SeCuRepair: Semantics-Aligned, Curriculum-Driven, and Reasoning-Enhanced Vulnerability Repair Framework

Chengran Yang[1] **Ting Zhang**[2] **Jinfeng Jiang**[1] **Xin Zhou**[1] **Haoye Tian**[3] **Mingzhe Du**[4]
Jieke Shi[1] **Junkai Chen**[1] **Yikun Li**[1] **Eng Lieh Ouh**[1] **Lwin Khin Shar**[1] **David Lo**[1]

[1]Singapore Management University, Singapore [2]Monash University, Australia
[3]Aalto University, Finland [4]National University of Singapore, Singapore

Abstract

The rapid accumulation of software vulnerabilities has outpaced manual remediation, creating an urgent need for Automated Vulnerability Repair (AVR). However, existing methods suffer from syntactic overfitting, mimicking surface forms without understanding the underlying repair logic, and fail to generalize to complex fixes. To transcend these limitations, we propose SeCuRepair, a reliable, scalable, and efficient RL-based AVR framework. By introducing a semantic-aware reward, SeCuRepair optimizes for code semantic equivalence rather than lexical mimicry. Furthermore, SeCuRepair incorporates an expert-aligned reasoning mechanism that explicitly grounds patch generation in a structured diagnosis. Finally, SeCuRepair introduces a difficulty-based curriculum that progressively disentangles the optimization barriers of entangled multi-hunk repairs. Extensive evaluations on a rigorous repository-level split show that SeCuRepair substantially outperforms state-of-the-art baselines, as confirmed by both automatic evaluation and human study.

1 Introduction

The rapid accumulation of software vulnerabilities has overwhelmed manual remediation capacities. The National Vulnerability Database (NVD) published 49,230 Common Vulnerabilities and Exposures (CVE) records in 2025 alone, yet due to slow human remediation, an overall total of 27,900 CVEs remain unanalyzed (nvd). With manual fixes often requiring weeks (Forsgren et al., 2021), there is a critical need for effective Automatic Vulnerability Repair (AVR).

To address this backlog, existing approaches rely on Supervised Fine-Tuning (SFT) (Chen et al., 2022; Zhou et al., 2024; Dong et al., 2025), training models to map vulnerable code to a specific human-written patch. However, this formulation is fundamentally misaligned with the intrinsic flexibility of AVR: real-world vulnerability admits diverse semantically equivalent fixes, while SFT optimizes for the exact reconstruction of a unique syntactic solution. This rigid objective penalizes generating semantically equivalent but syntactically different patches, forcing models to overfit to superficial coding patterns.

Reinforcement learning (RL) transcends SFT's rigidity by shifting the optimization objective from token-level imitation to outcome-driven exploration (Shao et al., 2024). However, applying RL to AVR is impeded by two fundamental challenges: *the lack of a reliable reward signal and the complexity of the exploration landscape.*

First, constructing a reliable reward signal for AVR is difficult. While execution-based rewards are standard in Code LLMs (Guo et al., 2025; Hui et al., 2024), real-world AVR datasets lack the large-scale build environments and test cases required for dynamic verification. Consequently, execution-free alternatives prove inadequate in AVR: 1) Lexical Rewards (e.g., SWE-RL (Wei et al., 2025)) rely on string similarity to the reference, which effectively reintroduces SFT's rigidity by penalizing valid but syntactically different patches; 2) LLM-as-Judge is unreliable in AVR, often failing to distinguish between vulnerable and patched functions (Ding et al., 2024), thereby providing noisy and indiscriminative signals. This leaves a critical gap for a reward design that is execution-free yet rewards semantic equivalence.

Second, efficient exploration is hindered by the inherent complexity of vulnerability repair. Direct RL on AVR often yields sub-optimal policies due to two critical characteristics: 1) *Reasoning-dependent Nature*: effective repair is an inherent chain-of-thought process, requiring the inference of underlying vulnerability logic and analyzing of the root cause before generating a patch. Bypassing this reasoning process leads to a reason-

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 32108–32123

ing gap, rendering the exploration phase unstable. 2) *Entangled Learning Objectives*: complex repairs spanning multiple code hunks (i.e., non-contiguous edited regions in a function) require the model to simultaneously master precise local edits (for security) and global structural synchronization (for syntactic validity). Attempting to optimize these entangled objectives concurrently overwhelms the exploration process, often trapping the LLM exploration in local optima.

To bridge these gaps, we propose SeCuRepair, which enables reliable, scalable, and efficient RL-based vulnerability repair. To bridge the gap of reliable reward, we propose a **multi-grained reward**. We proxy patch quality through a composite metric of lexical (BLEU), syntactic (AST), and semantic (Data and Control Flow Graph) similarity to human reference. This formulation anchors the reward in semantic invariants, rewarding semantically equivalent but syntactically diverse patches. Meanwhile, it enables scalable training across large-scale, real-world AVR datasets where build environments are missing.

For efficient exploration, we propose a **reasoning-enhanced initialization** followed by a **curriculum training strategy**. Specifically, we employ an expert-aligned reasoning protocol to initialize the policy, ensuring LLM executes a diagnostic process before attempting to fix. Meanwhile, we organize training samples by complexity, utilizing the number of code hunks as a proxy. This curriculum guides the LLM to first master isolated local fixes by starting with single-hunk samples, before progressively advancing to tackle global structural coordination in complex scenarios.

We evaluate the efficacy of SeCuRepair on established benchmark BigVul (Fan et al., 2020) and our newly proposed PrimeVul$_{\text{AVR}}$, which employing a strict repository-level splitting strategy to avoid data leakage. SeCuRepair demonstrates significant superiority over state-of-the-art baselines, achieving a 37.21% improvement in CodeBLEU on BigVul and 33.58% on PrimeVul$_{\text{AVR}}$. Human evaluation further confirms SeCuRepair performs on par with GPT-4o.

In summary, our main contributions are as follows: 1) We propose SeCuRepair, an RL-based framework that transcends SFT rigidity via a semantic-aware reward. 2) We devise a curriculum training strategy combined with reasoning-enhanced distillation for efficient exploration. 3) We establish a rigorous cross-repository evaluation protocol to eliminate the data leakage inherent in random splits, and introduce a new AVR benchmark PrimeVul$_{\text{AVR}}$.

2 Related Work

AVR approaches. Rule-based methods, ranging from template-guided (Lin et al., 2007) to constraint-based solvers (Zhang et al., 2022; Gao et al., 2021). They rely on either rigid, manually defined patterns or expensive runtime execution, making them ineffective for the majority of vulnerabilities where build environments are unavailable. Leading learning-based approaches such as VulMaster (Zhou et al., 2024) and FAVOR (Dong et al., 2025) improve repair performance by incorporating external knowledge and structural context based on the SFT objective. Yet, their reliance on SFT inherently encourages lexical mimicry over semantic understanding.

RL in software engineering. Reinforcement Learning has shown promise in various software engineering tasks (Guo et al., 2025; Hui et al., 2024; Wei et al., 2025; Yang et al., 2024), typically relying on execution feedback as reward. However, this is impractical for AVR due to the scarcity of large-scale build environments. Existing execution-free alternatives also fall short. Lexical rewards (e.g., SWE-RL (Wei et al., 2025)) measure string similarity to a reference, but this rigid metric penalizes syntactically diverse yet semantically equivalent solutions, providing a biased signal. Model-based rewards (Dutta et al., 2024; Sghaier et al., 2025) employ LLMs as judges, but these prove unreliable for security tasks, as even SOTA models struggle to distinguish between vulnerable code and its patched version (Ding et al., 2024), which is the core of AVR rewarding. In contrast, our approach bridges this gap by introducing a multi-grained, semantic-aware reward, proxying patch quality through static analysis of syntactic and semantic equivalence.

3 Problem Statement and Motivation

In this section, we formulate the task of AVR and analyze the limitations of existing AVR approaches: 1) Syntactic Overfitting; 2) Suboptimization on Complex Repair; and 3) Biased Evaluation Setting.

