VRPO: Policy Optimization for Functionality-Preserving Automated Vulnerability Repair

Anonymous ACL submission

Abstract

Automated Vulnerability Repair (AVR) has emerged as a practical approach to address the growing number of software vulnerabilities. Recently, Large Language Models (LLMs) have been used extensively for this task due to their superior coding capabilities. Despite being able to generate secure patches, LLMs often alter the original code's correct behavior. To understand this problem, we construct a benchmark consisting of diverse C vulnerable programs, with verifiable security and functionality checks, **VULNFUNC**, using an agentic test case generation pipeline. Evaluating existing work and LLMs reveals their limitations in producing complete, usable patches that are both functionally correct and secure. We also curate a large amount of code corpus to support model training. Based on this dataset, we propose **V**ulnerability **R**epairing **P**olicy **O**ptimization (VRPO), a training paradigm designed to improve LLMs' vulnerability repair capabilities. VRPO uses formal verification and LLM-as-a-Judge to guide the model in writing safe and semantically equivalent patches. Our extensive experiments show that VRPO substantially improves patch quality across model sizes and families. Evaluation in the VULNFUNC benchmark on Qwen3-8B shows the fully fixed rate doubles from 16.55% to 43.74%, with gains in compilation, functionality retention, and vulnerability patching capabilities. Further experiments demonstrate VRPO's effectiveness in addressing real-world vulnerabilities.

1 Introduction

The accelerating pace of software development has led to an increase in security defects. These flaws can severely compromise software systems, causing operational disruptions and unauthorized data access. These incidents result in financial, value, and trust losses for organizations (Shin et al., 2010; Council, 2023), underscoring the urgency of identifying and patching software security bugs. The patching process, including finding and understanding the root cause of the vulnerability and writing a code fix, is a significant burden on developers. These steps are usually performed manually, leading to delays and overlooked threats due to human error (Croft et al., 2023). Therefore, researchers are actively developing Automated Vulnerability Repair (AVR) tools to alleviate these labor-intensive tasks, saving time and resources.

Prior AVR tools leverage templates, heuristic search, or constraints to craft security patches (Kim et al., 2013; Weimer et al., 2009; Mechtaev et al., 2016), but these methods have limitations in generalizing across different types of vulnerabilities and are computationally inefficient. On the other hand, learning-based methods use deep learning models trained on code corpora to generate patches directly from vulnerable code. Recently, LLMs have shown superior capabilities and promising results in vulnerability patching (Pearce et al., 2023; Kulsum et al., 2024; Liu et al., 2024; Huang et al., 2022; Chen et al., 2022; Wen et al., 2025).

Despite advancements, an important aspect that is often overlooked is the patch's usability. AVR tools may generate overfitting patches (Fei et al., 2025) that alter the original code's intended behavior, even when the new code appears to be more secure under the scanners, or incomplete fixes that partially address existing vulnerabilities. Using these patches may introduce new bugs, leave core vulnerabilities open to exploitation, or even bloat the software without improving quality (Fei et al., 2025; Smith et al., 2015). This problem is further hindered by the evaluation and training process of prior work. They rely on text-similarity metrics against ground-truth patches (Huang et al., 2022; Chen et al., 2022), such as BLEU (Papineni et al., 2002), or CodeBLEU (Ren et al., 2020), or even manual review (Nong et al., 2025) to assess patch correctness. From the training perspective,

learning-based methods also optimize for ground-truth code similarity (Fu et al., 2022; Wen et al., 2025). However, similarity to a reference patch is not a good proxy as it neither guarantees that the fix is secure nor that the original functionality is preserved. A nearly perfect-matched patch may be syntactically incorrect or different in terms of logic and semantics, while golden patches themselves may also be incomplete or disrupt existing behavior (Li and Paxson, 2017). Besides, code-similarity metrics do not provide interpretable feedback on the fix's security and functionality. These observations point to two major challenges in building and understanding AVR tools that can reliably reason about and fix vulnerabilities without breaking program correctness: (1) the **evaluation pipeline** should be verifiable in terms of security and functionality, since it would be more trustworthy and explainable compared to a similarity-based or manually reviewed benchmark. To our knowledge, no benchmark satisfies all the requirements, as collecting vulnerable code and crafting security checks and functional test cases are not feasible. (2) A **training pipeline and dataset** that can effectively improve the tool's capability to simultaneously address vulnerabilities without altering the original code's correct behavior is not viable due to data availability and verifiability.

