

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

CHENGRAN YANG, Singapore Management University, Singapore

TING ZHANG, Monash University, Australia

JINFENG JIANG and XIN ZHOU, Singapore Management University, Singapore

HAOYE TIAN, Aalto University, Finland

JIEKE SHI, JUNKAI CHEN, YIKUN LI, ENG LIEH OUH, LWIN KHIN SHAR, and DAVID LO, Singapore Management University, Finland

Current learning-based Automated Vulnerability Repair (AVR) approaches, while promising, often fail to generalize effectively in real-world scenarios. Our diagnostic analysis reveals three fundamental weaknesses in state-of-the-art AVR approaches: (1) limited cross-repository generalization, with performance drops on unseen codebases; (2) inability to capture long-range dependencies, causing a performance degradation on complex, multi-hunk repairs; and (3) over-reliance on superficial lexical patterns, leading to significant performance drops on vulnerabilities with minor syntactic variations like variable renaming.

To address these limitations, we propose SeCuRepair, a semantics-aligned, curriculum-driven, and reasoning-enhanced framework for vulnerability repair. At its core, SeCuRepair adopts a reason-then-edit paradigm, requiring the model to articulate why and how a vulnerability should be fixed before generating the patch. This explicit reasoning enforces a genuine understanding of repair logic rather than superficial memorization of lexical patterns. SeCuRepair also moves beyond traditional supervised fine-tuning and employs semantics-aware reinforcement learning, rewarding patches for their syntactic and semantic alignment with the oracle patch rather than mere token overlap. Complementing this, a difficulty-aware curriculum progressively trains the model, starting with simple fixes and advancing to complex, multi-hunk coordinated edits.

We evaluate SeCuRepair on strict, repository-level splits of BigVul and newly crafted PrimeVul_{AVR} datasets. SeCuRepair significantly outperforms all baselines, surpassing the best-performing baselines by 34.52% on BigVul and 31.52% on PrimeVul_{AVR} in terms of CodeBLEU, respectively. Our human evaluation of patch correctness further shows that SeCuRepair generates 16% more workable patches than the best-performing baseline, GPT-4o. Comprehensive ablation studies further confirm that each component of our framework contributes to its final performance.

CCS Concepts: • **Security and privacy** → **Software and application security**.

Additional Key Words and Phrases: Vulnerability Repair, Large Language Model, Reinforcement Learning

Authors' Contact Information: Chengran Yang, cryang@smu.edu.sg, Singapore Management University, Singapore; Ting Zhang, ting.zhang@monash.edu, Monash University, Australia; Jinfeng Jiang, jfjiang@smu.edu.sg; Xin Zhou, xinzhou.2020@phdcs.smu.edu.sg, Singapore Management University, Singapore; Haoye Tian, haoye.tian@aalto.fi, Aalto University, Finland; Jieke Shi, jiekeshi@smu.edu.sg; Junkai Chen, junkaichen@smu.edu.sg; Yikun Li, yikunli@smu.edu.sg; Eng Lieh Ouh, elouh@smu.edu.sg; Lwin Khin Shar, lkshar@smu.edu.sg; David Lo, davidlo@smu.edu.sg, Singapore Management University, Finland.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2025/10-ART

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

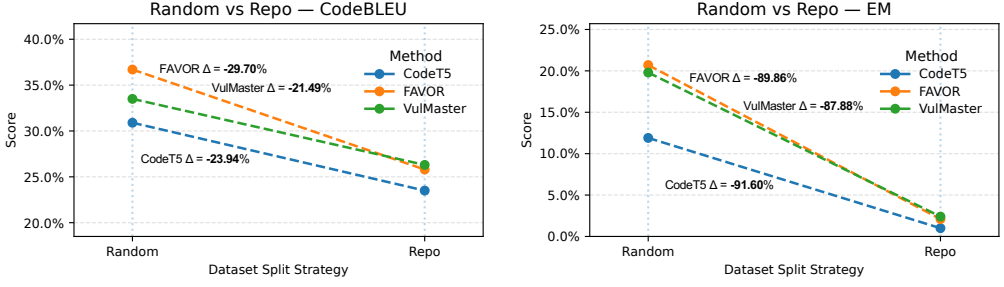


Fig. 1. Limitation 1: Performance comparison of VulMaster, FAVOR, and CodeT5 under random split and repository-level split. The performance of selected models degrades substantially under repository-level split by up to 91.6% in terms of Exact Match and up to 29.7% in terms of CodeBLEU.

ACM Reference Format:

Chengran Yang, Ting Zhang, Jinfeng Jiang, Xin Zhou, Haoye Tian, Jieke Shi, Junkai Chen, Yikun Li, Eng Lieh Ouh, Lwin Khin Shar, and David Lo. 2025. Semantics-Aligned, Curriculum-Driven, and Reasoning-Enhanced Vulnerability Repair Framework. 1, 1 (October 2025), 21 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 Introduction

Software vulnerabilities are weaknesses in software that attackers can exploit to violate security policies, e.g., confidentiality, integrity, or availability [1]. The number of reported software vulnerabilities continues to rise steadily. In 2024, the National Vulnerability Database (NVD) published 40,009 Common Vulnerabilities and Exposures (CVE) records, representing a 38% increase compared to 2023. Despite this growing trend in reporting, the verification and remediation of CVEs remain challenging and have not kept pace with this growth. There are still more than 25,900 CVEs awaiting initial analysis by NVD [8] (as of 01/09/2025) and they often take weeks to be fixed [4, 21], highlighting systemic triage pressure and slower downstream remediation. As a result, these trends motivate automatic vulnerability repair approaches to help in vulnerability patching.

A number of state-of-the-art and learning-based approaches have been proposed for automatic vulnerability repair (AVR), which frame AVR as function-level sequence-to-sequence generation [22]. Given a vulnerable function and optional auxiliary contexts, an AVR model is trained to generate a patch. Representative AVR approaches, including VREPAIR [17], VulMaster [44], and FAVOR [19], perform supervised fine-tuning (SFT) with single-reference patches, enriched with auxiliary signals. In particular, VREPAIR [17] introduces a transfer-learning pipeline that pretrains on large bug-fixing corpora. Also, VulMaster [44] accommodates long function input and integrates CWE knowledge to guide repair. Moreover, FAVOR [19] augments the input with historical patch patterns.

Despite their success, we identify three key limitations of existing learning-based AVR approaches.

Limitation 1: Limited cross-repository generalization. Prevailing AVR approaches [17, 19, 44] are evaluated on random-split datasets, which overlook cross-repository generalization. Given the scale of real-world code (over 420M GitHub repositories [5]) versus the limited available vulnerability corpora, a practical AVR approach must generalize to unseen projects. To assess this, we re-evaluate existing AVR approaches by splitting the dataset without repository overlap between the training and test sets (details in §2.2.1). As shown in Fig. 1, with this realistic setup, we reveal a substantial generalization gap of existing learning-based AVR approaches: they can not maintain high effectiveness as before, with CodeBLEU dropping by up to 29.7% and Exact-Match up to 91.6%.