3.1 Problem Definition

Following prior works (Chen et al., 2022; Zhou et al., 2024; Dong et al., 2025), we formulate AVR as a function-level sequence-to-sequence genera-

```
print_prefix(netdissect_options *ndo, const u_char *prefix, ...) {
    int plenbytes;
    char buf[64];   // [!] buffer is uninitialized
    if (prefix[0] >= 96 && is_ipv4_mapped_address(&prefix[1])) {
        // IPv4-mapped branch (details omitted)
    } else {
        plenbytes = decode_prefix6(ndo, prefix, buf, ...);
        // True FIX
        if (plenbytes < 0) return plenbytes;
    }
    ND_PRINT((ndo, "%s", buf));
    return plenbytes;
}
```

```
print_prefix(netdissect_options *ndo, const u_char *prefix, ...) {
    int nbytes;
    char buf[64];   // [!] buffer is uninitialized
    if (prefix[0] >= 96 && is_ipv4_mapped_address(&prefix[1])) {
        // IPv4-mapped branch (details omitted)
    } else {
        nbytes = decode_prefix6(ndo, prefix, buf, ...);
        // Hallucinated FIX
        ND_PRINT(*ndo);
    }
    ND_PRINT((ndo, "%s", buf));
    return nbytes;
}
```

Figure 1: Syntactic overfitting of SFT: The SFT model generates a correct fix for the buffer over-read guard with the original input, but fails to generalize when a local variable is simply renamed from PLENBYTES to NBYTES.

tion task. Given a vulnerable function X and auxiliary information L (e.g., information of vulnerability localization and Common Weakness Enumeration), the goal is to generate a patched function Y that fixes the vulnerability in X.

3.2 Syntactic Overfitting of SFT in AVR

To probe whether existing SFT approaches (Wang et al., 2021; Zhou et al., 2024; Dong et al., 2025) learn semantic repair patterns or overfit to syntactic cues, we follow (Rahman et al., 2024) and test their performance under a simple refactoring: renaming local variables of input while preserving code semantics (see Appendix A for details). We observe that all SFT models are sensitive to these minor changes, failing to generate valid patches in 40–70% of cases. Figure 1 provides an example of this failure. A simple variable renaming (`plenbytes` $\rightarrow$ `nbytes`) causes VulMaster to regress from a correct fix to an irrelevant hallucination. This provides empirical evidence that SFT models heavily rely on superficial variable correlations. In this paper, we propose an RL training framework that rewards semantically equivalent patches to mitigate syntactic overfitting.

3.3 Suboptimization on Multi-hunk Repair

A critical oversight in current AVR research is the lack of optimization for complex, multi-hunk repair. Real-world fixes often require changes across multiple discontinuous code regions, yet existing methods treat all samples uniformly. To quantify this impact, we stratified the performance of representative baselines, VulMaster (Zhou et al., 2024) and FAVOR (Dong et al., 2025), by the number of repair hunks in human reference. Performance drops with complexity: even moving from 1 to 2 hunks causes 11% to 20% CodeBLEU drops across all approaches (see Figure 6 for full results).

Our manual inspection reveals that 55% of multi-hunk failures stem from global inconsistencies. We attribute this failure to entangled learning objectives: the model faces the dual burden of executing precise local security edits while simultaneously maintaining global syntactic synchronization. We propose a curriculum-training strategy to mitigate this issue, which guides the LLM to first master local fixes by starting with single-hunk samples. This setting smooths the optimization landscape, allowing the model to learn patch patterns without global interference (Bengio et al., 2009). We then introduce multi-hunk samples, allowing LLM to focus more on aligning global dependencies.

3.4 Data Leakage in Conventional Evaluation

Existing AVR approaches typically rely on function-level random splitting (Zhou et al., 2024; Dong et al., 2025). However, this setting poses a significant data leakage risk (Zhang et al., 2025), as it allows models to exploit project-specific patterns or "peek" at future fixes within the same repository. We quantify this threat by comparing model performance under the random-split versus a strict cross-repository protocol, where all functions from a specific project are confined to a single split (train/-val/test). The results reveal a severe generalization gap. Under the strict repository-level split, performance degrades sharply across all baselines: CodeBLEU scores drop by a relative 21.49%–29.70%, while exact match scores plummet by 87.88% to 89.86% (see Appendix C for full details). These findings highlight the leakage issue in a random-split setting, inspiring us to use a cross-repository setting in the following sections.

4 Approach

Driven by the above limitations, we propose SeCuRepair, an RL-based AVR framework to enforce LLM to learn the underlying logic of vulnerability repair. The design of SeCuRepair is guided by three core principles:

1) Reasoning-enhanced Initialization: Vulnerability repair inherently demands complex logic:

root cause analysis, vulnerability type classification, and devising a fix strategy. To mitigate the optimization instability caused by this reasoning gap, we warm-start the model via knowledge distillation, leveraging a structured protocol that simulates a security-expert cognitive workflow.

2) Semantic-aware Optimization: To overcome the syntactic overfitting of SFT, we propose a multi-grained static reward signal for patch quality, which enables the measurement of semantic equivalence.

3) Difficulty-based Curriculum Learning : To dismantle the optimization barrier in complex multi-hunk repairs, we employ a difficulty-based curriculum. This approach decouples the learning process by progressively increasing task complexity, allowing the model to master local security edits before tackling global synchronization.

These principles are realized through a two-stage training pipeline, as shown in Figure 2. In the first stage, we bootstrap the model's reasoning ability. We employ a commercial teacher LLM to generate high-quality (reasoning, patch) examples with heuristic-based rejection sampling, and then perform SFT on the student model. In the second stage, we refine the model's patching ability using GRPO algorithm (Shao et al., 2024). The model is optimized through a semantic-aware RL optimization, which uses rewards measuring lexical, syntactic, and semantic correctness, and is guided by difficulty-based curriculum training, which gradually increases task complexity to multi-hunk fixes.

4.1 Reasoning-enhanced Initialization

The first stage of our pipeline aims to initialize the model to follow a reason-then-edit paradigm through knowledge distillation.

Knowledge Distillation. To achieve this, the first step is to construct a dataset of (reasoning, patch) pairs by distilling reasoning knowledge from a teacher LLM (i.e., GPT-5-mini). To fully exploit the reasoning capabilities of teacher LLMs, we define a structured reasoning protocol that mirrors the cognitive workflow of human security experts: (1) Vulnerability Classification: characterize the CWE type and security impact; (2) Root Cause Analysis: trace the logical flaw triggering the vulnerability; and (3) Repair Planning: formulate a high-level repairing strategy before patching. Extracted reasoning chains that adhere to this protocol effectively transfer the teacher's security diagnostic capabilities to the student model.

Rejection Sampling. While the expert workflow guides the reasoning process, it does not guarantee correctness. LLMs often produce plausible-sounding but functionally incorrect patches (Dong et al., 2025). Therefore, we adopt a two-step rejection-sampling strategy to denoise reasoning data: 1) we apply syntactic filtering to discard responses that violate the output schema; 2) we apply semantic filtering to retain data only if its patch is similar to the human-reference patch using CodeBLEU as a heuristic proxy.

SFT. After rejection sampling, we fine-tune the student model with a standard SFT objective. The resulting model is capable of generating reasoning before patching and serves as the initial policy for the subsequent reinforcement learning stage.

4.2 Semantic-aware RL

Following SFT stage, SeCuRepair employs RL with semantic-aware rewards, enabling the model to explore the solution space and reward semantic equivalent solutions over rigid textual identity.

We employ an on-policy RL algorithm GRPO (Shao et al., 2024) for this stage. During training, the model generates a (reasoning, patch) pair for a given vulnerability. We then compute a scalar reward Re based on the generated patch against its human reference. This semantics-aware reward moves beyond token overlap to measure the lexical (BLEU), syntactic (Abstract Syntax Tree, AST), and semantic (DFG and CFG) similarities. It is then used to update the model's policy, encouraging it to generate patches that achieve higher semantic agreement with human reference.

