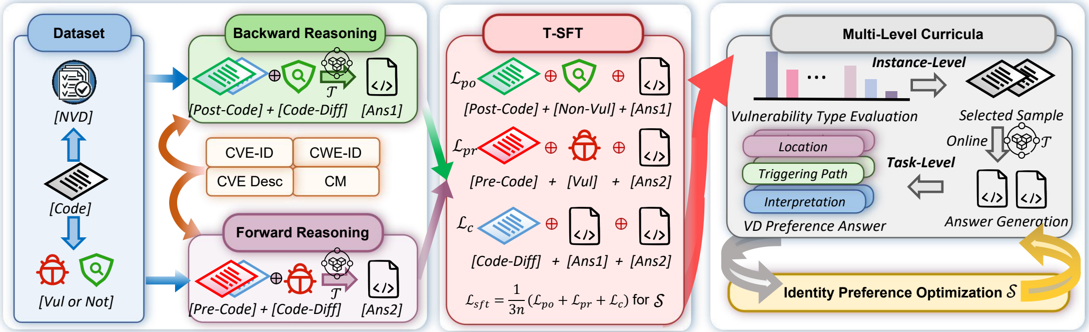
**Preference Optimization for Code:** Preference optimization techniques have recently been widely used in enhancing the output quality of LLMs across various natural language-related tasks (Xu et al., 2024; Austin et al., 2024; Xiao et al., 2024). These techniques have also been extended to code-related tasks, such as code generation (Zhang et al., 2024). For instance, Code-Optimize (Gee et al., 2024) has adopted Direct Preference Optimization (DPO) (Rafailov et al., 2023) as its preferred fine-tuning method, owing to its simplicity and broad acceptance. PPOCoder (Shojaee et al., 2023) has introduced a novel framework for code generation that effectively integrates CodePTMs with Proximal Policy Optimization (PPO) (Schulman et al., 2017). Additionally, PLUM (Zhang et al., 2023b) has developed methods to generate tests and utilize Knowledge Transfer Optimization (KTO) (Ethayarajh et al., 2024). However, there have been no studies exploring the application of preference optimization to address vulnerability-related tasks. This gap can be attributed to the inherent need for objective correctness in vulnerability management (Nyanchama, 2005), contrasting with the often subjective nature of preferences in natural language tasks. Moreover, constructing vulnerability data is costly and challenging to procure (Croft et al., 2023; Wen et al., 2023b), complicating efforts to generate feedback through test cases. In this paper, we design the multi-level curricula and optimization phases to enhance the effectiveness of vulnerability-specific preference optimization.

## 3 Proposed Framework

We provide an overview of ReVD workflow in Figure 2. ReVD mainly consists of three modules: (1) Bi-directional Vulnerability Data (BVD) Generation, (2) Triplet Supervised-Fine-Tuning (T-SFT), and (3) Curriculum Online Preference Optimization (COPO).

### 3.1 BVD Generation

For vulnerability detection, ReVD is designed to enhance the vulnerability detection capabilities in real-world scenarios, where the detector determines whether a code snippet contains a vulnerability and further explains the reason. However, directly deploying LLMs $\mathcal{T}$ for vulnerability detection in real-



**(1) Bi-directional Vulnerability Data Generation** **(2) Triplet SFT** **(3) Curriculum Online Preference Optimization**

Figure 2: **The overview of ReVD.** *First,* ReVD takes a vulnerable code and its corresponding fixed code as inputs to generate backward and forward reasoning answers by model $\mathcal{T}$. *Then*, we train the model $\mathcal{S}$ by the pair of pre-code, post-code, code-diff, and corresponding answers. *Finally*, we undergo multiple rounds of task- and instance-level preference data construction for COPO to optimize model $\mathcal{S}$ continuously.

world scenarios is still challenging due to the lack of high-quality reasoning data related to vulnerabilities in their pre-training corpus. To bridge this gap, we use the original vulnerability dataset $\mathcal{D}$ with the model $\mathcal{T}$ to generate the vulnerability interpretation data by forward and backward reasoning to produce an augmented dataset $\mathcal{D}_{\text{aug}}$. Specifically, we select a vulnerability patch dataset denoted as $\mathcal{D} = \{(Pr_c^{(i)}, Po_c^{(i)})\}_{i=1}^{n}$ of $n$ samples, where each sample comprises a pre-code $Pr_c^{(i)}$ (i.e., labeled as vulnerable) and its corresponding post-code $Po_c^{(i)}$ (i.e., labeled as non-vulnerable).

Following the prior work (Chen et al., 2024), we design both forward and backward reasoning processes. The *forward reasoning* process aims to deduce the causes that trigger the vulnerability in the code. In contrast, the *backward reasoning* process seeks to understand why the code changes in the fixed code prevent the recurrence of the vulnerability and outlines the steps taken to rectify the vulnerability. As illustrated in Figure 2, the model analysis leverages vulnerability information, including code-diff, CVE-ID (CVE, 2024), CWE-ID (CWE, 2024), CVE descriptions, and commit messages from the NVD (nvd, 2024). These descriptions provide relevant background and context of the vulnerabilities and their fixes, which are beneficial for understanding the root causes. Specifically, ReVD takes a code, corresponding code change in the patch, and vulnerability information as inputs to generate forward and backward reasoning traces. The detailed example is shown in Appendix F. We have the augmented vulnerability reasoning dataset $\mathcal{D}_{\text{aug}}$, where each data in $\mathcal{D}_{\text{aug}}$ consists of $(Pr_c^{(i)}, Po_c^{(i)}, Pr_a^{(i)}, Po_a^{(i)}, Di_c^{(i)}, Di_a^{(i)})$. It denotes the pre-code, forward reasoning answer for pre-code, post-code, backward reasoning answer for post-code, code-diff, and the corresponding answer, respectively.