Additionally, a qualitative case study on CVE-2018-16228 (Fig. 2) further highlights the brittleness of prevailing AVR approaches on semantically equivalent vulnerable code. While VulMaster

```

1 print_prefix(netdissect_options *ndo, const u_char *prefix, ...) {
2     int plenbytes;
3     char buf[64]; // [] buffer is uninitialized
4     if (prefix[0] >= 96 && is_ipv4_mapped_address(&prefix[1])) {
5         // IPv4-mapped branch (details omitted)
6     } else {
7         plenbytes = decode_prefix6(ndo, prefix, buf, ...);
8         // True FIX
9         if (plenbytes < 0) return plenbytes;
10    }
11    ND_PRINT((ndo, "%s", buf));
12    return plenbytes;
13 }

```

```

1 print_prefix(netdissect_options *ndo, const u_char *prefix, ...) {
2     int nbytes;
3     char buf[64]; // [] buffer is uninitialized
4     if (prefix[0] >= 96 && is_ipv4_mapped_address(&prefix[1])) {
5         // IPv4-mapped branch (details omitted)
6     } else {
7         nbytes = decode_prefix6(ndo, prefix, buf, ...);
8         // Hallucinated FIX
9         ND_PRINT(*ndo);
10    }
11    ND_PRINT((ndo, "%s", buf));
12    return nbytes;
13 }

```

(a) Perfect Repair Patches Generated by VulMaster

(b) Incorrect Patches Generated by VulMaster after Fine-tuned on Random-split Dataset. Renaming One Local Variable plenbytes to nbytes.

Fig. 2. VulMaster fails to generate the buffer over-read guard when a local variable is renamed with a similar naming convention.

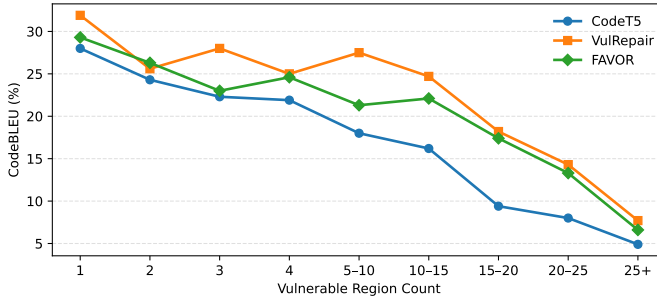


Fig. 3. Limitation 2: The performance of existing AVR approaches drops as the number of hunks increases.

correctly patches the original function, it fails after syntactic-modified but semantics-preserving edits (e.g., identifier renaming), revealing a critical sensitivity to syntactic variations.

We attribute this poor cross-repository generalization and syntactic brittleness to the models' overfitting on spurious and repository-specific lexical correlations rather than learning *syntactic* and *semantic* repair patterns. The prevalent training objective in AVR approaches, maximizing token likelihood against a single oracle patch, provides no supervision to distinguish meaningful semantic edits from superficial lexical cues (e.g., specific variable names or API idioms). The loss function simply rewards matching the oracle's tokens, allowing lexical patterns to be learned instead of the underlying repair logic. These findings motivate the advancement of AVR approaches that jointly account for syntactic and semantic features in repair, while emphasizing the necessity of rigorous, cross-repository evaluation.

Limitation 2: Unsolved multi-hunk repair optimization. Many real-world vulnerabilities require multi-hunk repairs: a set of coordinated edits across different, often distant, hunks of a function. For example, in vulnerability datasets like BigVul and PrimeVul, 39.33% and 47.07% of the vulnerable functions require multi-hunk repairs, respectively. Unfortunately, as shown in Fig. 3 (detailed in Sec 2.2.3), the performance of representative AVR approaches generally degrades with the number of hunks. Specifically, VulMaster, FAVOR, and CodeT5 performance in terms of CodeBLEU drops by 19.75%, 11.27%, and 13.22%, respectively, when moving from 1 to 2 hunks. Our finding highlights the need for AVR approaches to perform better for multi-hunk repairs, which are common in real-world scenarios.

Limitation 3: Explicit repair reasoning, underexplored yet broadly useful. Current AVR approaches operate like black-box translators: converting vulnerable code directly into a patched

version without generating any explicit reasoning about the fix. They do not articulate the vulnerability's root cause, the required logical changes, or a step-by-step repair plan. They force the models to generate a patch based only on local patterns learned from the data, without a mechanism to enforce the global consistency of the fix.

Recent work suggests a path forward: structured reasoning approaches have improved performance in vulnerability detection [37], while causal learning methods can eliminate reliance on spurious features [30]. We argue that AVR can benefit by making repair intent explicit. Rather than directly generating patches, models should first articulate a repair plan, then implement it. Yet this reasoning-then-edit approach remains largely unexplored for AVR.

Our Work. We present **SeCuRepair**, a Semantic and syntactic-aligned, Curriculum-driven, Reasoning-enhanced AVR framework, to address the three limitations above directly: 1) **Rewarding semantically and syntactically aligned patches** (addressing Limitation 1). To combat overfitting on spurious token patterns, SeCuRepair's training objective moves beyond SFT's token-matching. Rather, it uses a syntactic- and semantic-aligned reinforcement learning (RL) objective that rewards generated patches for their syntactic (Abstract Syntax Tree) and semantic (Data Flow Graph) similarity to the oracle patch. This encourages the model to learn the behavioral repair logic rather than just mimicking lexical patterns. 2) **Curriculum scheduling** (addressing Limitation 2). To master multi-hunk repairs, SeCuRepair employs a difficulty-aware curriculum. The model is trained progressively, starting with patches with less number of hunks before advancing to patches requiring multiple, non-contiguous edits. This approach allows the model to master foundational repair patterns before tackling the intricate dependencies of more complex vulnerabilities. 3) **Reason-then-edit generation** (addressing Limitation 3). To prevent inconsistent or partial fixes, SeCuRepair adopts a reason-then-edit paradigm. It first generates a concise, natural-language plan for the repair and then produces the code conditioned on that plan. This provides a strong guide for generating globally consistent and logically sound edits.

We conduct a comprehensive evaluation of SeCuRepair on two benchmarks: the widely-used BigVul dataset [20] and PrimeVul_{AVR}, a dataset we adapt from vulnerability detection and extend for repair. We compare SeCuRepair against state-of-the-art AVR approaches, including VulMaster [44], FAVOR [19], and the general-purpose GPT-4o. The results demonstrate the superiority of our approach. SeCuRepair outperforms the strongest baseline GPT-4o by a significant margin, achieving relative CodeBLEU improvements of 34.52% on BigVul and 31.52% on PrimeVul_{AVR}. Meanwhile, our human evaluation of patch correctness shows that SeCuRepair generates 16% more workable patches than the best-performing baseline, GPT-4o.

Furthermore, a series of ablation studies confirms the individual effectiveness of each core component within the SeCuRepair framework. In summary, the main contributions are as follows:

- **Empirical Analysis.** We are the first to systematically analyze the challenges of realistic cross-repository generalization and multi-hunk repairs in existing AVR approaches.
- **Approach.** We design and implement SeCuRepair, a novel AVR framework that integrates syntactic- and semantic-aligned reinforcement learning, curriculum-based training, and a reason-then-edit workflow to address these identified challenges.
- **Evaluation.** Our extensive experiments demonstrate that SeCuRepair consistently outperforms existing methods on BigVul and PrimeVul_{AVR}.

2 Problem Statement and Empirical Analysis

In this section, we first formally define the problem of AVR and then empirically analyze the challenges in existing AVR approaches: cross-repository generalization, multi-hunk repair performance, and brittleness to semantics-preserving code refactoring.