4.2.1 Reward Design

Our reward assesses the agreement between generated patch and the reference at three granularities:

Lexical Agreement. We calculate BLEU score between the generated and reference patches, which measures the proportion of overlapping n-grams. It introduces a token-level reward signal that incentivizes the use of correct keywords and identifiers, thereby suppressing token-level hallucinations. BLEU score for a generated patch $\hat{y}$ against an oracle patch y is defined as: $\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} \frac{\log p_n}{N}\right)$, where p_n is the n-gram precision, and BP is the brevity penalty to discourage excessively short generations.

Syntactic Agreement. AST similarity evaluates the structural alignment between the generated patch and the oracle. Each code snippet is de-

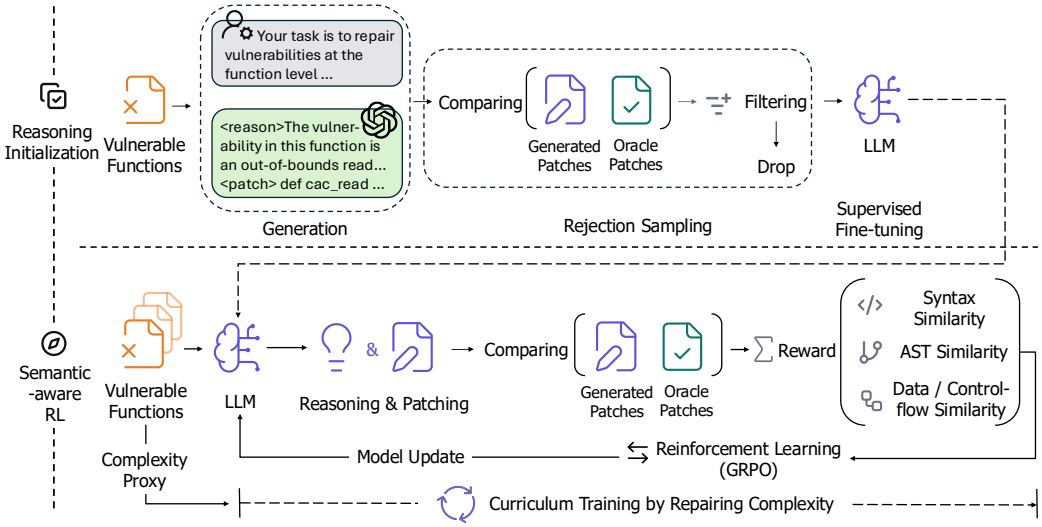

Figure 2: Overview of the SeCuRepair training pipeline. (1) Stage 1: Reasoning-enhanced Initialization. A teacher LLM generates (reasoning, patch) candidates, which are filtered via rejection sampling to create a high-quality dataset. The student model is then fine-tuned to learn the reason-then-edit format. (2) Stage 2: Semantic-guided RL. The SFT model is refined using RL. The policy is rewarded based on a composite score of lexical (BLEU), syntactic (AST), and sematic (CFG & DFG) similarity to the oracle. Stage 2 is organized with a curriculum, progressively training the model on repairs with an increasing number of hunks to master multi-hunk fixes.

composed into a set of subtrees, and similarity is computed as the proportion of matched subtrees. Formally, given $S(\hat{y})$ and $S(y)$ as the sets of subtrees from a generated patch $\hat{y}$ and an oracle patch y respectively, we define AST similarity as: $\text{Sim}_{\text{AST}}(\hat{y}, y) = \frac{|S(\hat{y}) \cap S(y)|}{|S(y)|}$.

Semantic Agreement. We measure semantic consistency from complementary perspectives: CFG (control-flow graph) and DFG (data-flow graph) similarity. The DFG represents how values propagate through a program. Each code snippet is decomposed into a set of data-dependency tuples (v_i, v_j). We measure similarity as the fraction of matched edges: $\text{Sim}_{\text{DFG}}(\hat{y}, y) = \frac{|E(\hat{y}) \cap E(y)|}{|E(y)|}$, where $E(\hat{y})$ and $E(y)$ denote the sets of data-flow edges extracted from $\hat{y}$ and y, respectively.

Complementarily, the CFG captures the algorithmic structure and execution logic. We extract CFGs using the Joern parser, where nodes represent statements and edges denote control transfers. However, computing precise structural similarity metrics, such as the Graph Edit Distance (GED) with structural matching (i.e., ignoring variable naming differences), is NP-hard. This computational complexity makes GED inappropriate for online RL, which requires instant reward feedback. To address this efficiency bottleneck, we employ the Weisfeiler-Lehman (WL) Graph Kernel (Shervashidze et al., 2011) as a fast, polynomial-time approximation. The WL kernel maps each graph to a high-dimensional feature vector $\phi(G)$ by iteratively aggregating neighborhood labels to capture topological sub-structures (e.g., chains, cycles). We define the CFG similarity as the normalized kernel value: $\text{Sim}_{\text{CFG}}(\hat{y}, y) = \frac{k_{\text{WL}}(\hat{y},y)}{\sqrt{k_{\text{WL}}(\hat{y},\hat{y}) \cdot k_{\text{WL}}(y,y)}}$.

Final Reward. We first define a syntactic check to ensure patches are structurally valid and can be parsed into AST via Tree-sitter; failed patches receive a zero reward. We then define a reward vector $\mathbf{r} = \langle \text{BLEU}, \text{Sim}_{\text{AST}}, \text{Sim}_{\text{DFG}}, \text{Sim}_{\text{CFG}} \rangle$ for valid fix, where the components represent these agreement scores. We normalize these scores into a single reward ranging from [0, 1] by taking their mean: $Re(\hat{y}, y) = \frac{\|\mathbf{r}\|_1}{n}$, where $\|\mathbf{r}\|_1$ is the L1-norm of the vector, representing the sum of the absolute values of its components. This balanced reward function encourages the model to generate patches that are lexically faithful, syntactically consistent, and semantically coherent with the oracle.

4.3 GRPO Optimization

In the RL stage, we optimize the policy model using Group Relative Policy Optimization (GRPO) (Shao et al., 2024) with the reward signal $Re(\hat{y}, y)$. Let $r_{ij}(\theta) = \frac{\pi_\theta(\hat{y}_{ij} \mid P_i)}{\pi_{\theta_{\text{old}}}(\hat{y}_{ij} \mid P_i)}$ be the importance weight, which measures how much more likely the new policy is to generate $\hat{y}_{ij}$ compared

to the old one. The GRPO surrogate loss is:

$$\mathcal{L}_{\text{GRPO}}(\theta) = \mathbb{E}_{i,j}\Big[\min\left(r_{ij}(\theta)\,A_{ij},\ \text{clip}(\epsilon_c)\,A_{ij}\right)\Big] - \beta\,\text{KL}[\pi_{\theta_{\text{old}}} \,\|\, \pi_\theta]$$

4.4 Curriculum Learning

Our analysis in Section 3.3 confirms that coordinating edits across multiple code hunks is a primary challenge for AVR models. To address this, we adopt a curriculum learning strategy that organizes training from simple to complex repairs. This approach decouples the learning process by progressively increasing task complexity, allowing the model to master local security edits before tackling global syntactic synchronization.

We use the number of code hunks in human reference as a proxy for repair difficulty. We define three stages: easy (1-2 hunks), medium (3-5 hunks), and hard (>5 hunks). Preliminary experiments showed that training only on single-hunk fixes caused the model to overfit to overly localized repairs; therefore, our easy stage includes two-hunk functions. Our curriculum proceeds through the three stages during the RL phase. The training set is *expanded cumulatively* at each epoch to prevent catastrophic forgetting, and the model's policy is always initialized from the previous stage's checkpoint.

5 Experimental Setup

5.1 Baselines and Evaluation Metrics

We benchmark SeCuRepair against specialized AVR approaches (VulMaster (Zhou et al., 2024), FAVOR (Dong et al., 2025)) retrained on our repository-level split, a commercial SOTA model (GPT-4o (Achiam et al., 2023)) prompted with identical instructions, and our own SFT-only base model (Qwen2.5-7B) to isolate the contributions of the RL stage. More details are in Appendix F.