## 3.2 Triplet Supervised Fine-Tuning

In order to enhance the LLM's ability to capture vulnerability patterns, we analyze the relationship among pre-code, post-code, and code-diff and construct a triplet loss to fine-tune the model $\mathcal{S}$ with the previously augmented dataset $\mathcal{D}_{\text{aug}}$ as below:

$$\mathcal{L}_{\text{T-SFT}} = \mathbb{E}_{\mathcal{D}_{\text{aug}}}\Big[\underbrace{\ell(\mathcal{S}(Q, Pr_c^{(i)}), Pr_a^{(i)})}_{\text{(i) Pre-code}} + \underbrace{\ell(\mathcal{S}(Q, Po_c^{(i)}), Po_a^{(i)})}_{\text{(ii) Post-code}} + \underbrace{\ell(\mathcal{S}(Q, Di_c^{(i)}), Di_a^{(i)})}_{\text{(iii) Code-Diff}}\Big] \quad (1)$$

where $Q$ denotes the question for vulnerabiliy detection and $\ell$ is the cross-entropy loss. Specifically, the loss $\mathcal{L}_{\text{T-SFT}}$ is composed of three losses that make full use of our augmented data: (i) the pre-code part for learning vulnerability patterns, (ii) the post-code part for learning how to fix the vulnerability, and (iii) the code-diff part for focusing on vulnerability-specific code changes rather than the similar code semantics. We aim to let the model concentrate on the vulnerability patterns in a multi-task learning approach. Different components in Eq. 1 are treated equally because understanding how vulnerabilities are triggered and how to fix them is equally important for vulnerability detection.

**Algorithm 1** Training Procedure of COPO [3.3]

```
1:  initialize: $r = 0$, $\mathcal{D}^{(0)} \leftarrow \varnothing$, Model $\pi \leftarrow s$, Regularize-Parameter $\tau \in R^{+}$, Type List $l$
2:  input: Round $\mathcal{C}$, Generation Model $\mathcal{T}$, Eval Set $E$, Training Set $\mathcal{E}$, Reference Policy $\pi_{\text{ref}}$,
3:  while $r < C$ do
4:      Test $E$ via $s$                ▷ Instance-level curriculum
5:      Get $acc_r = [t_1^r, t_2^r \ldots, t_n^r]$ via $l$
6:      $probabilities \leftarrow [1 - t_i^r$ for $t_i^r$ in $acc_r]$
7:      $S_s^{(r)} \leftarrow \{\}$
8:      for $data \in \mathcal{D}$ do
9:          $type_d \leftarrow$ Get_vul_type$(data, l)$
10:         $p_d \leftarrow probabilities[type_d]$
11:         $r_d \leftarrow$ random between 0 and 1
12:         if $r_d < p_d$ then
13:             $S_s^{(r)} \leftarrow S_s^{(r-1)} \cup \{data\}$
14:     $\mathcal{D}^{(r)} = (y_w, y_l, \bigcup_1^r x_r)) \leftarrow S_s^{(r)}$
15:     Decomposition                   ▷ Task-level curriculum
16:     Generate the preferred and dispreferred generations $y_w$ and $y_l$ via $\mathcal{T}$ and
17:     Optimization Process by Eq. (2) and (3) with $y_w$, $y_l$, $\pi$, $\pi_{\text{ref}}$ and $\tau$   ▷ Optimization
18:     $r \leftarrow r + 1$
19: output: $\pi_{\theta^*}$
```

### 3.3 Curriculum Online Preference Optimization (COPO)

To correctly distinguish vulnerabilities with fixed code, we develop the COPO module that is structured into two distinct phases: **Curriculum** and **Optimization**, as elaborated in our Algorithm 1.

**Multi-level Curricula.** It is evident that instances of different vulnerability types are unevenly distributed, as shown in Appendix B. The imbalanced distribution can lead to suboptimal performance, which stimulates the development of a curriculum-based method focusing more on types of vulnerabilities. According to the features of vulnerability detection, we build instance-level and task-level curricula. *(1) Instance-level curriculum.* We first construct a list of vulnerability types based on the CWE (CWE, 2024), denoted as $L$ and divide an evaluation set $E$. Detailed information is shown in Table B. Based on the current accuracy of the model $S$ (as mentioned in Sec. 3.2) in identifying different vulnerabilities within $E$, we filter instances in Lines 4-6. The lower the accuracy for the vulnerability type, the more likely it is that instances of this type from dataset $D$ will be selected as preference instances $data$ (Lines 6-13). The instance $data$ for the current round is then merged with the instances from the previous iteration $S_s^{(r-1)}$, to form a new preference dataset $S_s^{(r)}$. *(2) Task-level curriculum.* Then, to mitigate the scarcity of vulnerability data, we further generate a 3x larger dataset by breaking down the established one. In particular, based on three easy-to-hard vulnerability comprehension tasks, we decompose each preference instance into three different tasks: location of the vulnerable line, trigger path of program analysis, and root cause interpretation. We also use the BVD to regenerate instances by model $\mathcal{T}$ (as mentioned in Section 3.1). For each pre-code, we denote the explanation data as the referred responses $y_w$ and the corresponding fixed code as dispreferred responses $y_i$. It is important to note that each instance is exclusively utilized for one task in any iteration round, although the same instance may be employed across different rounds and for varied task assignments.

**Optimization.** Finally, we improve Identity Preference Optimization (IPO) (Azar et al., 2024). It avoids model $\mathcal{S}$ to overfit the preference data in the limited vulnerability dataset. Based on the IPO, we iteratively optimize the curriculum criteria each round which is calculated as follows:

$$h_\pi(y_w, y_l, x) = \log\left(\frac{\pi(y_w|\bigcup_1^r x_r)\pi_{\text{ref}}(y_l|\bigcup_1^r x_r)}{\pi(y_l|\bigcup_1^r x_r)\pi_{\text{ref}}(y_w|\bigcup_1^r x_r)}\right) \tag{2}$$

$$\mathop{\mathbb{E}}_{(y_w,y_l,x)\sim D}\left(h_\pi(y_w, y_l, x) - \frac{\tau^{-1}}{2}\right)^2. \tag{3}$$

Specifically, COPO iteratively optimizes the model to align with preferences data, which regress the gap between log-likelihood ratios $\log(\pi(y_w)/\pi(y_l))$ and $\log(\pi_{\text{ref}}(y_w)/\pi_{\text{ref}}(y_l))$ to regularization parameter $\frac{\tau^{-1}}{2}$.