2.1 Problem Definition

Following prior works [17, 19, 44], we formulate AVR as a function-level sequence-to-sequence generation task. Specifically, given a vulnerable function X and auxiliary information L (e.g., CWE information and localization information), the goal is to produce a repair function Y that fixes the vulnerability in X . An AVR model is trained on a dataset $\mathcal{D} = \{(X_i, L_i, Y_i)\}_{i=1}^N$ of N examples. Existing AVR approaches [17, 19, 44] typically optimize the model with supervised-fine-tuning (SFT) to maximize the likelihood of the reference patch Y_i given X_i and L_i :

$$\mathcal{L}_{SFT} = - \sum_{i=1}^N \log P(Y_i | X_i, L_i; \theta) \quad (1)$$

where θ denotes the model parameters.

2.2 Empirical Analysis

To systematically assess the limitations of current AVR approaches, we conduct a series of controlled experiments using a consistent evaluation protocol. We focus on three representative SFT-based baselines, i.e., VulMaster [44], FAVOR [19], and their base LLM CodeT5 [36]. Following standard practice [17, 19, 44], we report performance using CodeBLEU [31] and Exact Match (EM) scores. In addition, we also conduct manual evaluation to assess the correctness of generated patches and the quality of the reasoning. Each subsequent subsection introduces one experimental variant targeting (i) cross-repository generalization, (ii) multi-hunk repair, and (iii) robustness to semantics-preserving refactoring.

2.2.1 Cross-Repository Generalization Gap.

Setup. As noted in Section 1, 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 [20]. 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 1, 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.

The performance of selected AVR approaches drops substantially under repository-level split by up to 89.86% in terms of Exact Match and up to 29.70% in terms of CodeBLEU.

2.2.2 Brittleness to Semantics-Preserving Code Refactoring.

Setup. To probe whether AVR approaches learn real semantic repair patterns or merely overfit to lexical cues, we test their robustness under a simple refactoring: renaming local variables while

preserving semantics. We randomly selected 10 cases where baseline models (trained and evaluated under random splits) correctly generated a patch, then applied consistent variable renamings.

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 2 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.

The Brittleness of the Exact Match Metric. The case study above also 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.

The failure of existing AVR approaches on simple, semantics-preserving changes reveals their over-reliance on lexical patterns over semantic repair patterns. This same flaw suggests EM may have limitations as a metric for AVR, as it is confined to judging lexical equivalence with the oracle patch.

2.2.3 Suboptimal on Multi-Hunk Repair.

Setup. Real-world vulnerability fixes frequently span multiple, non-contiguous regions (*hunks*) that must be edited consistently. Prior work FAVOR [19]’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.

Observation. Figure 3 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.

The performance of AVR models decreases as the complexity of repairs increases. All studied models become progressively less effective as the number of required edits (hunks) increases.

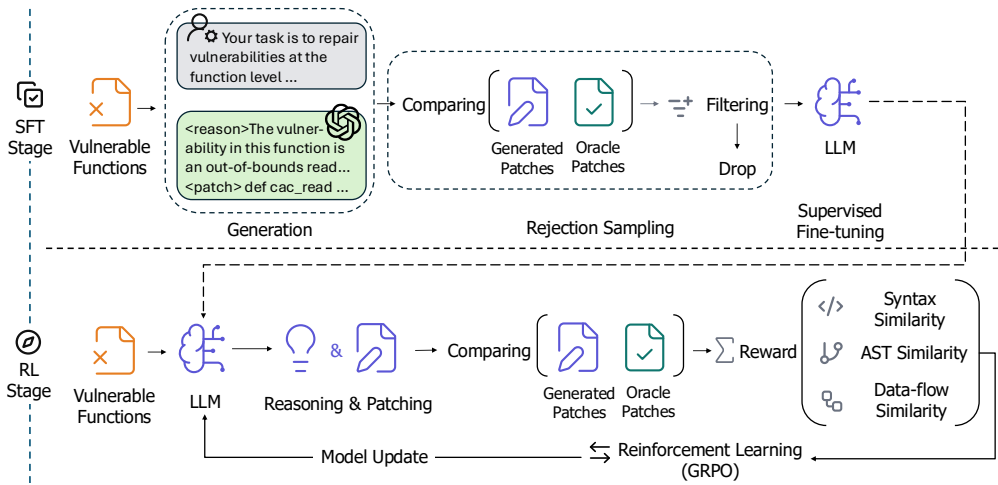


Fig. 4. Overview of the SeCuRepair training pipeline. (1) Stage 1: Reasoning-Enhanced SFT. A teacher LLM generates (reasoning, patch) candidates, which are filtered via rejection sampling to create a high-quality dataset. The student model is then fine-tuned to learn the reason-then-edit format. (2) Stage 2: Semantics-Guided RL. The SFT model is refined using RL. The policy is rewarded based on a composite score of syntactic (BLEU) and semantic (AST, DFG) similarity to the oracle patch. Stage 2 is organized with a curriculum, progressively training the model on repairs with an increasing number of hunks to master multi-hunk fixes.

3 Approach

Inspired by the analysis in Section 2.2, we propose SeCuRepair, a framework designed to learn the underlying syntactic and semantic logic of vulnerability repairs rather than imitating superficial, repository-specific lexical patterns. The design of SeCuRepair is guided by three core principles:

- **A Reason-then-Edit Paradigm:** To ensure logical consistency in repairs, the model is trained to first generate an explicit, natural-language plan that reasons the vulnerability and the proposed fix before it writes the code.
- **Semantics-Aware Optimization:** To promote robust generalization, the model is optimized using rewards that measure the syntactic and semantic correctness of a patch, moving beyond simple textual similarity to a single reference solution.
- **Difficulty-based Curriculum Learning:** To master complex, multi-hunk fixes, the model is trained on a curriculum of increasing difficulty, allowing it to learn foundational repair patterns before advancing to more challenging scenarios.

Framework. These principles are realized through a two-stage training pipeline, as shown in Figure 4. In the first stage, we bootstrap the model’s reasoning ability. We employ a commercial teacher LLM to generate high-quality (reasoning, patch) examples, and then perform Supervised Fine-Tuning (SFT) on our open-source student model using a subset of this curated data selected through rejection sampling. This stage provides a strong initial LLM checkpoint to master the Reason-then-Edit paradigm. In the second stage, we refine the model’s patching ability using RL. The model is optimized through a syntactic- and semantic-aware RL optimization, which uses rewards measuring both syntactic and semantic correctness, and guided by difficulty-based curriculum learning, which gradually increases task complexity to multi-hunk fixes. This stage encourages exploration and the generation of functionally correct patches. The final, trained SeCuRepair model takes a vulnerable function and produces a structured output containing both the repair strategy and the corresponding patch code.

3.1 Reasoning-transferred SFT

The first stage of our pipeline aims to initialize the model to follow a reason-then-edit paradigm. We posit that generating an explicit repair plan before generating patches improves both controllability and generalization. A plan forces the AVR model to first articulate the “what, where, and how” of a fix, providing a coherent guide for the subsequent patch generation. This abstract rationale guides the model to capture semantic repair patterns rather than the lexical details of a single patch.