5.2 Datasets

To strictly prevent data leakage and evaluate cross-project generalization, we enforce a repository-level split where no repository overlaps between training, validation, and test sets. Following selected baselines, we utilize BigVul (Fan et al., 2020). In addition, we construct a new out-of-distribution test set $\text{PrimeVul}_{\text{AVR}}$ derived from the vulnerability detection dataset PrimeVul (Ding et al., 2024). We rigorously filtered out any repositories present in the BigVul training set, resulting in 1,554 unique function pairs. Detailed construction pipelines and statistics are provided in Appendix D.

5.3 Implementation Details

We implement SeCuRepair based on Qwen2.5-7B-Instruct (Bai et al., 2023) using HuggingFace (Hugging Face) and Verl (ByteDance Seed). For SFT, we perform full-parameter fine-tuning for 3 epochs with a learning rate of 3×10^{-5} and a context length of 4,096, computing loss exclusively on LLM response. For RL, we employ the GRPO algorithm initialized from the best SFT checkpoint. We set the global batch size to 1,024 with $M = 8$ rollouts per prompt and train for up to 20 epochs with an actor learning rate of 1×10^{-6}. Full details are available at Appendix E. We use CodeBLEU as our primary metric following existing works (Zhou et al., 2024; Dong et al., 2025). We exclude the Exact Match (EM) metric from our main evaluation. Under the rigorous repository-level split, all models achieve negligible scores ($< 5\%$), rendering the metric indistinguishable (more discussion of EM is in Appendix G).

6 Results

We evaluate the effectiveness of SeCuRepair in repairing vulnerabilities under both automatic and human evaluation. The automatic evaluation results are summarized in Table 1, with the human study results in Table 2. We present the ablation performance of reasoning in Table 3 and Figure 4, semantic-aware RL in Figure 3, and curriculum training in Figure 5.

1) SeCuRepair performs better than SOTA baselines. Table 1 shows that SeCuRepair outperforms all baselines on both datasets by a significant margin in terms of CodeBLEU. Specifically, SeCuRepair surpasses the best-performing baseline on BigVul, VulMaster, by 37.21%; and surpasses the best-performing baseline on $\text{PrimeVul}_{\text{AVR}}$, GPT-4o, by 33.58%. This result demonstrates the superior ability of SeCuRepair to repair vulnerabilities and generalize to unseen repositories. A Wilcoxon signed-rank test (Woolson, 2007) confirms that all of SeCuRepair's performance gains over the baselines are statistically significant ($p < 0.001$).

Notably, SeCuRepair outperforms the monolithic SFT baseline (Qwen2.5-7B+SFT) with CodeBLEU gains of 21.98% on BigVul and 20.64% on $\text{PrimeVul}_{\text{AVR}}$, confirming the superiority of our training framework over standard SFT.

2) Human evaluation validates the superior performance of SeCuRepair. To complement automatic metrics, we conducted a blind human eval-

Table 1: Comparison of SeCuRepair with state-of-the-art baselines on BigVul and PrimeVul$_{\text{AVR}}$.

Approach	Strategy	CodeBLEU (%)	
		BigVul	PrimeVul$_{\text{AVR}}$
FAVOR	SFT	25.78	11.77
VulMaster	SFT	26.33	11.62
GPT-4o	NA	25.90	23.41
SeCuRepair	SFT	29.62	25.92
SeCuRepair	SFT & RL	**36.13**	**31.27**

Table 2: Human Preference Distribution across Different AVR Systems.

Score	SeCuRepair	GPT-4o	VulMaster
1	9	28	83
2	121	127	164
3	124	104	47
4	24	31	12
5	31	19	3
AVG	**2.83**	**2.63**	**1.99**

uation comparing SeCuRepair against VulMaster and GPT-4o. We recruited four experts with experience in software security and C++ to evaluate 309 statistically sampled instances (95% confidence level). Participants rated patch semantic similarity on a 5-point Likert scale relative to the human reference (full detail is available at Appendix H). Aligning with the goal of AI-assisted repair (Takerngsaksiri et al., 2025), we categorize scores ≥ 3 as "workable drafts," indicating patches that capture sufficient logic to serve as valid starting points for developers.

The human evaluation result in Table 2 confirms the superiority of our approach with moderate Kappa agreement. SeCuRepair achieves the highest rate of workable patches (58.0%), outperforming GPT-4o (50.0%) and VulMaster (20.0%). This result validates that a specialized training pipeline is more effective for AVR than relying solely on large-scale generalist models. Qualitative analysis further reveals that SeCuRepair frequently generates patches that are syntactically distinct yet semantically identical to human references, demonstrating its capacity to learn semantic repair logic rather than token-level mimicry. (see Appendix H.2 for case study). Meanwhile, patches rated as "workable", despite minor issues like over-defensive logic or verbose syntax, successfully capture the underlying fix semantics (see Appendix H.3 for detailed definition and criteria).

3) Each technical design progressively contributes to the final performance. SeCuRepair trains LLMs in a multi-stage process. Therefore, we conduct a progressive ablation study on each design to quantify their contributions, as shown in Figure 3. Specifically, we compare the following model variants: 1) Base Model (A0). The original, pre-trained Qwen2.5-7B-Instruct without any fine-tuning. 2) SFT with Reasoning (A1). The base model was fine-tuned on distilled reasoning traces. 3) SFT $\rightarrow$ Syntactic-aware RL (A2). We start from our A1 checkpoint and then apply RL, but with a reward function based only on the BLEU score. 4) Syntactic-aware RL $\rightarrow$ Syntactic- and Semantic-aware RL (A3). We enhance the A2 checkpoint by replacing the BLEU-only reward with our full semantics-aware reward function (BLEU + AST + DFG + CFG). 5) Full SeCuRepair Model (A4). Finally, we add the curriculum learning schedule to the A3 checkpoint We confirm that each design progressively contributes to the final performance, highlighting the effectiveness of each component and the synergistic nature of SeCuRepair.

Table 3: Repair effectiveness of *patch-only* vs *reasoning+patch* supervision on BigVul.

Variant	SFT Data	CodeBLEU (%)
Qwen2.5-7B-Instruct	NA	24.49
Qwen2.5-7B-Instruct$_{SFT}$	Patch-Only	25.78
Qwen2.5-7B-Instruct$_{SFT}$	Reasoning+Patch	**27.57** ($\uparrow$ 6.94%)

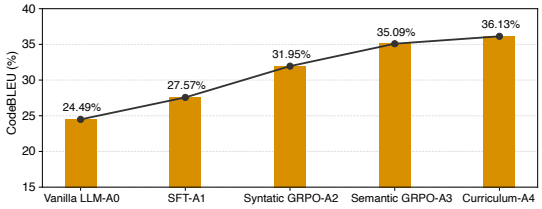

Figure 3: Step-wise ablation performance of SeCuRepair across training stages. We confirm that each technical design contributes to the final performance.

4) Reasoning-enhanced initialization is effective. We investigate whether the reasoning-enhanced SFT stage provides a better starting point for reinforcement learning. We compare two RL training runs: one initialized from our SFT checkpoint (θ_{SFT}), and another that starts directly from the base model. Figure 4 plots the learning curves for both settings. We observe SFT-initialized policy outperforms the non-SFT variant by exhibiting faster convergence (reaching peak reward 33% earlier), greater stability (avoiding mid-training volatility), and a higher final reward plateau (0.38 vs. 0.36), which directly correlates with improved downstream repair performance.

5) Semantic-aware reward improves repairing performance over lexical reward. As shown in

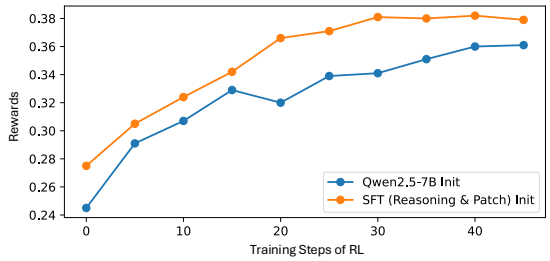

Figure 4: Training curves of the RL stage.