In this paper, we address these challenges by introducing **VULNFUNC**, a benchmark dataset that is security- and functionality-verifiable and diverse in vulnerability categories, resolving difficulty (1). This benchmark reveals that existing AVR methods, as well as LLMs such as GPT-5.4, struggle to generate functionally similar patches. Additionally, we provide a training dataset to improve LLMs' performance on AVR tasks. To address problem (2), we propose **V**ulnerability **R**epairing **P**olicy **O**ptimization (VRPO), a multi-objective RL pipeline that trains LLMs to simultaneously fix vulnerabilities and preserve correct behavior. VRPO leverages formal verification as a rigorous security signal and LLM-as-a-Judge as a functionality signal, enabling precise, semantically grounded reward feedback. Experimental results demonstrate that VRPO substantially improves patch quality across all evaluated dimensions, including compilation rates, security rates, and functionality similarity, across all model sizes and families. For Qwen3-8B, the compilation rate increases from 97.18% to 98.33%, functionality retention increases from 72.43% to 79.19%, and the vulnerability patching rate increases from 46.90% to 83.94%. Overall, the fully fixed rate doubles from 16.55% to 43.74% in the VULNFUNC benchmark. Additionally, VRPO improves the model's ability to produce complete patches for real-world vulnerabilities in the PRIMEVul dataset, from 21.22% to 24.76%, further highlighting its effectiveness.

2 Related Work

Automated Vulnerability Repair (AVR) tools can be broadly categorized into non-learning-based and learning-based approaches (Li et al., 2025). Non-learning methods use patching templates (Lin et al., 2007), search methods (Weimer et al., 2009), and constraint solving (Mechtaev et al., 2016) to find satisfying patches. These methods face difficulties with generalizability across different vulnerability types and codebase sizes. They can also be resource-intensive (Li et al., 2025). On the other hand, learning-based methods employ deep learning models to generate security patches directly from the vulnerable code, treating the task as a code-to-code translation problem. Earlier work explores different methods, such as language model pre-training (Chen et al., 2022; Fu et al., 2022) and fine-tuning (Huang et al., 2022; He et al., 2024) on vulnerable and secure code pairs, model architecture modification (Zhou et al., 2024), prompt engineering (Nong et al., 2024; Pearce et al., 2023; Fei et al., 2025) or by iteratively interacting with the environment (Kulsum et al., 2024). Recently, reasoning models (Wen et al., 2025) also show improvements over baselines and in addressing security bugs. Despite this progress, these methods still fall short of generating satisfiable patches that are simultaneously secure and functionally correct. In this work, we aim to improve LLMs' vulnerability-fixing ability by optimizing toward the two objectives mentioned above rather than relying on ground-truth matching.

3 VULNFUNC Dataset

3.1 Source Data and Validation

We leverage FORMAI-v2 (Tihanyi et al., 2025), a large collection of compilable, independent vulnerable C programs with existing diverse vulnerability classifications, ranging from deference failures to different types of overflows and violations. Each security bug in the dataset is detected using the formal verification tool ESBMC (Menezes et al., 2024). ESBMC (Efficient SMT-Based Model