3.1.1 Knowledge Distillation. To achieve this, the first step is to construct a dataset of (reasoning, patch) pairs by distilling reasoning knowledge from a powerful commercial teacher LLM (i.e., GPT-5 mini [6]). This process transfers the teacher’s advanced reasoning capabilities to our open-source student model, enabling it to infer nuanced, generalizable reasoning patterns from the generated examples. For each vulnerability in the training set, we prompt the teacher to produce one candidate (reasoning, patch) pairs. The student model (SeCuRepair) is then trained on a quality-filtered subset using a standard SFT objective. While SFT can lead to overfitting on lexical patterns (as discussed in Section 2.2), we intentionally leverage this tendency here. The model’s inclination for token-level mimicry is highly effective for learning the rigid output schema (i.e., ...<reason>...</reason><patch>...</patch>). The potential for overfitting on the patch content is a secondary concern at this stage, as it will be mitigated by the subsequent RL training. Our ablation studies (Section 5.3.2) also confirm that initializing with this SFT stage leads to more stable and effective RL training compared to starting from scratch.

3.1.2 Rejection Sampling. Outputs from a teacher LLM can be noisy, with flawed rationales and incorrect patches (prior work [19] reports that even GPT-4o often produces low-quality patches). To ensure the student model learns only from high-quality examples, we adopt a strict rejection-sampling strategy to denoise the distilled data. Specifically, we apply a two-step filtering process to the K candidates generated for each vulnerability:

- **Syntactic Filtering:** We first discard any responses that violate the output schema, such as those with missing, misordered, or malformed <reason> or <patch> tags.
- **Semantic Filtering:** For the remaining candidates, we adopt a strong heuristic proxy for reasoning quality: we retain a (reasoning, patch) pair only if its patch is similar to the ground-truth oracle patch with CodeBLEU > 0.5. Our assumption is that the similarity to the oracle solution serves as a strong, albeit imperfect, proxy of the quality of the associated reasoning.

After rigorous filtering, we retain 484 high-quality examples distilled from the teacher LLM. This denoised dataset is then used to fine-tune SeCuRepair, yielding a high-fidelity starting policy for the subsequent reinforcement learning stage.

3.1.3 Supervised-fine-tuning Objective. After rejection sampling, we obtain a high-precision keep set $\mathcal{D}_{\text{keep}} = \{(X_i, L_i, r_i, y_i)\}$, where r_i is the teacher’s reasoning trace. Each r_i instance is serialized in the reason-then-edit schema: <reason> r_i </reason> <patch> y_i </patch>.

Following existing approaches [17, 19, 44], we fine-tune the student model with a standard SFT objective over the keep set $\mathcal{D}_{\text{keep}}$:

$$\mathcal{L}_{\text{SFT}} = \frac{1}{\sum_i T_i^R} \sum_i \sum_{t=1}^{T_i^R} -\log \pi_{\theta}(r_{i,t}, y_{i,t} \mid P_i, r_{i,<t}),$$

where P_i is the input prompt (vulnerable function and instructions), the R_i is the model response that includes reasoning trace and patch (r_i, y_i), and θ represents the model’s parameters. Note that the loss is computed only on response tokens. T_i^R is the total number of tokens in the response.

The resulting model, denoted as θ_{SFT} , is now capable of generating structured reasoning before patching and serves as the initial policy for the subsequent reinforcement learning stage.

3.2 Reinforcement Learning

While the SFT stage provides a format-faithful starting point for reasoning, it falls short in enabling the model to learn generalizable repairs. By training the model to mimic a single reference patch with token-level maximum likelihood, SFT conflates essential repair logic with spurious lexical styles. To overcome this, our second stage uses RL with semantics-aware rewards to refine the model. Our RL design addresses this objective mismatch by enabling the policy model, initialized from θ_{SFT} , to explore the vast space of candidate patches and to learn their structural and semantic repair logic, rather than merely producing textually identical outputs.

We employ an on-policy RL algorithm GRPO [32] for this refinement process. During training, the model generates a (reasoning, patch) pair for a given vulnerability. We then compute a scalar reward based on the generated patch, $\hat{y}_i$. This semantics-aware reward represents a critical component of our RL design: it moves beyond simple token overlap to measure the syntactic (Abstract Syntax Tree, AST) and semantic (Data Flow Graph, DFG) similarities between the generated patch and the oracle solution using tree-matching. This reward signal is then used to update the model's policy, encouraging it to generate patches that achieve higher semantic agreement with the oracle. The final, tuned policy model, π_θ , serves as the backend model for SeCuRepair.

3.2.1 Generation. The reinforcement learning process begins with on-policy generation. At each iteration, for each input prompt $P_i = (L_i, X_i)$ in a training batch, the current policy π_θ (initialized from θ_{SFT}) generates M complete (reasoning, patch) sequences. We then parse these M responses to automatically extract the set of candidate patches (i.e., $\hat{y}_{i1}, \dots, \hat{y}_{iM}$) by matching the patch tags. This collection of generated patches represents the policy's current behavior and is then passed to our reward function for evaluation, providing the necessary signal for the policy update.

3.2.2 Reward. To guide the policy toward learning syntactic and semantic patterns for vulnerability repair, we designed a semantics-aware reward function, Re . This function computes a scalar score that estimates the quality of a generated patch, $\hat{y}_i$, by measuring its syntactic and semantic agreement with the oracle patch, y_i . A higher reward signal indicates that the generated patch is more functionally aligned with the ground-truth solution. The reward is then converted into normalized advantages and used to guide the policy update.

Specifically, our reward function assesses the agreement between the generated patch and the oracle patch at three distinct levels: 1) lexical agreement; 2) syntactic agreement; and 3) semantic agreement. The final reward is a combination of these three scores. Crucially, our reward function is training-free and rule-based, ensuring it provides a consistent and efficient signal. Below, we provide a detailed description of the reward design.

Lexical Agreement. Inspired by existing works in code generation and AVR [17, 31, 44], we adopt BLEU to approximate token similarity between patches. BLEU measures the proportion of overlapping n -grams, providing a baseline reward for using correct keywords and identifiers to mitigate token-level hallucination. Formally, the BLEU score for a generated patch $\hat{y}$ against an oracle patch y is defined as:

$$\text{BLEU} = \text{BP} \cdot \exp \left(\sum_{n=1}^N \frac{\log p_n}{N} \right), \quad (2)$$

where p_n is the n -gram precision, and BP is the brevity penalty to discourage excessively short generations.

Syntactic Agreement. This aspect of the reward measures how closely the generated patch's structure matches the oracle's structure via the AST similarity measure. The AST represents the code's hierarchical syntax, where each node corresponds to a syntactic construct, and edges encode their hierarchical relationships. Each code snippet is decomposed 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:

$$\text{Sim}_{\text{AST}}(\hat{y}, y) = \frac{|S(\hat{y}) \cap S(y)|}{|S(y)|}. \quad (3)$$

Semantic Agreement. In addition to AST, we leverage the DFG to capture semantic consistency in terms of variable usage and data dependencies. The DFG represents how values propagate through a program: each node corresponds to a variable, and directed edges denote data dependencies. Each code snippet is decomposed into a set of data-dependency tuples in the form (v_i, v_j) , representing a flow edge from variable v_i to v_j . We then measure similarity as the fraction of matched edges:

$$\text{Sim}_{\text{DFG}}(\hat{y}, y) = \frac{|E(\hat{y}) \cap E(y)|}{|E(y)|}, \quad (4)$$

where $E(\hat{y})$ and $E(y)$ denote the sets of data-flow edges extracted from $\hat{y}$ and y , respectively.