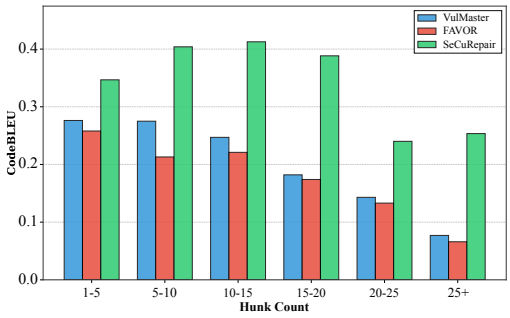

Figure 5: CodeBLEU across different hunk counts.

Figure 3, GRPO with our semantic-aware reward performs better than using a lexical reward signal that compares the string similarity to the human reference based on BLEU(from A2 to A3 checkpoint). This reward design helps the model to learn from syntactically different but semantically similar patches to the human reference, guiding the model to learn the underlying repair logic over lexical mimicry, leading to 7.67% relative improvement in CodeBLEU.

6) Difficulty-aware curriculum adds a performance boost. We observe from Figure 3 (from A3 to A4), introducing a curriculum learning schedule improves the performance of SeCuRepair by 2.97%. A per-bucket analysis reveals that the curriculum yields substantial improvements on complex fix: functions with 2-10 code hunks by 4.24% and functions with >10 vulnerable regions by 9.46% in terms of CodeBLEU. In addition, we compare the performance of SeCuRepair and baselines across different count hunk in Figure 5, we observe SeCuRepair consistently outperforms baselines in all hunk counts, highlighting the effectiveness of this curriculum training strategy.

7 Analysis

1) SFT with vs. without reasoning. We assess whether training on explicit reasoning traces improves patch quality. We compare two SFT Qwen-2.5-7B: one trained on our distilled (vulnerable code, reasoning, ground-truth patch) triples, and the other model trained on the same dataset without reasoning text, i.e., (vulnerable code, ground-truth patch). The results in Table 3 show that while standard SFT improves over the base model, incorporating reasoning provides a distinct advantage. SFT (Reasoning+Patch) outperforms SFT (Patch-Only) by a margin of 6.94% on CodeBLEU. This performance gain indicates that training on explicit reasoning helps the model learn to coordinate semantic edits, moving it beyond simple token-level imitation toward a more robust understanding of the repair task.

2) Performance on critical vulnerabilities. We further analyze performance across the MITRE Top-10 most dangerous CWEs (MITRE, 2024). As shown in Table 4, SeCuRepair consistently outperforms the SFT baseline across nearly all categories. The gains are particularly pronounced for high-impact injection vulnerabilities: SQL Injection (CWE-89) improves from 0.201 to 0.430, and XSS (CWE-79) from 0.232 to 0.303. These results confirm that our semantics-aware RL is highly effective at capturing the precise structural patterns required to mitigate complex security flaws. Full results are available at Appendix I.

3) Robustness to code perturbation. We perform an ablation study and confirm that SeCuRepair is robust to code perturbation. Following the existing code perturbation strategy (Yang et al., 2022), we replace the first local variable in the input with a randomized string to create a syntactically valid variant. Since local variable names do not dictate control flow, this modification guarantees that the code syntax remains strictly equivalent, with minimal effect on code semantics. On the perturbed PrimeVul dataset, VulMaster's CodeBLEU dropped by 14.75%, while SeCuRepair declined only 3.39%, highlighting the superior robustness of SeCuRepair to code perturbation.

8 Conclusion and Future Work

We reveal that current AVR approaches struggle with syntactic brittleness and multi-hunk complexity. In response, we propose SeCuRepair, which leverages a reason-then-edit mechanism, semantics-driven RL, and curriculum-training strategy to ground repairs in diagnostic planning and structural correctness. SeCuRepair achieves state-of-the-art performance on BigVul and our proposed $\text{PrimeVul}_{\text{AVR}}$, validating the necessity of aligning training objectives with code semantics rather than surface forms. We plan to broaden SeCuRepair to diverse programming languages in future work.

Limitations

We propose SeCuRepair, an automated vulnerability repair framework that combines expert-level reasoning with semantics-aware reinforcement learning and curriculum training. While SeCuRepair demonstrates substantial improvements over state-of-the-art baselines, we acknowledge several primary limitations.

First, reliance on static proxy metrics. Our training framework uses static semantic rewards rather than execution-based feedback. While execution feedback provides ground-truth verification, it is computationally expensive and brittle in practice. We intentionally relax this constraint to ensure scalability across large-scale, real-world datasets such as BigVul, where build environments are often incomplete. Although strictly "noisier" than dynamic testing, our results confirm that these proxies effectively guide the model toward effective repair. Meanwhile, this reward remains grounded in reference comparison. Nonetheless, we posit that this approach serves as a middle ground to mitigate syntactic overfitting. It enables SeCuRepair to recognize and reward patches that are syntactically diverse yet structurally faithful to the developer's intent, marking a crucial step toward fully reference-agnostic semantic repair.

Second, language specificity. Our current evaluation is restricted to C/C++ vulnerabilities. Although the underlying reason-then-edit paradigm and semantic-aware RL are designed to be language-agnostic, their generalizability to other ecosystems, such as Python and Java, or to multilingual settings, remains to be empirically validated.

Third, dependency on closed-source teachers. Our cold-start reasoning data is distilled from proprietary LLMs, such as GPT-5-mini. The black-box nature of these teachers poses challenges for full reproducibility and may introduce distillation bias. To mitigate these issues, we release our distilled datasets and enforce a strict security-expert-aligned workflow to filter out low-quality rationales, ensuring that the student model learns robust patterns.

Lastly, our automatic evaluation relies on the static metric CodeBLEU. To mitigate this issue, we complement it with human evaluation as well as qualitative and quantitative case studies, which together validate the effectiveness of SeCuRepair in helping developers.

Ethical Considerations

The primary goal of SeCuRepair is to enhance software security by automating the remediation of software defects. We recognize the theoretical risk that such models could be repurposed for malicious vulnerability discovery. However, given the rapid growth of software vulnerabilities outpacing manual repair capacity, we argue that advancing automated defensive tools is an ethical imperative. All data utilized in this study are publicly available, and we strictly adhere to the usage policies of the respective data sources.

Acknowledgments

This research is supported by the National Research Foundation, Singapore under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

NVD dashboard. https://nvd.nist.gov/general/nvd-dashboard. Accessed: 2025-09-07.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Tushar Aggarwal, Swayam Singh, Abhijeet Awasthi, Aditya Kanade, and Nagarajan Natarajan. 2025. Nextcoder: Robust adaptation of code lms to diverse code edits. In *Forty-second International Conference on Machine Learning*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, and 1 others. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*.

ByteDance Seed. verl: Volcano engine reinforcement learning for LLMs. https://verl.readthedocs.io/en/latest/index.html. Accessed: 2025-09-07.

Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165.

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

Qingao Dong, Yuanzhang Lin, Hailong Sun, and Xiang Gao. 2025. Enhancing automated vulnerability repair through dependency embedding and pattern store. *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 193–204.

Sujan Dutta, Sayantan Mahinder, Raviteja Anantha, and Bortik Bandyopadhyay. 2024. Applying rlaif for code generation with api-usage in lightweight llms. *arXiv preprint arXiv:2406.20060*.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th international conference on mining software repositories*, pages 508–512.

Nicole Forsgren, Bas Alberts, Kevin Backhouse, Grey Baker, Greg Cecarelli, Derek Jedamski, Scot Kelly, and Clair Sullivan. 2021. 2020 state of the Octoverse: Securing the world's software. *arXiv preprint arXiv:2110.10246*.

Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–27.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Hugging Face. Transformers: State-of-the-art machine learning for PyTorch, TensorFlow, and JAX. https://huggingface.co/docs/transformers/en/index. Accessed: 2025-09-07.