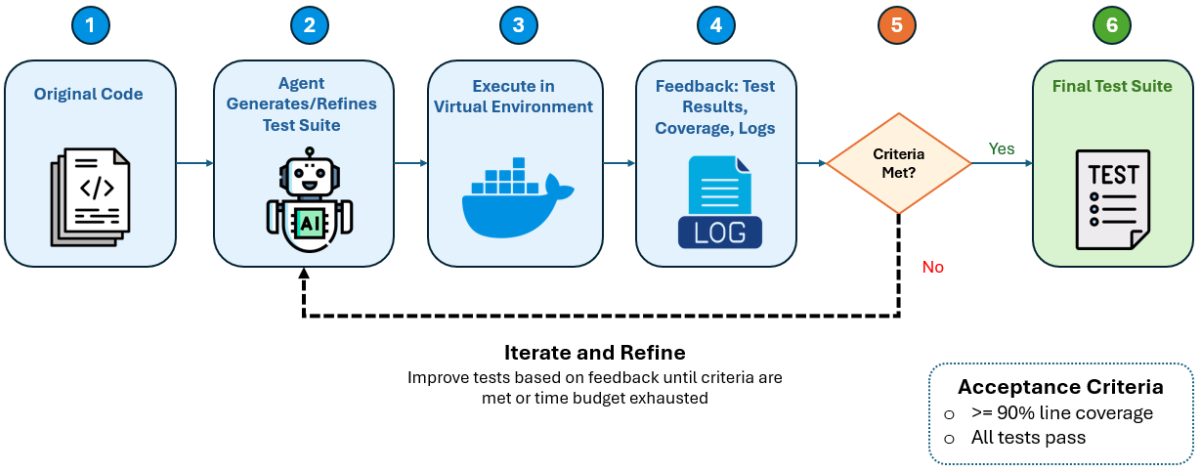


Figure 1: Overview of the automated test generation pipeline. Given the original source code, an agent iteratively generates and refines a test suite by executing in an isolated virtual environment and observing feedback until criteria are satisfied or the time budget is exhausted.

Checker) is a bounded model checker supporting many programming languages that verifies safety properties of programs using SMT solvers. In particular, it checks the program for up to k iterations of each loop and encodes it as a logical formula to detect counterexamples (bounded model checking). As a result, ESBMC is deterministic and verifiable by examining counterexamples. Using this tool, we re-validate the identified vulnerabilities in the FORMAI-v2 dataset, clean the data points, and split the verified vulnerable programs into train and test sets.

3.2 Agentic Test Suite Generation

From the test set, we aim to build a verifiable benchmark incorporating both security and functionality dimensions. Regarding security, we continue to use ESBMC for vulnerability detection as it is verifiable and deterministic. For functionality, we generate functional unit tests to check the correct behavior of the original programs. Given the original code, the generated test suite is required to achieve: (1) at least 90% line coverage and (2) all the tests passed. This threshold ensures exhaustive validation of the original program's correct behaviors and verifies that new patches do not break them. If the new code fails one of the test cases, it means that the patch changes the original code's underlying semantics and logic.

To meet these criteria, we designed an agentic test case generation pipeline that iteratively constructs and refines test suites through interaction with a virtual environment (Fig 1). First, we craft the virtual environment using the prerequisite packages from the FORMAI-v2 repositories, and install the necessary packages for C testing procedures. Then, given the original code file, the agent generates an initial test suite and runs it in the virtual environment, receiving the test results, metrics, and logs. The agent analyzes this feedback to revise the test suite and resubmits it for execution. This loop continues until all requirements are met or the allocated time budget is exhausted, with each iteration progressively improving the test suite's quality and coverage. In our pipeline, we use Claude Sonnet 4.6 (Anthropic, 2026) as the backbone model.

Statistic	Train	Val	Test
# Datapoints	9,949	19	426
# Total vulnerabilities	55,434	56	2,442
Avg. vulnerabilities / program	5.57	2.95	5.73
Avg. test cases / program	—	13.58	17.25
Line coverage (%)	—	99.11	99.04

Table 1: Dataset statistics for VULNFUNC splits.

3.3 Dataset Statistics

Finally, we obtain (1) a benchmark test set of 426 vulnerable programs, averaging 5.73 vulnerability violations, 17.25 unit tests, and 99.04% line coverage per program, (2) a training set of 9,949 samples and a validation set, as in Tab 1. Test coverage reaches 99% on both validation and test sets, well above the 90% requirement, demonstrating the effectiveness of the agent system in generating constrained test suites. Details on vulnerability classifications and the data-cleaning process across all splits are provided in Appx. A.