Final Reward Computation. We define a reward vector $\mathbf{r} = \langle \text{BLEU}(\hat{y}, y), \text{Sim}_{\text{AST}}(\hat{y}, y), \text{Sim}_{\text{DFG}}(\hat{y}, y) \rangle$, which components are 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 L1norm 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.

3.3 GRPO Optimization

In the RL stage, we optimize the policy model using Group Relative Policy Optimization (GRPO) [32] with the reward signal $Re(\hat{y}, y)$, which is applied exclusively during training phase with training dataset. In each iteration, we rollout the policy to explore repair spaces, assign semantic-aware rewards to the patches, and update the policy to favor the patches with higher reward.

3.3.1 Overall Process. Each update iteration of GRPO follows three steps:

- **Rollout.** For each vulnerable function P_i , the current policy π_θ generates M candidate outputs in the reasoning-then-edit format $\{(r_{i1}, \hat{y}_{i1}), \dots, (r_{iM}, \hat{y}_{iM})\}$. These represent diverse repair hypotheses explored by the policy.
- **Reward and Advantage Estimation.** Each candidate $\hat{y}_{ij}$ is scored by our rewarder Re , yielding $R_{ij} = Re(\hat{y}_{ij}, y_i)$. To account for reward scale differences across inputs, GRPO normalizes scores within each group. Let μ_i and σ_i be the mean and standard deviation of the M rewards for prompt P_i , then the normalized advantage is $A_{ij} = \frac{R_{ij} - \mu_i}{\sigma_i + \epsilon}$, where ϵ is a small constant used in GRPO implementation for numerical stability. Intuitively, a patch is considered *good* if its reward is higher than its siblings for the same vulnerable function.
- **Policy Update.** The policy is updated to increase the likelihood of candidate patches with higher relative advantage derived from the reward. This is done using importance ratios between new and old policies, clipped to avoid large updates, and weighted by the advantages.

3.3.2 Policy Objective. Formally, let $r_{ij}(\theta) = \frac{\pi_\theta(\hat{y}_{ij} | P_i)}{\pi_{\theta_{\text{old}}}(\hat{y}_{ij} | P_i)}$ 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:

$$\mathcal{L}_{\text{GRPO}}(\theta) = \mathbb{E}_{i,j} \left[\min(r_{ij}(\theta) A_{ij}, \text{clip}(r_{ij}(\theta), 1 - \epsilon_c, 1 + \epsilon_c) A_{ij}) \right] - \beta \text{KL}[\pi_{\theta_{\text{old}}} \| \pi_\theta], \quad (5)$$

where $\epsilon_c > 0$ is the clipping threshold that restricts the change of importance ratios, preventing overly aggressive policy updates, and $\beta \geq 0$ is the coefficient of the KL-regularization term that penalizes significant deviations from the previous policy.

3.3.3 Outcome. Through iterative rollouts, group-relative reward normalization, and clipped updates, GRPO steers the policy toward generating patches that are lexically faithful, syntactically consistent, and semantically coherent with the oracle.

3.4 Curriculum Learning

Our analysis in Section 2.2.3 confirms that coordinating edits across multiple code hunks is a primary challenge for AVR models, causing significant performance degradation. To address this, we adopt a curriculum learning strategy that organizes training from simple to complex repairs. This allows the model to gradually master the skill of performing consistent, multi-hunk edits.

3.4.1 Difficulty Definition. We use the number of vulnerable hunks as a proxy for repair difficulty, which correlates with the cognitive complexity of the task. We define three stages: easy (1-2 hunks), medium (3-5 hunks), and hard (>5 hunks). Notably, our “easy” stage intentionally includes two-hunk functions. Preliminary experiments showed that training only on single-hunk fixes caused the model to overfit to overly localized repairs. By including simple multi-hunk cases from the start, we ensure the model begins learning the principles of cross-hunk coordination early in its training.

Our curriculum proceeds through the three stages during the RL phase. The training set is *expanded cumulatively* at each stage to prevent catastrophic forgetting, and the model’s policy is always initialized from the previous stage’s checkpoint.

3.4.2 Outcome. This curriculum aligns the training process with the natural hierarchy of repair complexity. By first mastering a foundation of simple, common repair patterns, the model is better equipped to learn the more complex compositions and nuanced logic required for difficult, multi-hunk vulnerabilities, leading to more robust and generalizable performance across the board. Implementation-wise, our curriculum learning is seamlessly integrated with the RL pipeline (Sec. 3.2), as it gradually controls how the training data is presented to the RL process.

4 Experiment Setup

4.1 Implementation Details

We implement SeCuRepair with HuggingFace Transformers [9] for SFT training, vllm [11] for inference, and Verl [10] for RL training. Due to excessive training costs for RL, we select one base model, Qwen2.5-7B-Instruct [15], as proof-of-concept. Qwen2.5 is commonly used in software engineering tasks [13, 25, 39] and relevant vulnerability detection task [37] as the base model. This setting aligns with existing AVR approaches [17, 19], 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 cutoff 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.

4.2 Dataset

For fair comparison with prior AVR approaches, we adopt the BigVul corpus [20] as used by VulMaster [44] and FAVOR [19] for the main evaluation. We reuse the *deduplicated* BigVul split from FAVOR, yielding 9,333 vulnerable functions paired with corresponding patches in C/C++. 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. Following existing AVR approaches [19, 44], each vulnerable hunk in the input is bracketed with localization markers `<vul_start>` and `<vul_end>`.

To rigorously test how well models generalize to entirely unseen projects, we constructed PrimeVul_{AVR}, a new AVR test dataset derived from PrimeVul [18], a corpus originally created for vulnerability detection in C/C++. Following the data extraction pipeline from VREPAIR [17], 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.

4.3 Baselines and Evaluation Metrics

We evaluate SeCuRepair against the following groups of baselines:

- **Learning-based AVR approaches.** We select two state-of-the-art AVR approaches, VulMaster [44] and FAVOR [19]. 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 [12] 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 [15], on our repository-level BigVul training split as one baseline. We select the best checkpoint based on the performance of the validation set.

On Excluding Agentic Frameworks. We considered including agent-based program repair frameworks as baselines given their recent success. However, the standard agentic loop, which iterates over code localization, repair generation, and execution-based verification [14, 16, 34], does not apply to the function-level AVR task. Vulnerabilities are already localized by definition, and providing execution feedback from a potentially vulnerable system is often disallowed for general security applications. Consequently, the agentic loop collapses to a single repair-generation step. This is functionally equivalent to our zero-shot evaluation of GPT-4o. We therefore report the GPT-4o results directly without additional agent scaffolding.

Automatic Evaluation Metrics. We use CodeBLEU as our primary metric following existing works [19, 44]. Differently, we do not use EM as discussed in our empirical analysis (Section 2.2.2).

Table 1. Comparison of SeCuRepair with state-of-the-art baselines on BigVul and PrimeVul_{AVR}.