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

Kaiyao Ke. 2025. Niodebugger: A novel approach to repair non-idempotent-outcome tests with llm-based agent. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 762–762. IEEE Computer Society.

Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. 2007. Autopag: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 329–340.

Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. 2023. Explaining software bugs leveraging code structures in neural machine translation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 640–652. IEEE.

MITRE. CWE-476: NULL pointer dereference. https://cwe.mitre.org/data/definitions/476.html. Accessed: 2025-12-09.

MITRE. 2024. 2024 CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html. Accessed: 2025-09-07.

NIST National Vulnerability Database. CVE-2021-29513. https://nvd.nist.gov/vuln/detail/CVE-2021-29513. Accessed: 2025-12-09.

Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2024. Towards causal deep learning for vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 153:1–153:11. ACM.

Oussama Ben Sghaier, Rosalia Tufano, Gabriele Bavota, and Houari Sahraoui. 2025. Leveraging reward models for guiding code review comment generation. *arXiv preprint arXiv:2506.04464*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. 2011. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9).

Wannita Takerngsaksiri, Jirat Pasuksmit, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Ruixiong Zhang, Fan Jiang, Jing Li, Evan Cook, Kun Chen, and Ming Wu. 2025. Human-in-the-loop software development agents. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 342–352. IEEE.

vLLM Project. vLLM: Easy, fast, and cheap LLM serving with PagedAttention. https://docs.vllm.ai/en/latest/usage/index.html. Accessed: 2025-09-07.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*.

Martin Weyssow, Chengran Yang, Junkai Chen, Ratnadira Widyasari, Ting Zhang, Huihui Huang, Huu Hung Nguyen, Yan Naing Tun, Tan Bui, Yikun Li, and 1 others. 2025. R2vul: Learning to reason about software vulnerabilities with reinforcement learning and structured reasoning distillation. *arXiv preprint arXiv:2504.04699*.

Robert F Woolson. 2007. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials*, pages 1–3.

Chengran Yang, Hong Jin Kang, Jieke Shi, and David Lo. 2024. Acecode: A reinforcement learning framework for aligning code efficiency and correctness in code language models. *arXiv preprint arXiv:2412.17264*.

Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1482–1493.

Jian Zhang, Shangqing Liu, Xu Wang, Tianlin Li, and Yang Liu. 2023. Learning to locate and describe vulnerabilities. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 332–344. IEEE.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020a. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1385–1397.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. 2020b. Learning to handle exceptions. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 29–41.

Ting Zhang, Chengran Yang, Yindu Su, Martin Weyssow, Hung Nguyen, Tan Bui, Hong Jin Kang, Yikun Li, Eng Lieh Ouh, Lwin Khin Shar, and 1 others. 2025. Benchmarking large language models for multi-language software vulnerability detection. *arXiv preprint arXiv:2503.01449*.

Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. 2022. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 691–702.

Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

A Variable Perturbation

To probe whether AVR approaches learn real semantic repair patterns or merely overfit to lexical cues, we test the robustness of baselines (VulMaster (Zhou et al., 2024), FAVOR (Dong et al., 2025), and VulMaster's base LLM CodeT5 (Wang et al., 2021)) under a simple refactoring: renaming local variables while preserving code semantics. We randomly selected 10 cases where baseline models (trained and evaluated under random splits) correctly generated a patch, then applied consistent variable renamings. For each instance, we manually renamed locally initialized variables to contextually appropriate alternatives (e.g., swapping len for size), ensuring the code remained natural and semantically equivalent, before re-evaluating the models.

Observation. All studied models are highly sensitive to these minor changes, failing to generate functionally equivalent patches in 40–70% of cases, based on manual inspection by the first author. Figure 1 provides an example of this failure. In the original patch (a), VulMaster correctly inserts a safety guard to prevent a buffer over-read. However, after simply renaming the variable `plenbytes` to `nbytes` (b), the model fails. It does not generate the equivalent safety check but instead hallucinates an irrelevant function call, leaving the vulnerability unresolved. This case study provides evidence that the model *has not learned the underlying semantic rule*, i.e., "the return value of the decoding function must be checked before proceeding." Instead, it learns a superficial correlation between the specific variable name `plenbytes` and the patch. While in limited scale, the consistent failures strongly motivates the need for an AVR learning framework that prioritizes syntactic and semantic understanding of repair patterns over simple lexical mimicry.

B Performance Degradation on Multi-Hunk Repairs

Setup. Real-world vulnerability fixes frequently span multiple, non-contiguous regions (*hunks*) that must be edited consistently. Prior work FAVOR (Dong et al., 2025)'s case study notes that multi-hunk repair significantly increases AVR complexity and is a common failure mode for sequence-to-sequence models. To quantify this, we evaluate baseline models on our repository-split test set, stratifying the results by the number of hunks in each function.

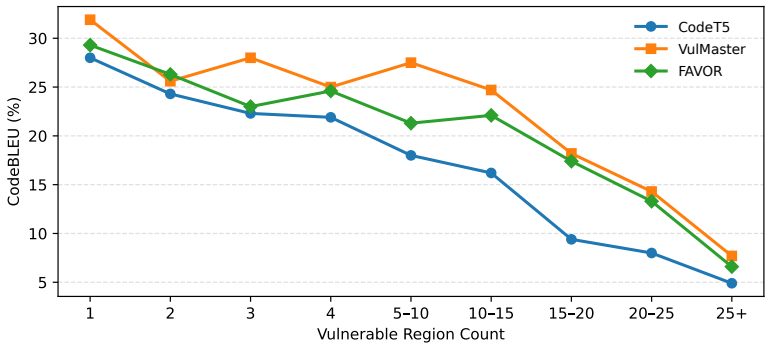

Figure 6: The performance of existing AVR approaches drops as the number of hunks increases.

Observation. Figure 6 shows that performance declines generally as the number of hunks increases. When moving from 1 to 2 hunks, VulMaster, FAVOR, and CodeT5 performance drops by 19.75%, 11.27%, and 13.22% in terms of CodeBLEU, respectively. This downward trend continues for repairs requiring more than two hunks. This observation calls for an AVR approach that can work better in terms of multi-hunk repair performance.

C Cross-Repository Generalization Gap

Setup. AVR models must generalize beyond the repositories seen in training. We formalize this through a cross-repository evaluation protocol. We evaluate the baselines on the common dataset used by VulMaster and FAVOR's dataset: BigVul dataset (Fan et al., 2020). We compare their performance under two distinct data splitting strategies: the conventional random split (using the same split as FAVOR) and our strict repository-level split. In the repository-level split, all functions from a given project are confined to a single set (training, validation, or test). To ensure a fair comparison, we fine-tune all models for each strategy. To prevent data contamination, we exclude the pre-trained adaptor modules of VulMaster, as their training relies on a bug-fixing corpus with known project-level overlaps with the BigVul test set.

Observation. The results, summarized in Figure 7, reveal a significant cross-repository generalization gap. Under the strict repository-level split, the performance of all baseline models degrades sharply compared to the conventional random split. Across the tested models, CodeBLEU scores drop by a relative 21.49% to 29.70%, while EM scores plummet by 87.88% to 89.86%. The collapse of the EM is particularly revealing. It strongly suggests that existing SFT-based AVR approaches are not learning semantic repair behaviors that can transfer to unseen projects. Instead, they are primarily overfitting on repository-specific surface forms and lexical patterns.

D Dataset

For fair comparison with prior AVR approaches, we adopt the BigVul corpus (Fan et al., 2020) as used by VulMaster (Zhou et al., 2024) and FAVOR (Dong et al., 2025) for the main evaluation. We construct a *repository-level* split: all functions from a given repository appear in exactly one of the training, validation, and test sets (no repository overlap). We use an 8:1:1 train/val/test ratio and verify that no repository crosses splits.

To rigorously test how well models generalize to entirely unseen projects, we constructed $\text{PrimeVul}_{\text{AVR}}$, a new AVR test dataset derived from PrimeVul (Ding et al., 2024), a corpus originally created for vulnerability detection in C/C++. Following the data extraction pipeline from VREPAIR (Chen et al., 2022), we created vulnerable-patched function pairs from PrimeVul's CVE-linked commits. To ensure a truly external test set, we then filtered out any function pair whose repository was present in our BigVul training set. This process resulted in a clean, out-of-distribution test set containing 1,554 C/C++ function pairs for evaluating cross-repository generalization.