## 4 Experimental Setup

In vulnerability detection, the existing methods focus on C/C++ due to their widespread use in real-world programming and the presence of well-known vulnerabilities that vulnerability researchers correctly label. We choose two widely-used and higher-quality vulnerability datasets, including PrimeVul (Ding et al., 2024), and SVEN (He and Vechev, 2023), to provide a more accurate evaluation under real-world conditions. PrimeVul employs a data labeling technique and contains 140 CWEs and 6,968 vulnerable samples across 755 projects and 6,827 commits. We classified the vulnerability types based on the CWE (CWE, 2024)

Table 1: **Evaluation results of ReVD compared with vulnerability detection baselines on the PrimeVul and SVEN datasets.** COT = Chain of Thought. SFT = Supervised Fine-Tuning. The prompt template used in our experiments follows the approach outlined by Ding et al. (Ding et al., 2024). The **highest score** for each metric in the same dataset are highlighted in **bold** text. The (↑) / (↓) represents the performance of the ReVD compared with the best-performing method on this metric. **ReVD significantly surpasses all SOTA baselines in the PrimeVul and SVEN datasets. When integrated with Qwen2.5-Coder-7B-Instruct, ReVD achieves the highest performance.**

| Method | Dataset Type | PrimeVul (Ding et al., 2024) | | | SVEN (He and Vechev, 2023) | | |
|---|---|---|---|---|---|---|---|
| | | Accuracy ↑ | F1 Score ↑ | VP-S ↑ | Accuracy ↑ | F1 Score ↑ | VP-S ↑ |
| CodeBERT (Feng et al., 2020) | SFT | 51.03 | 16.47 | 2.07 | 51.77 | 49.79 | 3.53 |
| UniXCoder (Guo et al., 2022) | SFT | 49.89 | 16.48 | -0.23 | 51.90 | 43.45 | 3.80 |
| LineVul (Fu and Tantithamthavorn, 2022) | SFT | 49.77 | 17.70 | -0.46 | 51.90 | 40.40 | 3.80 |
| Llama3.1-70B-Instruct (Dubey et al., 2024) | COT | 49.77 | 55.90 | -0.46 | 50.00 | 31.34 | 0.00 |
| Qwen2.5-32b-Coder-Instruct (Hui et al., 2024) | COT | 50.00 | 26.64 | 0.00 | 51.36 | 29.25 | 2.72 |
| GPT-4 (OpenAI, 2023) | COT | 51.72 | 43.40 | 3.44 | 51.36 | 56.76 | 2.72 |
| Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) | COT | 49.77 | 29.86 | -0.46 | 52.31 | 27.63 | 4.62 |
| | **ReVD** | **58.05** (↑6.33%) | 63.83(↑7.93%) | **16.09** (↑12.65%) | **63.72** (↑11.82%) | **69.49** (↑12.73%) | **27.44** (↑23.64%) |
| Llama-3.1-8B-Instruct (Dubey et al., 2024) | COT | 48.16 | 40.58 | -3.68 | 50.00 | 46.82 | 0.00 |
| | ReVD | 56.21 (↑4.49%) | 54.91 (↓-0.99%) | 12.41(↑8.97%) | 61.82 (↑9.92%) | 60.70 (↑3.94%) | 23.65 (↑19.85%) |
| StarCoder2-7B (Lozhkov et al., 2024) | COT | 50.11 | 0.91 | 0.23 | 50.00 | 0.00 | 0.00 |
| | ReVD | 55.63 (↑3.91%) | **68.74**(↑12.84%) | 11.26(↑7.82%) | 57.34(↑5.44%) | 53.96(↓2.80%) | 14.68(↑10.88%) |

and GraphSPD (Wang et al., 2023). As shown in Appendix 5, the ratio distribution is imbalanced and contains different probabilities for instance-level curriculum. SVEN dataset manually vets vulnerabilities from multiple repositories, containing 416 vulnerable and fixed codes from real-world C/C++ projects, with the highest 94% reported accuracy.

### 4.1 Metrics

Following the previous work (Wen et al., 2023a; Fu and Tantithamthavorn, 2022), we choose the Accuracy and F1 to evaluate ReVD's performance. Because the F1 may bias towards models which predict vulnerable more often (Zhou et al., 2024), we propose the Vulnerability Pair-Score (VP-S= $\frac{Correct_{pair} - Wrong_{pair}}{All_{pair}}$) to evaluate the performance for vulnerability detection. $Correct_{pair}$ and $Wrong_{pair}$ denote the correctly and inversely predict the ground-truth labels, respectively.

### 4.2 Training and Inference Details

For the training of T-SFT, we employ the full SFT and train each LLM for three epochs. In contrast, for COPO, we utilize the LoRA (Hu et al., 2024), which updates only a subset of the parameters in the base model while keeping the remaining components unchanged. We conduct three rounds of preference optimization, and each round is trained for one epoch. The maximum sequence length is set to 2048 tokens. Further details about the hyperparameters used in our training procedure are available in Table 4 of the Appendix C.

## 5 Experimental Results

### 5.1 Comparison with State-of-the-Art

In this section, we compare ReVD with nine other representative vulnerability detection methods on the PrimeVul and SVEN datasets, spanning three widely-used vulnerability detection methods, three larger LLMs (exceeding 30B), and three base models. Detailed in Appendix A.