AVR Approach	Base Model	Training Strategy	CodeBLEU (%)	
			BigVul	PrimeVul _{AVR}
FAVOR [19]	CodeT5	SFT	25.78	11.77
VulMaster [44]	CodeT5	SFT	26.33	11.62
GPT-4o [12]	GPT-4o	NA	25.90	23.41
SeCuRepair	Qwen2.5-7B	SFT	29.62	25.92
SeCuRepair	Qwen2.5-7B	Reasoning SFT & RL	35.42	30.79

5 Experiment Results

5.1 Research Questions

Our experimental evaluation is designed to answer the following research questions:

- **RQ1: How effective is SeCuRepair in repairing vulnerabilities?**
- **RQ2: How reasoning-transferred SFT improves repair quality and interpretability?**
- **RQ3: How does semantic-aware RL contribute to the performance?**

5.2 RQ1. Effectiveness of SeCuRepair

5.2.1 Automatic Evaluation. Experiment Setting. To evaluate the overall effectiveness of SeCuRepair, we compare it against all baselines on BigVul and our crafted PrimeVul_{AVR} test set (detailed in Section 4.2). For a controlled comparison, all open-source models (our baselines and SeCuRepair) are retrained on our BigVul training split. During evaluation, we use deterministic decoding by setting the temperature to 0 for all models.

Results. The results of our automatic evaluation, presented in Table 1, show 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 34.52%; and surpasses the best-performing baseline on PrimeVul_{AVR}, GPT-4o, by 31.52%. This result demonstrates the superior ability of SeCuRepair to repair vulnerability and generalize to unseen repositories. A Wilcoxon signed-rank test [38] confirms that all of SeCuRepair’s performance gains over the baselines are statistically significant ($p < 0.001$), underscoring the effectiveness and robust generalization capability of our framework.

The superiority of our full pipeline becomes clear when compared against a standard SFT baseline (i.e., Qwen2.5-7B+SFT). The two approaches represent fundamentally different training philosophies: the baseline uses a single-stage, brute-force SFT on all available data (6,104 examples), whereas SeCuRepair employs a two-stage process. SeCuRepair’s initial SFT stage is highly selective, using only 484 high-quality distilled examples to establish a strong reasoning-based foundation before its RL stage trains on the full dataset. The final SeCuRepair model decisively outperforms the monolithic SFT approach, achieving a 19.58% higher CodeBLEU score on BigVul and an 18.79% improvement on PrimeVul_{AVR}. This result strongly validates that our two-stage tuning process is far more effective than simply SFT on all available data in a single step.

5.2.2 Human Evaluation. 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 PrimeVul_{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

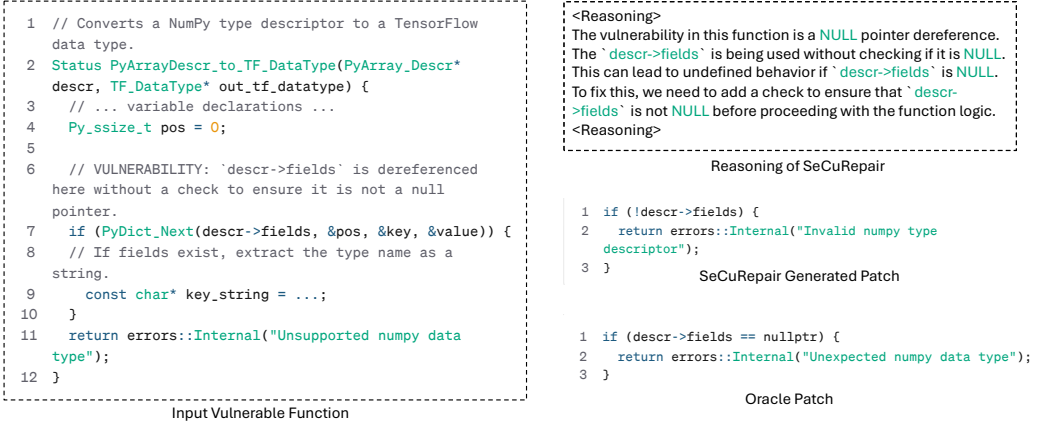


Fig. 5. An example of SeCuRepair performing semantic repair with syntactic different patch with oracle.

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 [29, 40–42], 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 ≥ 3 are regarded as workable, aligning with our goal of providing developers with actionable repair suggestions. Recent work [33] 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.

Results. The human evaluation confirms the superiority of our approach. SeCuRepair produces the highest proportion of workable patches: 58.0% compared to 50.0% for GPT-4o and 20.0% for VulMaster. Specifically, SeCuRepair outperforms the best-performing baseline, GPT-4o, by 16% in the number of workable patches generated, demonstrating that our specialized training pipeline is more effective for AVR than using a raw larger model alone. The success of SeCuRepair demonstrates that the combination of a reason-then-edit paradigm and syntactic- and semantic-aware RL is a key factor in guiding language models to generate functionally correct and logically sound repairs.

5.2.3 Case Analysis. We show one example of SeCuRepair, in which SeCuRepair-generated patch is syntactically different but semantically identical to oracle, in Figure 5. 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 PrimeVul_{AVR} (CVE-2021-29513 [2]), contains a critical NULL pointer dereference vulnerability (CWE-476 [3]). 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 5, 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.

Answer to RQ1: SeCuRepair significantly and consistently outperforms all baselines on both the BigVul and PrimeVul_{AVR} test sets, with improvements of at least 34.52% and 31.52%, respectively ($p < 0.001$). Moreover, our human evaluation of patch correctness shows that SeCuRepair generates 16% more workable patches than the best-performing baseline, GPT-4o.

5.3 RQ2. Contribution of Reasoning

To isolate the impact of our reason-then-edit approach, we conduct two targeted ablation studies:

- (1) **SFT with vs. without Reasoning:** First, we assess whether training on explicit reasoning traces improves patch quality. We compare two SFT models: one trained on our distilled (vulnerable code, reasoning, ground-truth patch) triples, and a baseline model trained only on (vulnerable code, ground-truth patch) pairs. This experiment directly measures the benefit of incorporating reasoning into the SFT stage.
- (2) **RL with vs. without SFT Initialization:** Second, we investigate if 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. This experiment evaluates the contribution of the SFT warm-up to the stability and final performance of the RL fine-tuning.

Table 2. Repair effectiveness of *patch-only* vs *reasoning+patch* supervision on subset of 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 (↑ 6.94%)

5.3.1 SFT with Reasoning Improves Patch Quality. To measure the impact of training on explicit reasoning, we compare two models: SFT (Patch-Only), trained on ground-truth patches, and SFT (Reasoning+Patch), trained on our distilled examples. To ensure a fair comparison and isolate the effect of the reasoning data, the SFT (Patch-Only) model is trained using the ground-truth patches for the exact same subset of vulnerabilities that are distilled for the SFT (Reasoning+Patch) model.

The results in Table 2 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.

5.3.2 SFT Initialization is beneficial for RL. To measure the impact of our SFT stage for RL training, we compare the RL training dynamics of a policy initialized from our SFT (Reasoning+Patch) checkpoint against one starting from the base model. Figure 6 plots the learning curves for both, making the differences in stability and final performance clear.