E Implementation Details

We implement SeCuRepair with HuggingFace Transformers (Hugging Face) for SFT training, vllm (vLLM Project) for inference, and Verl (ByteDance Seed) for RL training. Due to excessive training costs for RL, we select one base model, Qwen2.5-7B-Instruct (Bai et al., 2023), as proof-of-concept. Qwen2.5 is commonly used in software engineering tasks (Yang et al., 2024; Aggarwal et al., 2025; Ke, 2025) and relevant vulnerability detection task (Weyssow et al., 2025) as the

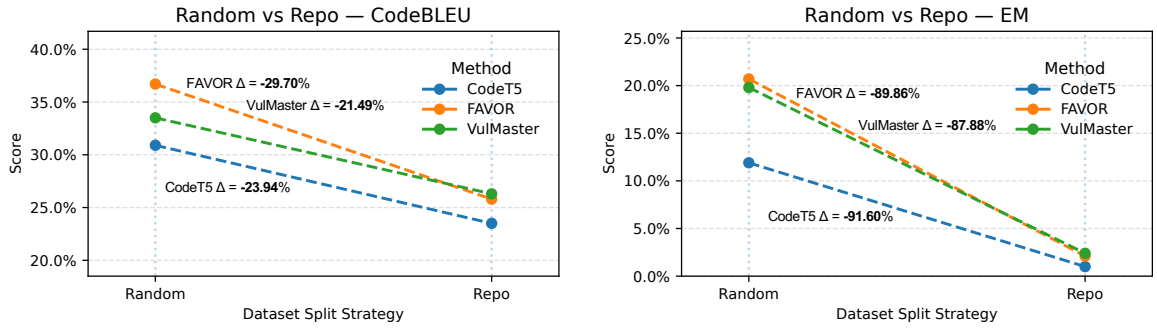

Figure 7: Performance comparison of VulMaster, FAVOR, and CodeT5 under random split and repository-level split. The performance of selected models degrades substantially under repository-level split by up to 91.6% in terms of Exact Match and up to 29.7% in terms of CodeBLEU.

base model. This setting aligns with existing AVR approaches (Chen et al., 2022; Dong et al., 2025), which fine-tune one base model.

SFT implementation. We fine-tune Qwen2.5-7B-Instruct with full-parameter SFT. Loss is computed only on model response: `<reason>` and `<patch>` spans. Sequences are truncated to a cut-off length of 4096 tokens to avoid out-of-memory errors. We train for 3 epochs with cosine learning rate decay and use 10% of the dataset for warm-up. Base learning rate is 3.0×10^{-5}. Per-device batch size is 4 with gradient accumulation of 4.

RL implementation. We further optimize SeCuRepair with GRPO-style reinforcement learning using the Verl library, initialized from the best SFT checkpoint on the validation set. We set the train batch size to 1024, with PPO mini-batches of 64. The actor learning rate is 1.0×10^{-6}, with gradient checkpointing enabled and FSDP offloading for parameters and optimizer states to reduce memory overhead. We generate M=8 rollouts per prompt. We train for at most 20 epochs, saving checkpoints every epoch and saving the best checkpoint on the validation set.

For a controlled comparison, all open-source models (our baselines and SeCuRepair) are trained on our BigVul training split. During evaluation, we use deterministic decoding by setting the temperature to 0 for all models.

F Baselines

We evaluate SeCuRepair against the following groups of baselines:

- **Learning-based AVR approaches**. We select two state-of-the-art AVR approaches, VulMaster (Zhou et al., 2024) and FAVOR (Dong et al., 2025). VulMaster applies CWE expert knowledge to guide the repair process and can handle long input sequences. FAVOR augments the input function with CFG and historical patches. We retrain both on our repository-level split using their recommended hyperparameters.

- **Commercial LLMs**. We also evaluate against a top-tier commercial model GPT-4o (Achiam et al., 2023) to benchmark against the general-purpose state-of-the-art. We prompt GPT-4o with the same instructional wrapper used for SeCuRepair to ensure a fair, direct comparison of repair capabilities.

- **SeCuRepair base model with SFT**. We fine-tune the base model of SeCuRepair, Qwen2.5-7B-Instruct (Bai et al., 2023), on our repository-level BigVul training split as one baseline. We select the best checkpoint based on the performance of the validation set.

G The Brittleness of the Exact Match Metric

The case study in Figure 1 highlights the fundamental brittleness of the Exact Match (EM) metric for evaluating AVR. EM is fundamentally sensitive to syntactic variations that preserve the code's semantics. For example, a logically equivalent change like rewriting `if (a > b)` to `if (b < a)` would still result in an EM score of 0. Indeed, an ideal AVR approach capable of semantic repairing *should* be able to generate a diverse set of semantically correct and equivalent patches, most of which would be wrongly evaluated by EM as they are lexically different from the single oracle patch. This strictness makes EM an unreliable indicator of a patch's correctness and ill-suited for evaluating advanced, semantics-aware AVR approaches. In contrast, CodeBLEU considers AST similarity, making it robust to syntactic variations. Therefore, we consider CodeBLEU a more meaningful metric

for AVR and use it as our primary measure in the following sections.

H Human Evaluation

H.1 Experiment Setting

To complement automatic metrics, we conduct a human evaluation to assess the correctness of the generated patches. We compare SeCuRepair against the best-performing open-source (VulMaster) and commercial baseline (GPT-4o). We recruited four participants, all of whom have at least two years of experience in software security and C/C++ programming. From the $\text{PrimeVul}_{\text{AVR}}$ test set, we randomly sampled 309 examples (calculated for a 95% confidence level with a 5% margin of error). For each sample, participants were shown the vulnerable code alongside the generated patch (presented as a diff). The ground-truth patch was also provided for reference. To prevent bias, the outputs from the different models for each sample were presented in a random order. Given the time-intensive nature of formally verifying a patch's functional correctness, we adopted a widely used proxy for correctness: participants evaluated the extent to which a generated patch preserved the same semantic functionality as the ground-truth solution. Following existing works (Mahbub et al., 2023; Zhang et al., 2020a, 2023, 2020b), each sample is rated by all four participants on a 5-point Likert scale (from 1: not similar at all to 5: exactly the same semantics).

In this work, we aim to support developers with useful repair suggestions rather than replace them with fully automated fixes. Accordingly, we treat scores 1–2 as negative, indicating that the generated patch is of poor quality and unsuitable as a draft for developers. Scores 3–5 are considered positive (workable patch drafts): a score of 3 reflects a patch that captures key logical elements of the oracle patch but is incomplete, while scores 4 and 5 correspond to nearly correct and fully correct patches, respectively. Thus, patches with scores $\geq$3 are regarded as workable, aligning with our goal of providing developers with actionable repair suggestions. 1) Score-3: semantically similar but partial fixes, either incomplete or containing superfluous logic. 2) Score-4: high semantic alignment but differing implementation. 3) Score 5: exact logical match.

Recent work (Takerngsaksiri et al., 2025) also emphasizes the importance of keeping humans in the loop, with automated tools serving primarily as recommenders. Following this perspective, we evaluate how well SeCuRepair delivers workable patch drafts that serve as strong starting points for developers.

H.2 Qualitative Case Study: Syntactically Diverse Semantic Repair

We show one example of SeCuRepair, in which SeCuRepair-generated patch is syntactically different but semantically identical to oracle, in Figure 8. This example highlights SeCuRepair's performance in terms of its reasoning quality, the generated patch's correctness, and its ability to achieve semantic equivalence through a syntactically different solution.

The vulnerable function sampled from $\text{PrimeVul}_{\text{AVR}}$ (`CVE-2021-29513` (NIST National Vulnerability Database)), contains a critical NULL pointer dereference vulnerability (CWE-476 (MITRE)). The code at line 7 directly dereferences the `descr->fields` pointer in a call to `PyDict_Next` without first verifying that it is not a null pointer. This could lead to a crash if the function is called with an improperly initialized descriptor.