4 Vulnerability Repairing Policy Optimization

4.1 Problem Formulation

Given a vulnerable program x, let π_θ be an LLM policy that generates a patch $y \sim \pi_\theta(\cdot \mid x, V(x))$, where $V(\cdot)$ is a vulnerability scanner that identifies the vulnerabilities in the input code. The output of $V(\cdot)$ is a set of vulnerabilities, with each item containing information such as vulnerability classification, severity, and location. We further define a functionality checker $F(x, y) \in [0, 1]$ that measures the behavioral similarity between the original code x and the generated patch y. The higher the value of $F(\cdot, \cdot)$, the more identical the two code snippets are in terms of semantics.

Our goal is to find an optimal policy π_θ^* that, given any vulnerable program x, produces a patch $y \sim \pi_\theta^*(\cdot \mid x, V(x))$ simultaneously satisfying two conditions: (1) **security**: the patch resolves all vulnerabilities $V(y) = \emptyset$; and (2) **functionality**: the patch fully preserves the original program behavior, meaning that $F(x, y) = 1$.

4.2 Methodology

Multi-objective Reinforcement Learning. Exactly satisfying both conditions simultaneously is intractable as a constrained optimization problem, since $V(y) = \emptyset$ and $F(x, y) = 1$ are hard constraints over a discrete, combinatorial output space. Therefore, we relax this into a multi-objective reward maximization problem, where the policy is optimized to jointly maximize a security reward r_{sec} and a functionality reward r_{func}:

$$\pi_\theta^* = \arg\max_{\pi_\theta} \mathbb{E}_{x \sim \mathcal{D},\, y \sim \pi_\theta(\cdot \mid x, V(x))} \left[f\left(r_{\text{sec}}(x, y),\, r_{\text{func}}(x, y)\right) \right] \tag{1}$$

where $r(x, y) = f(\cdot, \cdot)$ is a reward function to be defined. We design each reward signal to continuously approximate its corresponding hard constraint, guiding the policy toward patches that are both secure and functionally correct.

Security Reward. The security reward is designed to positively correlate with the number of bugs addressed by the patch relative to the original vulnerable code, reaching its maximum when the patch is fully safe, while being clipped to zero when the patch introduces new vulnerabilities:

$$r_{\text{sec}}(x, y) = \text{clip}\left(\frac{|V(x)| - |V(y)|}{|V(x)|}, 0, 1 \right) \tag{2}$$

Functionality Reward. Since the original code is compilable, the functionality reward firstly penalizes non-compilable patches and gives them zero scores. Otherwise, the reward is the behavioral similarity between the original and patched code:

$$r_{\text{func}}(x, y) = \begin{cases} 0 & \text{if } y \text{ is not compilable} \\ F(x, y) & \text{if } y \text{ is compilable} \end{cases} \tag{3}$$

During the training process, $F(x, y)$ is computed via an LLM-as-a-Judge that; thus, provides a computationally efficient yet reliable approximation of functional similarity. To boost the reliability of the judging process, we use rubric-based grading (Liu et al., 2023b) and randomly swap the positions of the before/after patches to mitigate position bias (Zheng et al., 2023). We also explore different methods for checking functional similarity, discussed in Sec. 7.

Final Reward and Optimization Objective. The combined reward $r(x, y)$ is then defined as:

$$r(x, y) = \lambda \cdot r_{sec}(x, y) + (1 - \lambda) \cdot r_{func}(x, y) \tag{4}$$

where $\lambda \in (0, 1]$ varies the concentration in producing more secure or functional patches. We use DAPO (Yu et al., 2025a) Clip-Higher's training objective to maximize our defined reward function:

$$\mathcal{J}_{\text{VRPO}}(\theta) = \mathbb{E}_{\substack{x \sim \mathcal{D}, \\ \{y_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid x, V(x))}} \left[\frac{1}{\sum_{i=1}^{G} |y_i|} \sum_{i=1}^{G} \sum_{t=1}^{|o_i|} \min\left(r_{i,t}(\theta)\, \hat{A}_{i,t},\ \text{clip}(r_{i,t}(\theta),\, 1 - \varepsilon_{\text{low}},\, 1 + \varepsilon_{\text{high}})\, \hat{A}_{i,t} \right) \right] \tag{5}$$

where the importance sampling $r_{i,t}(\theta) = \frac{\pi_\theta(y_{i,t} \mid x, V(x), y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t} \mid x, V(x), y_{i,<t})}$ and the group-based advantage is $\hat{A}_{i,t} = \frac{R_i - \text{mean}(\{R_i\}_{i=1}^{G})}{\text{std}(\{R_i\}_{i=1}^{G})}$.