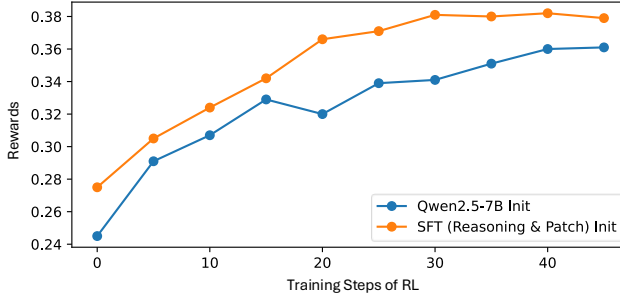


Fig. 6. Training curves of the RL stage between two initialized models: original model v.s. the SFT initialized one. The **x-axis** is RL training steps; the **y-axis** is the per-step, semantic-aware reward computed from the generated patches against the oracle. Steeper slopes indicate faster learning; higher plateaus correlate with better endpoint performance.

The SFT-initialized policy demonstrates superior performance across all three key metrics: 1) **Faster Convergence**: The SFT-initialized policy is significantly more sample-efficient, reaching its best reward by step 30, approximately 33% faster than the non-SFT policy, which requires around 40 steps. 2) **Greater Stability**: The SFT policy’s reward curve rises smoothly with minimal oscillation. In contrast, the non-SFT curve exhibits a noticeable dip between steps 15–20, indicating less stable learning dynamics. 3) **Higher Final Performance**: The SFT-initialized policy converges to a higher reward plateau (around 0.38) compared to the non-SFT policy (around 0.36). This higher final reward directly correlates with better downstream repair performance.

Answer to RQ2: Incorporating reasoning improves SFT patch quality by 6.94% on CodeBLEU. Furthermore, this SFT stage provides a crucial warm-start for RL, leading to more stable and more effective policy optimization with faster coverage.

5.4 RQ3. The Contribution of Syntactic- and Semantic-aware RL

Our semantic-aware RL stage has three key design elements: the RL optimization algorithm (GRPO), the syntactic- and semantics-aware reward, and the curriculum learning schedule. To quantify the contribution of each, we conduct a progressive ablation study, comparing the following model variants:

- **A0: Base Model.** The original, pre-trained Qwen2.5-7B-Instruct without any fine-tuning.
- **A1: SFT with Reasoning.** The base model was fine-tuned on distilled reasoning traces.
- **A2: SFT → Syntactic-aware RL.** We start from our A1 checkpoint and then apply RL, but with a reward function based only on the BLEU score. This measures the benefit of RL optimization itself without the semantic component.
- **A3: Syntactic-aware RL → Syntactic- and Semantic-aware RL.** We enhance the A2 checkpoint by replacing the BLEU-only reward with our full semantics-aware reward function (BLEU + AST + DFG). This isolates the contribution of the semantic reward signal.
- **A4: Full SeCuRepair Model.** Finally, we add the curriculum learning schedule to the A3 checkpoint. This measures the incremental benefit of the curriculum learning schedule.

Results. We show three key insights of the effectiveness of our RL pipeline, as illustrated in Fig 7.

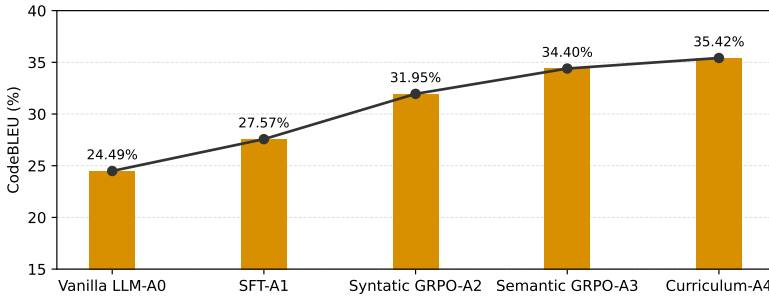


Fig. 7. Step-wise Performance Improvement of SeCuRepair Across Training Stages.

- **GRPO-style RL optimization boost the performance of SeCuRepair under SFT (from A1 to A2).** The RL stage allows the model to learn from its own generated outputs and recover from generating patches far off track, a crucial capability that cannot be learned from imitating a single reference patch via SFT. The RL design achieves a 15.89% improvement over the SFT.
- **Semantic-aware reward improves over a purely syntactic signal (from A2 to A3).** Our composite reward aligns the training objective with functional correctness by incorporating structural (AST) and data-flow (DFG) similarity. This guides the model to learn the underlying behavior of a correct patch, not just its lexical form, leading to 7.67% improvement in CodeBLEU.
- **Difficulty-aware curriculum adds performance boost (from A3 to A4).** Introducing the curriculum learning schedule improves the performance of SeCuRepair by 2.97%. Although the aggregate gain is modest, a per-bucket analysis reveals that the curriculum yields substantial improvements precisely where they are needed most: functions with 2-10 vulnerable regions by 4.24% and functions with >10 vulnerable regions by 9.46% in terms of CodeBLEU.

Answer to RQ3: The ablation study highlights that each stage of SeCuRepair contributes to the final performance. Specifically, GRPO-style RL optimization provides a 15.89% improvement over the SFT adopted in prior works. The semantics-aware reward adds a 7.67% gain, while difficulty-aware curriculum learning enhances the model's ability to fix multi-hunk vulnerabilities by 9.49% for functions with more than 10 vulnerable regions.

6 Discussions

6.1 Discussion of Most Dangerous CWE Performance

To better understand the impact of SeCuRepair, we conducted a fine-grained analysis of model performance on the top-10 most dangerous CWE types [7] within the PrimeVul_{AVR} dataset. Table 3 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.

Table 3. 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

6.2 Threats to Validity

Our study is subject to several potential threats. For internal validity, our reasoning transfer is dependent on the teacher model’s output quality; we mitigate this by employing a strict, two-stage rejection sampling filter to ensure only high-quality examples are used in SFT. Regarding external validity, while our current evaluation is on C/C++, the semantics-aware rewards based on AST and DFG are language-agnostic, suggesting that SeCuRepair could be extended to other languages with proper fine-tuning. Finally, to address construct validity, we acknowledge the limitations of Exact Match as a metric. We mitigate this threat by using the more robust CodeBLEU for automatic evaluation, complemented by a human study to ensure a more holistic assessment of patch quality.

7 Related Work

AVR. In this work, we primarily compare with learning-driven AVR approaches, such as VulMaster [44]. For completeness, we briefly introduce other categories of AVR methods. Following the prior work [26], we also classify AVR approaches into four types: template-guided, search-based, constraint-based, and learning-driven. Template-guided methods rely on predefined templates, which often lack flexibility. For instance, AutoPaG [28] was proposed to repair out-of-bound vulnerabilities. Given a working exploit, it applies fixed templates to patch both out-of-bound reads and writes. When a bound check detects an out-of-bound write, the instrumentation either truncates the write or ignores it silently. Search-based approaches employ heuristic rules to explore and generate candidate patches. For example, VulnFix [43] introduces snapshot fuzzing to infer precise patch invariants by heuristically mutating program states (i.e., snapshots) at specific execution points. Similar to AutoPaG, VulnFix also requires an exploit that triggers a known vulnerability as input. Constraint-based approaches formalize repair as satisfying a set of constraints. ExtractFix [23], for example, aims to mitigate overfitting by extracting constraint specifications directly from an observed vulnerability. Given a vulnerability with an exploit, ExtractFix leverages sanitizers to derive constraints representing the faulty behavior. In contrast to these methods, our proposed SeCuRepair adopts a learning-driven approach that does not rely on exploits as input. Because our evaluation datasets are much larger and no exploits are available, a direct comparison with these methods is not feasible. Furthermore, since SeCuRepair is a learning-based approach, it is not entirely fair to compare it with methods that do not require training.