**(1) ReVD sets new SOTA performance in software vulnerability detection.** The experimental results presented in Table 1 demonstrate that ReVD consistently outperforms all baseline methods across all datasets. Specifically, when integrated into the Qwen2.5-Coder-7B-Instruct, ReVD exhibits superior performance across all metrics. It achieves an accuracy of up to 58.05%, an F1 score of 63.83%, and a VP-Score of 16.09% on the PrimeVul dataset. This achievement underscores the effectiveness of ReVD in capturing vulnerability patterns across complex real-world scenarios, encompassing 140 types of vulnerabilities. Furthermore, ReVD exhibits a higher accuracy of 63.72%, an F1 score of 69.49%, and a VP-Score of 27.44% on the SVEN dataset. This indicates that ReVD is equally robust on other datasets that do not require training. Notably, the SVEN dataset includes only nine types of vulnerabilities, which may contribute to the enhanced performance of ReVD.

**(2) ReVD is versatile across a variety of models.** We select two types of LLMs, including the CodeLLMs (i.e., Qwen2.5 and StarCoder2) and the general LLM (i.e., LLama-3.1), as base models to validate the extensive applicability of ReVD. As illustrated in Table 1, ReVD's performance remains

Table 2: **Ablation study.** The experimental results of ReVD and corresponding variants in PrimeVul and SVEN datasets. The "*w/o BVD + T-SFT*" uses the original data and the full supervised fine-tuning for training. The "*w/o COPO*" replaces the COPO (described in Algorithm 1). Please refer to Section. 5.2 for a thorough discussion.

| **Base Model** | **Dataset Varient** | **PrimeVul** | | | **SVEN** | | |
|---|---|---|---|---|---|---|---|
| | | **Accuracy ↑** | **F1 Score ↑** | **VP-S ↑** | **Accuracy ↑** | **F1 Score ↑** | **VP-S ↑** |
| GPT-4 (OpenAI, 2023) | - | 51.72 | 43.40 | 3.44 | 51.36 | 56.76 | 2.72 |
| Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) | w/o BVD + T-SFT | 51.03 | 28.28 | 2.07 | 50.00 | 0.00 | 0.00 |
| | w/o COPO | 55.75 | 56.50 | 11.49 | 54.21 | 61.83 | 8.43 |
| | ReVD | **58.05** | **63.83** | **16.09** | **63.72** | **69.49** | **27.44** |
| Llama-3.1-8B-Instruct (Dubey et al., 2024) | w/o BVD + T-SFT | 50.00 | 0.00 | 0.00 | 50.00 | 0.00 | 0.00 |
| | w/o COPO | 55.16 | 59.79 | 10.34 | 60.87 | 63.82 | 21.74 |
| | ReVD | **56.21** | **54.91** | **12.41** | **61.82** | **60.70** | **23.65** |
| StarCoder2-7B (Lozhkov et al., 2024) | w/o BVD + T-SFT | 50.00 | 0.00 | 0.00 | 50.00 | 0.00 | 0.00 |
| | w/o COPO | 52.87 | 58.59 | 5.74 | 54.21 | 61.83 | 8.43 |
| | ReVD | **55.63** | **68.74** | **11.26** | **57.34** | **53.96** | **14.68** |

significantly superior to previous methods in most cases, regardless of the base model. This can be attributed to ReVD's guide model to capture vulnerability patterns in code changes, focusing on the triggering reasoning of vulnerabilities rather than merely their semantics. On the other hand, some vulnerabilities that are not typically well-handled by LLMs are directly addressed by COPO, which ensures performance enhancements. This finding highlights the potential advantages of ReVD in real-world scenarios, where the types and samples of vulnerabilities are continuously expanding.

### 5.2 Ablation Study

In this section, we explore the impact of different modules of ReVD across two datasets, including the BVD, T-SFT, and COPO. The experimental results are shown in Table 2.

**(1) Bi-directional vulnerability reasoning data improves the reasoning abilities of LLMs for software vulnerability detection.** To assess the importance of the bi-directional vulnerability interpretation data, we deploy one variant (i.e., w/o BVD + T-SFT) by using the original data proposed by PrimeVul. Due to the T-SFT being designed for BVD, this variant only uses the full supervised fine-tuning for training. The results show that ReVD with the bi-directional vulnerability interpretation data significantly outperforms its counterpart. It averagely improves 6.29% of PrimeVul, and 10.96 % of SVEN in accuracy. This finding is crucial as it indicates that existing LLMs can reason about vulnerabilities, but rather lack the domain-specific knowledge required to effectively invoke such reasoning. BVD and T-SFT are fundamental components of ReVD, providing accurate interpretations of vulnerabilities and contributing substantially to its enhanced performance.

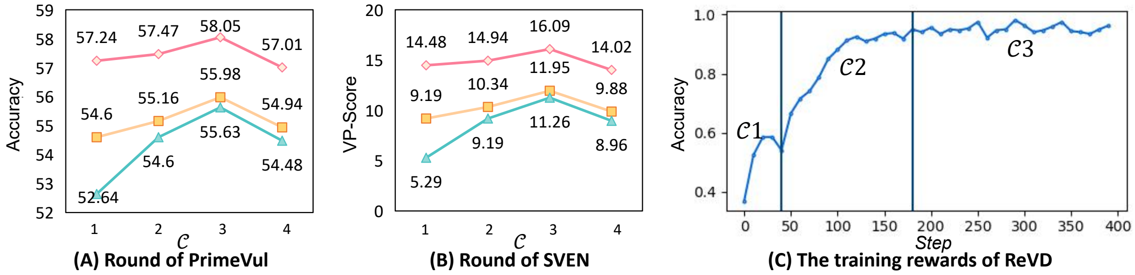
**(2) COPO enables more effective vulnerability detection.** To evaluate the effectiveness of COPO in Algorithm 1, we also conduct a variant without the COPO (i.e., w/o COPO). The results reveal that the variant average exhibits a decrease of 2.04% on Primevul and 4.53% on SVEN in accuracy, suggesting that COPO contributes to enhancing the upper performance bound of ReVD. This improvement can be attributed to COPO's ability to effectively identify the samples and vulnerability types that the current model struggles with. It then strategically selects these samples in subsequent training phases, thereby optimizing the performance of ReVD.