5 Experiments Setup

5.1 Models and Datasets

Configurations. We experimented VRPO on Qwen 3 models (Team, 2025) such as Qwen3-1.7B, Qwen3-4B, Qwen3-8B, and Llama-3.2-3B (Grattafiori et al., 2024) with the hyperparameters discussed in Appx B. The main results are reported using VRPO with $\lambda = 0.5$ unless further specified.

Baselines. We compare our methods again different baselines including SafeCoder (He et al., 2024), VulMaster (Zhou et al., 2024), and VulRepair (Fu et al., 2022). SafeCoder uses a masked

token log-likelihood loss on secure code tokens and masked unlikelihood loss on vulnerable tokens to guide the LLM to generate benign programs. VulRepair uses a pretrained CodeT5 model and fine-tunes it on a large dataset of vulnerable/secure code pairs to patch vulnerabilities. VulMaster further enhances performance by turning the vulnerable code, AST graph, vulnerability's CWE information, and its fixing strategy into embeddings using corresponding encoders, combining them into a unified embedding, and feeding it to a decoder to generate the secure code. Additionally, we evaluate the performance of base models before applying VRPO training, as well as OpenAI's GPT-5.4 (OpenAI, 2026). We set the reasoning effort of this model to "medium" during the evaluation. While we are aware that Vul-R2 (Wen et al., 2025) closely aligns with our method, the authors do not release the model weights; thus, we cannot include it as a baseline.

Datasets. We evaluate all methods on the test set of the VULNFUNC dataset. Additionally, we test VRPO and baselines on the real-world PRIMEVul dataset (Ding et al., 2024). PRIMEVul is a comprehensive, highly accurate benchmark for vulnerability detection that covers security bugs from public repositories such as NVD, Bugzilla, and Debian trackers. These vulnerabilities are then deduplicated, automatically labeled, and split into vulnerable and benign code snippet pairs. We use the PRIMEVul paired test set and select only C programming language snippets, resulting in a benchmark of 245 data points.

5.2 Evaluation Metrics

For VULNFUNC, we use the following four metrics: (1) **Compilation Rate** (COMP%) measures the percentage of generated patches that are syntactically valid and successfully compile. (2) **Functional Correctness** (FUNC%) measures the percentage of unit tests that the compilable patches passed in the test suites, showing the functional similarity between the before and after patched code. (3) **Security Rate** (SEC%) measures the percentage of vulnerabilities addressed, conditioned on fully functional and compilable patches. (4) **Fully Fixed Rate** (FIX%) is the end-to-end metric, measuring the percentage of the complete patches. A complete patch is a patch that is compilable, functionally correct, and fully secure, satisfying both $F(x, y) = 1$ and $V(y) = \emptyset$. This is the primary metric for overall patch quality.

	COMP %↑	FUNC %↑	SEC %↑	FIX %↑
Qwen3-1.7B family				
Base	89.49%	**83.07%**	19.98%	7.92%
VRPO	**97.57%**	80.43%	**65.64%**	**31.77%**
Qwen3-4B family				
Base	95.70%	74.75%	38.84%	17.06%
VRPO	**98.36%**	**80.57%**	**76.20%**	**37.87%**
Qwen3-8B family				
Base	97.18%	72.43%	46.90%	16.55%
VRPO	**98.83%**	**79.19%**	**83.94%**	**43.74%**
Llama3.2-3B family				
Base	83.26%	66.08%	29.05%	6.89%
VRPO	**99.45%**	**81.47%**	**79.93%**	**46.71%**
Baselines				
SafeCoder	59.31%	85.18%	2.60%	1.06%
VulRepair	0%	0%	0%	0%
VulMaster	1.17%	**85.71%**	11.43%	0%
GPT-5.4 (reasoning medium)	**94.1%**	25.8%	**87.1%**	**8.45%**