As shown in Figure 8, SeCuRepair's reasoning for the fix is both concise and accurate. It correctly pinpoints the exact cause: the `descr->fields` pointer being used without validation. Furthermore, the generated patch is functionally correct and semantically equivalent to the oracle, successfully mitigating the vulnerability by introducing a guard condition before the pointer is dereferenced.

We also highlight that SeCuRepair's patch and oracle patch are syntactically different. Our system employs the C-style null check `!descr->fields`, while the oracle patch uses the explicit C++ `descr->fields == nullptr`. Additionally, the error messages, while functionally similar, use different strings. This distinction is significant as it reveals that the SeCuRepair is not merely performing a surface-level pattern match. Instead, it understands the underlying intent of the security fix, allowing it to generate a valid and effective repair.

H.3 Quantitative Analysis

Human annotators are asked to summarize the common failure modes in those workable but imperfect patches (scored 3 and 4). We observe score-3 patches exhibit three common failure modes: over-defensive fixes (e.g., excessive null checks that

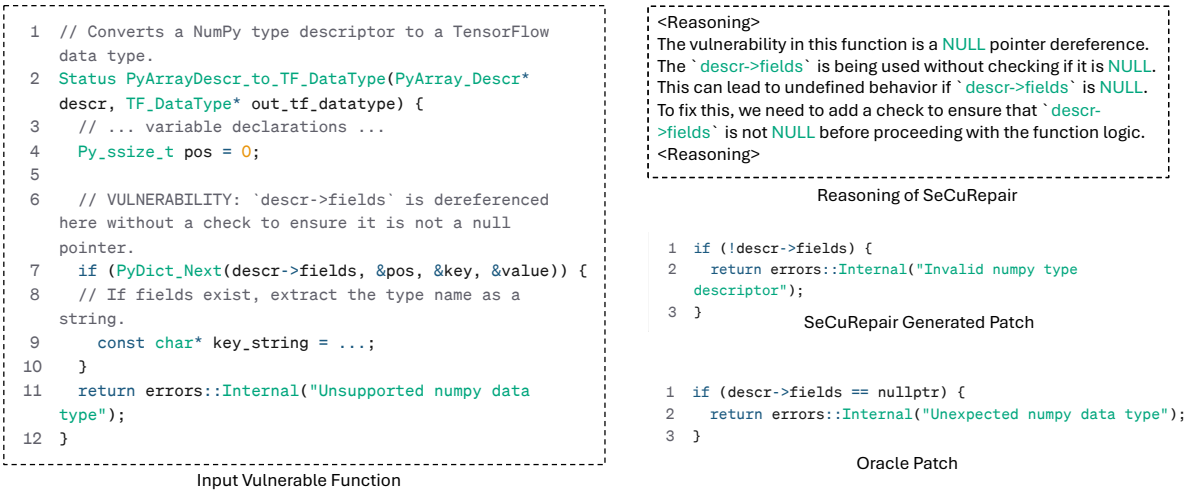

Figure 8: An example of SeCuRepair performing semantic repair with syntactic different patch with oracle.

Table 4: Comparison of SeCuRepair with SeCuRepair-SFT on top-10 most dangerous CWE.

Rank	CWE-Type	Name	SeCuRepair	SeCuRepair-SFT	# Samples
1	CWE-79	Cross-Site Scripting	**0.303**	0.232	7
2	CWE-787	Out-of-bounds Write	**0.313**	0.273	230
3	CWE-89	SQL Injection	**0.430**	0.201	2
4	CWE-352	Cross-Site Request Forgery	**0.541**	0.486	1
5	CWE-22	Path Traversal	**0.297**	0.267	23
6	CWE-125	Out-of-bounds Read	**0.359**	0.276	174
7	CWE-78	OS Command Injection	0.160	**0.165**	9
8	CWE-416	Use After Free	**0.272**	0.265	57
9	CWE-862	Missing Authorization	NA	NA	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	NA	NA	0

may alter intended behavior), minor logic gaps or partially applied fixes, and patches entangled with faulty refactoring. Score-4 patches are semantically similar to the Oracle but differ in implementation, often being syntactically verbose, less readable, over-commented, or using alternative APIs that may have minor semantics differences.

I Per-CWE Performance on MITRE Top-10 Vulnerabilities

To better understand the impact of SeCuRepair, we conducted a fine-grained analysis of model performance on the top-10 most dangerous CWE types (MITRE, 2024) within the $\text{PrimeVul}_{\text{AVR}}$ dataset. Table 4 compares the CodeBLEU scores of the full SeCuRepair framework against SeCuRepair-SFT, the SFT-only variant of SeCuRepair that performs best among all baselines.

The results clearly demonstrate the benefits of SeCuRepair. SeCuRepair consistently outperforms SeCuRepair-SFT across nearly all the most dangerous CWE categories where data is available. The improvements are particularly pronounced for high-impact vulnerabilities such as SQL Injection (CWE-89), where SeCuRepair achieves a score of 0.430 compared to 0.201 for SeCuRepair-SFT, and Cross-Site Scripting (CWE-79), with an improvement from 0.232 to 0.303. This suggests that the RL stage with syntactic- and semantics-aware reward is highly effective at guiding the model to learn the structural patterns required to fix common injection and memory safety flaws.

J Comparison with the Teacher Model

SeCuRepair's SFT stage is bootstrapped with reasoning data distilled from GPT-5-mini. A natural question is whether the subsequent RL stage enables SeCuRepair to surpass its teacher, or whether the observed gains merely reflect the distilled knowledge. To answer this, we evaluate GPT-5-mini on BigVul under the same prompting and evaluation protocol used for GPT-4o in Table 1.

As shown in Table 5, SeCuRepair achieves a 16.70% relative improvement over GPT-5-mini in CodeBLEU. Since the teacher's reasoning traces serve only as a warm start for the SFT stage, this

Table 5: Comparison between SeCuRepair and its teacher model GPT-5-mini on BigVul.

Model	CodeBLEU (%)	Δ over Teacher
GPT-5-mini (Teacher)	30.96	–
SeCuRepair	**36.13**	**+16.70%**

Table 6: Backbone-controlled comparison on BigVul. All models share the Qwen2.5-7B-Instruct backbone.

Model	CodeBLEU (%)
Qwen2.5-7B-Instruct (zero-shot)	24.49
VulMaster-Qwen (S1, Naive)	23.23
VulMaster-Qwen (S2, Structured)	26.94
Qwen2.5-7B-Instruct SFT (Patch-only)	25.78
Qwen2.5-7B-Instruct SFT (Patch + Reasoning)	27.57
SeCuRepair	**36.13**

gap indicates that the subsequent semantics-aware RL training is the primary source of SeCuRepair's performance gains, enabling the student model to surpass its teacher.

K Controlled Comparison under Shared Backbone

The baselines in Table 1, FAVOR and VulMaster, are built on CodeT5, whereas SeCuRepair is built on Qwen2.5-7B-Instruct. To isolate the contribution of the training framework from that of the backbone, we re-implement VulMaster on the same Qwen2.5-7B-Instruct backbone and compare it against SeCuRepair on BigVul under identical training and evaluation protocols.

Since VulMaster was originally designed as an encoder-decoder model with input fusion modules, adapting it to a decoder-only backbone is non-trivial. We consider two variants: **S1 (Naive)** concatenates all input components (vulnerable function, AST features, CWE exemplars) into a flat prompt; **S2 (Structured)** adds explicit segment delimiters and task-oriented instructions to better organize the input. Both variants follow SeCuRepair's SFT protocol.

As shown in Table 6, SeCuRepair outperforms the best VulMaster-Qwen variant (S2) by a relative 34.11% in CodeBLEU under the identical backbone. Since the two systems share the same base model, training data, and evaluation protocol, this gap isolates the contribution of SeCuRepair's training framework from that of backbone capacity.