RL in software engineering. RL has recently shown promise in addressing diverse software engineering challenges [24, 27, 35, 37, 39]. Wang et al.[35] propose RLCoder, which applies an RL framework for repository-level code completion. Liang et al.[27] introduce GARL, which combines a genetic algorithm with RL to efficiently generate diverse, realistic UAV landing failures under practical resource constraints. The most relevant work is RLRep [24], which recommends smart contract repairs using RL. Unlike our approach SeCuRepair, RLRep is tailored specifically for smart

contract vulnerabilities: it employs an LSTM-based encoder–decoder architecture rather than LLMs, and its reward function is designed around vulnerability-specific characteristics of smart contracts. Distinct from all these approaches, SeCuRepair aims to repair general software vulnerabilities.

8 Conclusion and Future Work

In this paper, we first conducted an empirical analysis that identified three issues in modern learning-based Automated Vulnerability Repair approaches: (1) poor cross-repository generalization, (2) a performance collapse on complex, multi-hunk repairs, and (3) brittleness to simple semantic-preserving code changes. These findings reveal that existing models, which are often trained to imitate a single reference patch, fail to learn the underlying semantic principles of a correct fix.

To address these limitations, we introduced SeCuRepair, a novel AVR framework whose components are specifically designed to counteract these identified weaknesses. By combining a reason-then-edit paradigm, a syntactic- and semantics-aware RL reward, and a difficulty-aware curriculum, SeCuRepair moves beyond superficial pattern matching. It learns to reason about the root cause of a vulnerability and generate patches that are structurally and semantically sound.

Our comprehensive evaluation on strict, repository-level test sets demonstrates that SeCuRepair significantly outperforms state-of-the-art baselines. This success underscores the importance of moving beyond syntactic imitation to a training paradigm that aligns with the semantic correctness of a repair. In future work, we aim to broaden the evaluation of SeCuRepair by applying it to vulnerabilities across diverse programming languages.

References

- [1] The common vulnerability scoring system (cvss) and its applicability to federal agency systems. Technical Report NISTIR 7435. (Accessed 07/09/2025).
- [2] Cve-2021-29513. <https://nvd.nist.gov/vuln/detail/CVE-2021-29513>. (Accessed on 12/09/2025).
- [3] Cwe-476. <https://cwe.mitre.org/data/definitions/476.html>. (Accessed on 12/09/2025).
- [4] Edgescan report. <https://www.edgescan.com/january-2022-vulnerability-statistics-snapshot/>. (Accessed 07/09/2025).
- [5] Github repository statistics. <https://coinlaw.io/github-statistics/>. (Accessed 07/09/2025).
- [6] Gpt-5 mini. <https://platform.openai.com/docs/models/gpt-5-mini>. (Accessed 07/09/2025).
- [7] Most dangerous cwes. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html. (Accessed 07/09/2025).
- [8] Nvd dashboard. <https://nvd.nist.gov/general/nvd-dashboard>. (Accessed 07/09/2025).
- [9] Transformers. <https://huggingface.co/docs/transformers/en/index>. (Accessed 07/09/2025).
- [10] Verl. <https://verl.readthedocs.io/en/latest/index.html>. (Accessed 07/09/2025).
- [11] Vllm. <https://docs.vllm.ai/en/latest/usage/index.html>. (Accessed 07/09/2025).
- [12] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- [13] Tushar Aggarwal, Swayam Singh, Abhijeet Awasthi, Aditya Kanade, and Nagarajan Natarajan. Nextcoder: Robust adaptation of code lms to diverse code edits. In *Forty-second International Conference on Machine Learning*, 2025.
- [14] Antonis Antoniadis, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*, 2024.
- [15] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.
- [16] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024.
- [17] Zimin Chen, Steve Kommrusch, and Martin Monperrus. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering*, 49(1):147–165, 2022.
- [18] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv: 2403.18624*, 2024.
- [19] Qingao Dong, Yuanzhang Lin, Hailong Sun, and Xiang Gao. Enhancing automated vulnerability repair through dependency embedding and pattern store. *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 193–204, 2025.

- [20] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 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, 2020.
- [21] Nicole Forsgren, Bas Alberts, Kevin Backhouse, Grey Baker, Greg Cecarelli, Derek Jedamski, Scot Kelly, and Clair Sullivan. 2020 state of the octoverse: Securing the world’s software. *arXiv preprint arXiv: 2110.10246*, 2021.
- [22] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 935–947, 2022.
- [23] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(2):1–27, 2021.
- [24] Hanyang Guo, Yingye Chen, Xiangping Chen, Yuan Huang, and Zibin Zheng. Smart contract code repair recommendation based on reinforcement learning and multi-metric optimization. *ACM Transactions on Software Engineering and Methodology*, 33(4):1–31, 2024.
- [25] Kaiyao Ke. 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, 2025.
- [26] Ying Li, Faysal Hossain Shezan, Bomin Wei, Gang Wang, and Yuan Tian. Sok: Towards effective automated vulnerability repair. *arXiv preprint arXiv:2501.18820*, 2025.
- [27] Linfeng Liang, Yao Deng, Kye Morton, Valtteri Kallinen, Alice James, Avishkar Seth, Endrowednes Kuantama, Subhas Mukhopadhyay, Richard Han, and Xi Zheng. Garl: Genetic algorithm-augmented reinforcement learning to detect violations in marker-based autonomous landing systems. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 613–613. IEEE Computer Society, 2025.
- [28] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. 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, 2007.
- [29] Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. 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, 2023.
- [30] Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 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, 2024.
- [31] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.
- [32] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.
- [33] Wannita Takerngsaksiri, Jirat Pasuksmit, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Ruixiong Zhang, Fan Jiang, Jing Li, Evan Cook, Kun Chen, and Ming Wu. 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, 2025.
- [34] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems*, 37:51963–51993, 2024.
- [35] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. Rlcoder: Reinforcement learning for repository-level code completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 165–177. IEEE Computer Society, 2024.
- [36] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.
- [37] Martin Weyssow, Chengran Yang, Junkai Chen, Ratnadira Widayarsi, Ting Zhang, Huihui Huang, Huu Hung Nguyen, Yan Naing Tun, Tan Bui, Yikun Li, et al. R2vul: Learning to reason about software vulnerabilities with reinforcement learning and structured reasoning distillation. *arXiv preprint arXiv:2504.04699*, 2025.
- [38] Robert F Woolson. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials*, pages 1–3, 2007.
- [39] Chengran Yang, Hong Jin Kang, Jieke Shi, and David Lo. Acecode: A reinforcement learning framework for aligning code efficiency and correctness in code language models. *arXiv preprint arXiv:2412.17264*, 2024.
- [40] Jian Zhang, Shangqing Liu, Xu Wang, Tianlin Li, and Yang Liu. Learning to locate and describe vulnerabilities. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 332–344. IEEE, 2023.
- [41] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1385–1397, 2020.

- [42] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. Learning to handle exceptions. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 29–41, 2020.
- [43] Yuntong Zhang, Xiang Gao, Gregory J Duck, and Abhik Roychoudhury. Program vulnerability repair via inductive inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 691–702, 2022.
- [44] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 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, 2024.