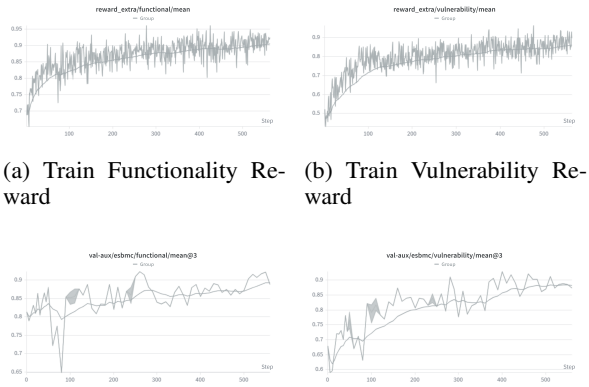
Table 2: VULNFUNC benchmark results across model families and baselines.

In PRIMEVul, as the code snippets cannot be directly compiled and run, we use text-similarity metrics against the ground truth, as in the literature (Zhou et al., 2024; Fu et al., 2022; Wen et al., 2025). Specifically, we use BLEU-4 (Papineni et al., 2002), and CodeBLEU (Ren et al., 2020). Additionally, since these metrics do not explicitly reveal the patch's security and functionality, we use LLM-as-a-Judge to provide functionality and security (FUNC% and SEC%) feedback, using benign code as an anchor. From that, we determine the Fully Fixed Rate FIX%. We use Qwen3-30B-A3B-Instruct-2507 (Team, 2025) as the Judge model due to its performance and efficiency.

All the mentioned metrics are reported as metric@1, averaged over $k = 3$ independent samples per task, following DeepSeek-R1's pass@1 (Guo et al., 2025).

6 Result Analysis

RQ1: How well do baselines and base LLMs generate security patches in VULNFUNC? The results in Tab 2 reveal that the baselines consistently generate patches with severe functionality deviations. First, they struggle to produce compilable code: many patches are syntactically erroneous, especially those from VulRepair and VulMaster, with 0% and 1.17%, respectively. LLMs are significantly more likely to generate usable patches, thanks to their code comprehension and producing abilities: most models' compilable rates are above 80%, except SafeCoder, which is only 59.31%. Second, among the patches that do compile, functional correctness degrades further, indi-



(a) Train Functionality Reward

(b) Train Vulnerability Reward

(c) Validation Functionality Reward

(d) Validation Vulnerability Reward

Figure 2: Reward curves during training of the Qwen3-8B. Train and validation rewards for functionality and vulnerability objectives are shown across training steps.

```
--- original_vulnerable.c
+++ patched_secure.c
@@ -1,4 +1,7 @@
-// File: gpt4o_mini-11075.c
+// Corrected version of the code addressing all identified violations
+// Violation 1: Buffer overflow on scanf was fixed by adding width specifiers
+// Violation 2: generate_dna_sequence had a potential buffer overflow due to null-termination beyond array bounds
+
 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>
@@ -23,7 +26,7 @@

     printf("Welcome to the Genome Sequencing Simulator!\n");
     printf("Please enter the number of sequences to generate (max %d): ", MAX_SEQUENCE);
-    scanf("%d", &sequence_count);
+    scanf("%10d", &sequence_count); // Added width specifier to prevent buffer overflow

     if (sequence_count < 1 || sequence_count > MAX_SEQUENCE) {
         fprintf(stderr, "Invalid number of sequences! Exiting...\n");
@@ -38,7 +41,7 @@

         // Ask for mutations
         printf("Enter number of mutations for sequence %d: ", i + 1);
-        scanf("%d", &mutations);
+        scanf("%10d", &mutations); // Added width specifier to prevent buffer overflow

         if (mutations < 0) {
             fprintf(stderr, "Number of mutations cannot be negative! Skipping mutations for this sequence.\n");
@@ -60,7 +63,9 @@
     for (int i = 0; i < length; i++) {
         sequence[i] = BASES[rand() % 4]; // Randomly select a base from A, C, G, T
     }
-    sequence[length] = '\0'; // Null terminate the sequence string
+    if (length > 0) {
+        sequence[length] = '\0'; // Null terminate the sequence string
+    }
 }
```

Figure 3: Differences between the original and patched file for the project "gpt4o_mini-11075.c". The generated patch fully resolved 2 violations and passed all of the 19 test cases.