### 5.3 COPO Round

Considering the limited labeled data of vulnerability and training efficiency, we conduct experiments with the number of training rounds in COPO, which are proportional to the training time. As shown in Figure 3 (A) and (B), we observe that the ReVD performance increases as the round increases in the initial phase, reaching its peak at round 3. However, further increases in training rounds result in a decline in performance. It can be attributed to the limited pool of training samples, despite our efforts to augment the data through task answer decomposition and setting the temperature to 1 for generating interpretative answers online. In addition, we also find that the reward accuracies have Furthermore, we observe that the reward accuracies exhibit oscillations in round 3 in Figure 3 (C). The continuation of curriculum training has a minimal impact on this situation.

### 5.4 Token Length

The capability of LLMs to accurately detect vulnerabilities across varying code lengths is essential for addressing real-world challenges. To evaluate



**(A) Round of PrimeVul** **(B) Round of SVEN** **(C) The training rewards of ReVD**

Figure 3: **The impact of different rounds of COPO.** *Left*: The performance on PrimeVul. *Middle*: The performance on SVEN. *Right*: The training reward accuracies of ReVD across COPO steps. The pink, orange, and green lines denote the ReVD involved with Qwen2.5-Coder-7B-Instruct, Llama-3.1-8B-Instruct and StarCoder2-7B metrics, respectively. $C$ denotes the number of **C**urriculum Round.

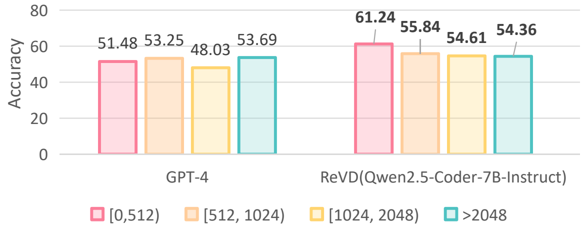


Figure 4: **The accuracy of ReVD under varying token length.** The pink, orange, yellow, and green denote the performance for token numbers containing less than 512 tokens, between 512 and 1024 tokens, between 1024 and 2048 tokens, and more than 2048 tokens.

this, we conduct a supplementary experiment on PrimeVul, where we classify the token lengths of examples into four categories: less than 512 tokens (ratio of 38.85%), 512-1024 tokens ((ratio of 26.55%), 1024-2048 tokens (ratio of 17.47%), and more than 2048 tokens (ratio of 17.13%).

We compare the performance of ReVD with the best-performing LLM, GPT-4, and the results are depicted in Figure 4. We observe that ReVD exhibits a slight decline in performance as the number of tokens increases, yet it significantly outperforms GPT-4 across all token ranges. Additionally, we find the following observations: (1) ReVD, integrated with Qwen-2.5-7B-Instruct, generally performs best in most scenarios (in 3 out of 4 cases). (2) The capability for vulnerability detection remains relatively consistent across the 512-2048 token range, with a notable decline only occurring beyond 2048 tokens, indicating a need for more data involving longer token lengths. The complete results of ReVD are provided in Appendix D.

Table 3: **The effectiveness of the instance-level and task-level curriculum.**

| **Base Model** | **Dataset Varient** | **PrimeVul**<br>**Accuracy ↑** | **F1 Score ↑** | **VP-S ↑** |
|---|---|---|---|---|
| Qwen2.5-7B-Coder-Instruct | w/o IC | 57.47 | 61.70 | 14.94 |
| | w/o TC | 57.13 | 60.36 | 14.25 |
| | COPO | 58.05 | 63.83 | 16.09 |
| Llama-3.1-8B-Instruct | w/o IC | 55.52 | 55.11 | 11.03 |
| | w/o TC | 52.41 | 27.62 | 4.83 |
| | COPO | 56.21 | 54.91 | 12.41 |
| StarCoder2-7B | w/o IC | 53.10 | 53.54 | 6.21 |
| | w/o TC | 51.26 | 23.74 | 2.53 |
| | COPO | 55.63 | 68.74 | 11.26 |

### 5.5 Curriculum Strategy

To evaluate the efficacy of instance- and task-level curriculum strategies, we conducted two variants. The "*w/o IC*" variant uses the equal selection criteria for all samples (i.e., $p_d = 0.5$) without considering the vulnerability types. The "*w/o TC*" variant only uses one task for preference optimization.

As shown in Table 3, the results indicate that the variants do not achieve the same performance as the ReVD. This suggests that the instance- and task-level curriculum strategy enhances the performance of models. Additionally, we observe that the impact of the curriculum strategy on Qwen2.5 is minimal, which may be attributed to the inherently superior capabilities of the base model.

## 6 Conclusion

This paper introduces a novel framework ReVD to boost the vulnerability detection of LLMs through vulnerability-specific preference optimization with reasoning-centric data synthesis. Specifically, we propose the first fully automated pipeline to synthesize vulnerability reasoning data. Then, we develop a two-stage framework ReVD on our synthetic reasoning data, which turns open-source LLMs into strong vulnerability detectors by aligning their coding knowledge with vulnerability semantics, patterns, and types. Experimental results underscore the effectiveness of ReVD for vulnerability detection compared with the state-of-the-art approaches and shed novel insights on training more generalizable and versatile LLM experts for vulnerability detection in real-world scenarios.

## 7 Limitations

Due to computing resource constraints, ReVD only uses LLMs with sizes of 7B and 8B. This limitation could slightly bias the final performance of the ReVD. Additionally, since all vulnerability reasoning samples are less than 4,096 tokens, ReVD may struggle to assess vulnerabilities that exceed this length. It may lead to a potential limitation when evaluating longer code snippets for software vulnerability detection. Conducting experiments with a larger computing budget will be our future work.

## 8 Acknowledgment

This research is supported by National Key R&D Program of China (No. 2022YFB3103900), National Natural Science Foundation of China under project (No. 62472126, 62202462), Natural Science Foundation of Guangdong Province (Project No. 2023A1515011959), Shenzhen International Science and Technology Cooperation Project (No. GJHZ20220913143008015), Shenzhen-Hong Kong Jointly Funded Project (Category A, No. SGDX20230116091246007), Shenzhen Basic Research (General Project No. JCYJ20220531095214031), and CCF-Huawei Populus Grove Fund.