cating that they break the intended program behavior. Surprisingly, a strong model such as GPT-5.4 exhibits a noticeably low functionality score of 25.8%. We investigated the model's outputs and found a consistent pattern: it rewrites the program too aggressively. Specifically, rather than producing a minimal, targeted fix, the model introduces regressions by rewriting surrounding code structure, altering function signatures, marking functions as "static", or omitting headers (e.g., <stdint.h>) required by the test harness. While the generated code often compiles in isolation, it breaks harness-level linking by violating the expected API surface. Noticeably, baselines exhibit limited vulnerability-fixing capability. VulRepair produces no compilable code, thus a zero security score. SafeCoder and VulMaster achieved only 2.6% and 11.43%, falling short of base LLMs. These models range from 19.98% for Qwen3-1.7B to 46.0% for Qwen3-8B, which is still low. GPT-5.4 shows superior ability to generate benign code with a score of 87.1%.

In conclusion, functional deviation and poor fixing ability are reflected in the low complete patch rate across all methods. VulMaster and VulRepair produce no usable codes, whereas SafeCoder achieves 1.8% FIX%. Due to the extreme functionality of the steering, GPT-5.4 only generates 8.45% complete patches. Llama and Qwen models can only resolve a maximum of 17.06% of the benchmark, showing improvement headroom.

RQ2: Can VRPO enhance both vulnerability patching and functionality retention effectiveness? Tab. 2 shows that VRPO yields substantial gains in the security metric across all evaluated model families, with improvements at least twice as large before applying VRPO. Interestingly, we observe a gain of over 50 percentage points on Llama3.2-3B, outperforming similar size Qwen3-4B model. Our inspection shows that Llama3.2-3B does not overthink as much as Qwen3 models; therefore, Llama's generated patches are more efficient and more likely to be correct.

We investigate the vulnerabilities addressed by the Qwen3-8B base and the VRPO model in Tab 3. Overall, the VRPO model fixes 2,151 out of 2,442 vulnerabilities, compared to 1,626 for the base model, representing a 32.3% relative improvement. This advantage is consistent across nearly all violation categories, demonstrating that VRPO learns broadly generalizable fixing behavior rather than overfitting to a narrow class of bugs. An example of the patch generated by VRPO-8B is shown in the Fig 3. The project is a genome-sequence simulator that requires user input. The two vulnerabilities are buffer overflows in the inputs and the "sequence" array. The patch correctly shows the locations of the vulnerabilities, adds a minimal modification that successfully resolves those bugs, as well as an explanatory comment. These behaviors are discovered and reinforced throughout the training process by the policy itself.

Additionally, VRPO improves both the compilation rate and the functionality retention rate across all trained models. Qwen3-1.7B shows a better functionality score compared to its corresponding VRPO model, but suffers from a lower compilation

Violation Type	Total	Base	VRPO
Dereference Failures			
NULL pointer	551	502	**526**
Invalid pointer	107	31	**61**
Forgotten memory	44	21	**34**
Array bounds violated (general)	59	35	**56**
Invalidated dynamic object	18	14	**15**
Access to object out of bounds	15	7	**12**
Accessed expired variable pointer	2	0	0
Free of non-dynamic memory	1	1	1
IBTA	1	1	1
Arithmetic Overflows			
On `sub`	114	72	**98**
On `add`	102	57	**82**
On `mul`	91	46	**80**
Floating-point `ieee_mul`	91	36	**54**
Floating-point `ieee_add`	50	17	**27**
Floating-point `ieee_div`	49	25	23
Floating-point `ieee_sub`	22	5	**7**
On `div`	24	8	**20**
On `shl`	27	17	**24**
On `modulus`	15	11	11
On `neg`	1	1	0
Buffer Overflows			
On `scanf`	709	316	**631**
On `fscanf`	36	14	**21**
On `sscanf`	28	12	**23**
Array Bounds Violations			
Upper bound	104	103	103
Lower bound	64	62	**63**
VLA array size overflows	40	39	**40**
Miscellaneous Vulnerabilities			
Division by zero	57	52	**54**
Pointer to file must be valid	20	19	**20**
Total	**2,442**	**1,626**	**2,151**

Table 3: Per-violation-type breakdown of resolved vulnerabilities. **Total** is the benchmark count; **Base** and **VRPO** report the number resolved by Qwen3-8B model Base and VRPO.

rate. By combining the compilation process and the LLM-as-a-Judge paradigm as the functionality reward during training, VRPO encourages models to produce patches that maintain the original program's intended behavior. Thanks to both vulnerability patching and code-semantic preserving abilities, VRPO significantly improves fully fix rate over the base model and baselines, demonstrating its effectiveness in helping LLMs generate correct and complete patches.