## References

2024. Common weakness enumerations.

2024. Cve.

2024. National vulnerability database.

[n.d.]. Cwe-369: Divide by zero.

David Eric Austin, Anton Korikov, Armin Toroghi, and Scott Sanner. 2024. Bayesian optimization with llm-based acquisition functions for natural language preference elicitation. In *Proceedings of the 18th ACM Conference on Recommender Systems, RecSys 2024, Bari, Italy, October 14-18, 2024*, pages 74–83. ACM.

Mohammad Gheshlaghi Azar, Zhaohan Daniel Guo, Bilal Piot, Rémi Munos, Mark Rowland, Michal Valko, and Daniele Calandriello. 2024. A general theoretical paradigm to understand learning from human preferences. In *International Conference on Artificial Intelligence and Statistics, 2-4 May 2024, Palau de Congressos, Valencia, Spain*, volume 238 of *Proceedings of Machine Learning Research*, pages 4447–4455. PMLR.

Huanqia Cai, Yijun Yang, and Winston Hu. 2025. Mm-iq: Benchmarking human-like abstraction and reasoning in multimodal models. *arXiv preprint arXiv:2502.00698*.

Huanqia Cai, Yijun Yang, and Zhifeng Li. 2024. System-2 mathematical reasoning via enriched instruction tuning. *arXiv preprint arXiv:2412.16964*.

Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1456–1468. ACM.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep learning based vulnerability detection: Are we there yet? *CoRR*, abs/2009.07235.

ChatGPT. 2022. Chatgpt. `https://chat.openai.com/`.

Justin Chih-Yao Chen, Zifeng Wang, Hamid Palangi, Rujun Han, Sayna Ebrahimi, Long T. Le, Vincent Perot, Swaroop Mishra, Mohit Bansal, Chen-Yu Lee, and Tomas Pfister. 2024. Reverse thinking makes llms stronger reasoners. *CoRR*, abs/2411.19865.

Xiao Cheng, Xu Nie, Ningke Li, Haoyu Wangand Zheng Zheng, and Yulei Sui. 2022. How about bug-triggering paths?-understanding and characterizing learning-based vulnerability detectors. IEEE.

Roland Croft, Muhammad Ali Babar, and M. Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. *CoRR*, abs/2301.05456.

CVE-2021-37683. 2021. Cve-2021-37683 detail.

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

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, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 10507–10521. Association for Computational Linguistics.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela

Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. KTO: model alignment as prospect theoretic optimization. *CoRR*, abs/2402.01306.

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, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *MSR*, pages 608–620. ACM.

Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. 2023. Chatgpt for vulnerability detection, classification, and repair: How far are we? *CoRR*, abs/2310.09810.

Leonidas Gee, Milan Gritta, Gerasimos Lampouras, and Ignacio Iacobacci. 2024. Code-optimise: Self-generated preference data for correctness and efficiency. *CoRR*, abs/2406.12502.

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), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7212–7225. Association for Computational Linguistics.

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. *arXiv preprint arXiv:2501.12948*.

Jingxuan He and Martin T. 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, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 1865–1879. ACM.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *CoRR*, abs/2308.10620.

Jerry Yao-Chieh Hu, Maojiang Su, En-Jui Kuo, Zhao Song, and Han Liu. 2024. Computational limits of low-rank adaptation (lora) for transformer-based models. *CoRR*, abs/2406.03136.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1):474–499.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian J. McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, and et al. 2024. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173.

Changhua Luo, Wei Meng, and Shuai Wang. 2024. Strengthening supply chain security with fine-grained safe patch identification. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 89:1–89:12. ACM.

Matunda Nyanchama. 2005. Enterprise vulnerability management and its role in information security management. *Inf. Secur. J. A Glob. Perspect.*, 14(3):29–56.

OpenAI. 2023. GPT-4 technical report. *CoRR*, abs/2303.08774.

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 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. 2023. Execution-based code generation using deep reinforcement learning. *Trans. Mach. Learn. Res.*, 2023.

Statista. 2024. Number of common it security vulnerabilities and exposures (cves) worldwide from 2009 to 2024 ytd.

Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T. Barr, and Wei Le. 2024. A comprehensive study of the capabilities of large language models for vulnerability detection. *CoRR*, abs/2403.17218.

Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. 2024. Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning. *CoRR*, abs/2401.16185.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971.

Shu Wang, Xinda Wang, Kun Sun, Sushil Jajodia, Haining Wang, and Qi Li. 2023. Graphspd: Graph-based security patch detection with enriched code semantics. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 2409–2426. IEEE.

Xin-Cheng Wen, Cuiyun Gao, Feng Luo, Haoyu Wang, Ge Li, and Qing Liao. 2024. LIVABLE: exploring long-tailed classification of software vulnerability types. *IEEE Trans. Software Eng.*, 50(6):1325–1339.

Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. 2023a. When less is enough: Positive and unlabeled learning model for vulnerability detection. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 345–357. IEEE.

Xin-Cheng Wen, Xinchen Wang, Cuiyun Gao, Shaohua Wang, Yang Liu, and Zhaoquan Gu. 2023b. When less is enough: Positive and unlabeled learning model for vulnerability detection. *CoRR*, abs/2308.10523.

Wenyi Xiao, Zechuan Wang, Leilei Gan, Shuai Zhao, Wanggui He, Luu Anh Tuan, Long Chen, Hao Jiang, Zhou Zhao, and Fei Wu. 2024. A comprehensive survey of datasets, theories, variants, and applications in direct preference optimization. *CoRR*, abs/2410.15595.

Haoran Xu, Amr Sharaf, Yunmo Chen, Weiting Tan, Lingfeng Shen, Benjamin Van Durme, Kenton Murray, and Young Jin Kim. 2024. Contrastive preference optimization: Pushing the boundaries of LLM performance in machine translation. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

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

Kechi Zhang, Ge Li, Yihong Dong, Jingjing Xu, Jun Zhang, Jing Su, Yongfei Liu, and Zhi Jin. 2024. Codedpo: Aligning code models with self generated and verified source code. *CoRR*, abs/2410.05605.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023b. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 769–787. Association for Computational Linguistics.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *CoRR*, abs/2303.17568.

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. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand. Association for Computational Linguistics.

Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large language model for vulnerability detection and repair: Literature review and the road ahead. *CoRR*, abs/2404.02525.

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 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*, pages 10197–10207.

## A Baselines

**Pretrained Code Models** We select three prominent pre-trained models: CodeBERT (Feng et al., 2020), UniXcoder (Guo et al., 2022) and LineVul (Fu and Tantithamthavorn, 2022). These models use source code as input and are further fine-tuned for the downstream. It is applied to vulnerability detection by fine-tuning.

**Large Language Models** We choose two larger open-source LLMs: Llama3.1-70B-Instruct (Dubey et al., 2024) and Qwen2.5-32B-Coder-Instruct (Hui et al., 2024) for their proficiency in text and code generation, respectively. Additionally, we also incorporate the closed-source LLMs: GPT-4 (OpenAI, 2023) for vulnerability detection, given their robust capabilities in handling code-related tasks.

## B Vulnerability Types Distribution in PrimeVul

In this section, we detail the vulnerability types distribution in PrimeVul (Ding et al., 2024). They are provided by CWE (CWE, 2024) and GraphSPD (Wang et al., 2023) for COPO in Figure 5.

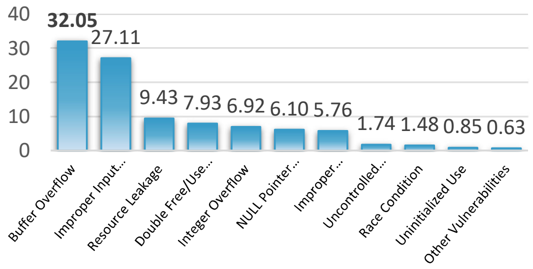


Figure 5: **Primevul (Ding et al., 2024)'s distribution of vulnerability types.** These vulnerability types are provided by CWE (CWE, 2024) and GraphSPD (Wang et al., 2023) for COPO.

## C Training Details

In this section, we detail the training details and hyperparameters employed for training Triplet SFT (T-SFT) and curriculum online preference optimization (COPO), as outlined in Table 4. These hyperparameters are primarily adapted for the base model, specifically including Qwen2.5-Coder-7B-Instruct (Hui et al., 2024), Llama-3.1-8B-Instruct (Dubey et al., 2024), and StarCoder2-7B (Lozhkov et al., 2024). We all use Qwen2.5-32B-Coder-Instruct (Hui et al., 2024) as the data

Table 4: Hyperparameters of ReVD

| **Hyperparameter** | **Value** |
|---|---|
| **Triplet SFT** | |
| Max text length | 2048 |
| Fine-tuning types | Full |
| Fine-tuning epochs | 3 |
| Warmup ratio | 0.1 |
| Learning rate | $10^{-5}$ |
| Batch size | 8 |
| **COPO** | |
| Finetuning types | Lora |
| Pref $\beta$ | 0.1 |
| Pref Loss | IPO (Azar et al., 2024) |
| Round | 3 |
| Learning rate | $5 \times 10^{-6}$ |
| Warmup steps | 300 |
| Batch size | 8 |
| Training epochs per round | 1 |

generation model $\tau$ for vulnerability detection interpretation and preference data construction.

Our implementation relies extensively on the tools and protocols provided by LLaMA-Factory (Zheng et al., 2024). All processes adhere to the standard procedures established by LLaMA-Factory, ensuring consistency and reliability in our training approach.

For inference, to ensure that the experiment is replicable, we employ greedy decoding with `temperature` = 0 to ensure that results can be reproduced. All experiments are conducted using 8 A100 (40 GB) GPUs.

## D Complete Experiment Results in Token Length

In this section, as shown in Table 6, we show the complete experiment results in different token lengths of ReVD involved with Qwen2.5-Coder-7B-Instruct, Llama-3.1-8B-Instruct and StarCoder2-7B, respectively.

## E Prompts for Vulnerability Detection

In this section, we describe prompts of LLMs to detect vulnerabilities. We utilize the prompting technique developed by Ding et al (Ding et al., 2024). For practical implementation, we configure the system to accept a maximum input length of 8192 tokens and allow a maximum of 2048 tokens to be generated for each code snippet.

The prompt of LLMs to detect vulnerabilities.

```
[System Prompt]:
You are a security expert that is good at static program
analysis.
[User Prompt]:
”””
Please analyze the following code:
“‘
func
“‘
Please indicate your analysis result with one of the
options:
(1) YES: A security vulnerability detected.
(2) NO: No security vulnerability.
Make sure to include one of the options above ”explicitly”
(EXPLICITLY!!!) in your response.
Let’s think step-by-step.
”””
```

## F Complete Answer in Figure 1

In this section, we present a comprehensive analysis of the responses provided by GPT-4 and ReVD for the example (CVE-2021-37683, 2021) illustrated in Figure 1. Our findings indicate that GPT-4 fails to accurately distinguish between the pre-code, which is vulnerable, and the post-code, which is non-vulnerable. Conversely, ReVD not only accurately identifies the pre-code as vulnerable and the post-code as non-vulnerable but also specifies the type of vulnerability, the mechanisms triggering the vulnerability, and the underlying causes. This detailed assessment underscores the superior diagnostic capabilities of ReVD in identifying and analyzing software vulnerabilities.

## G Manual Analysis

We conducted a careful examination of our generative reasoning data. Due to time constraints and heavy human costs, we randomly selected 20 examples for manual vulnerability analysis. This analysis was independently performed by two developers, each with over five years of experience in the field of vulnerability security. If two developers have differing opinions on the same reasoning answer, the two experts will discuss with each other to reach a consensus. Finally, our dataset exhibits higher consistency than those derived from the NVD, highlighting the robustness of our data generation process (95% accuracy). We also provided a detailed case included in the appendix, where the BVD accurately identifies specific types of vulnerabilities, providing insights into the program’s data flow and control flow, and specifying the triggers involved. More examples will be updated in our open-source repository.

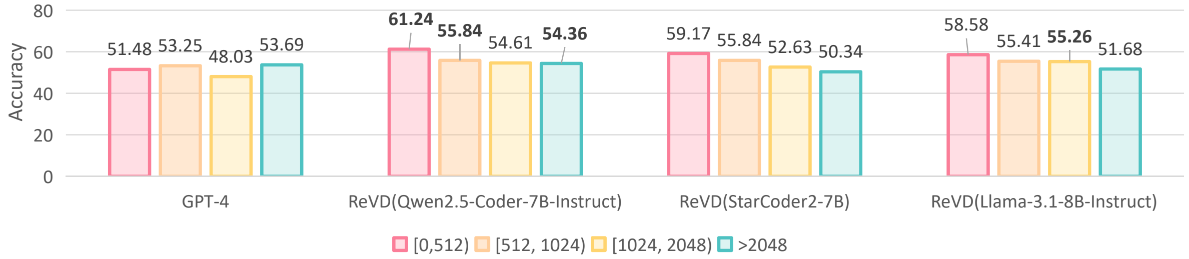


Figure 6: **The accuracy of ReVD under varying token length.** The pink, orange, yellow, and green denote the ReVD performance for token numbers containing less than 512 tokens, between 512 and 1024 tokens, between 1024 and 2048 tokens, and more than 2048 tokens.

The complete answer of pre-code and post-code by GPT-4 in Figure 1.

```
For Pre-code Answer:
The provided code is a TensorFlow Lite function for evaluating a division operation. It takes two input tensors,
performs a division operation, and stores the result in an output tensor. The function supports different data types
for the output tensor, including floating point, integer, and quantized unsigned integer.

The function begins by obtaining the parameters and data for the operation from the node's built-in and user data.
It then retrieves the input and output tensors. If the output tensor's type is either floating point or integer, it
calls the 'EvalDiv' function. If the output tensor's type is a quantized unsigned integer, it calls the 'EvalQuantized'
function. If the output tensor's type is not supported, it reports an error and returns a failure status.

From a security perspective, the function seems to be safe. It checks the return status of all function calls that
could potentially fail, and it handles unsupported output tensor types gracefully. It does not perform any unsafe
memory operations, and it does not seem to be susceptible to common security vulnerabilities such as buffer overflows,
integer overflows, or use-after-free vulnerabilities.

Therefore, my analysis result is:

(2) NO: No security vulnerability.

For Post-code Answer:
The provided code is a part of TensorFlow Lite, which is a set of tools to help developers run TensorFlow models on
mobile, embedded, and IoT devices. It enables on-device machine learning inference with low latency and a small binary
size.

The function 'Eval' is used to evaluate a division operation on two tensors. It checks the type of the output tensor and
performs the division operation accordingly. For 'kTfLiteFloat32' type, it directly performs the division operation.
For 'kTfLiteInt32' and 'kTfLiteUInt8' types, it first checks if the divisor is non-zero to avoid division by zero error.
If the output type is not one of these three, it reports an error.

From a security perspective, the code seems to be safe. It checks for division by zero for integer types, which could
otherwise lead to undefined behavior. It also checks the validity of the input and output tensors before performing
operations on them. If any of these checks fail, it returns an error status.

However, the code assumes that the 'node' pointer passed to the function is not null and that the 'builtin_data' and
'user_data' fields of the 'node' structure are valid pointers. If this is not the case, the code could lead to a
segmentation fault. But this is more of a robustness issue than a security vulnerability, as exploiting it would require
control over the internal state of the program, which is not typically possible for an external attacker.

Therefore, my analysis result is:

(2) NO: No security vulnerability.

However, it's recommended to add null pointer checks for 'node', 'node->builtin_data', and 'node->user_data' to improve
the robustness of the code.
```

The complete answer of pre-code and post-code by ReVD in Figure 1.

**For Pre-code Answer:**
YES: A security vulnerability detected.
Reason:
[Interpretation]:

1. The vulnerability in target code arises in the lack of a check for empty input tensors before the division operation is performed. Specifically, line 16 of the target code does not check if the input tensors are empty before proceeding with the division.
2. Specifically, line 16 of the target code contains the ‘EvalDiv<kernel_type>‘ function call, which is executed if the output tensor type is either ‘kTfLiteFloat32‘ or ‘kTfLiteInt32‘. The control flow includes checking the type of the output tensor and then calling the appropriate evaluation function. The data flow includes passing the context, node, parameters, data, and input tensors to the evaluation function.
3. First, the vulnerability in the division operation arises from the lack of a check for empty input tensors before the division is performed. Then, in combination with the control flow that directly proceeds to the division evaluation without any checks and the data flow that passes the input tensors directly to the division function, this can lead to a division by zero error if the input tensors are empty. Finally, this target code highlights the importance of validating input data sizes before performing operations that assume non-empty input data.

**For Post-code Answer:**
NO: No security vulnerability.
Reason:
[Interpretation]:

1. The fixed code in target code arises in lines 23-25 and 31-33.
2. Specifically, line 23 contains the macro ‘TF_LITE_CHECK_DIV_NON_ZERO(data_type)‘ which is used to check if the division operation is supported for a given data type. The control flow contains a check for the output tensor type and then a conditional check for the input tensor type. The data flow contains the input tensors being validated before they are used in the division operation.
3. First, the fixed reason in line 23 arises from the removal of the division by zero check for the ‘kTfLiteFloat32‘ output tensor type. Then, in combination with the control flow in lines 23-25 and the data flow in lines 31-33, the vulnerability is mitigated because the check for division by zero is now only performed for the ‘kTfLiteInt32‘ and ‘kTfLiteUInt8‘ output tensor types, and even then, it is only a debug assertion for ‘kTfLiteUInt8‘. Finally, this target code highlights that the vulnerability does not trigger because the necessary checks are in place to prevent division by zero for the supported tensor types.