RQ3: What are the effects of changing λ in the reward formula? We analyze the role of the hyperparameter λ in the reward function, where a

	COMP %↑	FUNC %↑	SEC %↑	FIX %↑
Base-8B	95.70%	74.75%	38.84%	17.06%
VRPO-8B				
λ-0.1	99.30%	**84.07%**	64.21%	33.49%
λ-0.3	97.10%	81.47%	76.71%	39.91%
λ-0.5	98.36%	80.57%	76.20%	37.87%
λ-0.7	99.06%	78.28%	**80.33%**	**41.16%**
λ-1	97.42%	54.79%	79.50%	27.23%
VRPO-8B-Lag	**99.53%**	81.60%	73.84%	37.32%

Table 4: Effect of security-functionality trade-off coefficient λ on VRPO-trained Qwen3-8B. Higher λ upweights the security reward and vice versa.

	BLEU-4↑	CodeBLEU↑	FUNC %↑	SEC %↑	FIX %↑
Qwen-8B	0.7606	0.8097	69.74%	67.16%	21.22%
VRPO-8B	0.6696	0.7904	74.31%	78.26%	24.76%
VulRepair	0.6778	0.7151	0.73%	1.22%	0.00%
VulMaster	**0.7674**	**0.8198**	2.20%	3.35%	0.41%
SafeCoder	0.1007	0. 2662	5.66%	5.47%	0.14%
GPT-5.4	0.5500	0.6868	**86.53%**	**96.24%**	**49.39%**

Table 5: PRIMEVul benchmark results across model families and baselines.

large λ highly rewards secure code, and a lower λ encourages the model to maintain functionality. We train different Qwen3-4B models varying λ and observe the results. Tab 4 clearly demonstrates the trade-off: as λ increases from 0.1 to 1.0, the model achieves progressively higher vulnerability repair rates from 64.21% to 80.33%, but at the cost of reduced functional correctness, from 84.07% to 54.79%. Notably, the λ = 1 functionality score is significantly lower than that of other baselines, underscoring the LLM-as-a-Judge's role in guiding the policy to maintain the intended behaviors. We also observe that for $\lambda = 0.1$, despite only a small amount of security reward is given to the policy training, the model's vulnerability-fixing improves significantly, from 38.84% to 64.21%. Overall, $\lambda = 0.7$ is the most performant, with a 41.16% complete patch rate, contradicting our hypothesis that $\lambda = 0.5$ would be the best setting due to the balance between functionality and security.

Since λ is an important hyperparameter in VRPO, we aim to mitigate the hassle of hyperparameter sweeping. We study a dual-optimization framework as in (Dai et al., 2023), VRPO-Lag, that automatically updates λ during training. The detailed formulation of the optimization objective and the update rules for this method are provided in Appx. C. We received mixed results, as shown in Tab. 4. Overall, VRPO-Lag can generate more complete and secure patches than the base model. Compared to other VRPO baselines, it achieved a high rate of compilation and functionality retention while addressing fewer vulnerabilities. The main drawback we observed is its instability during training. Specifically, the model is overly verbose: the number of output tokens often exceeds the maximum output length (10k) despite the prompt length, and the clip ratio can reach 1.0, meaning every update gets clipped. Attempts such as reducing the model and λ learning rates or adding a KL term failed to alleviate this problem. We worked around by adding a length penalty to the reward,

but the training process remained volatile. Therefore, more study on this approach is needed.

RQ4: Can VRPO produce usable patches against real-world vulnerabilities? To shed light on VRPO's capabilities for addressing real-world vulnerabilities, we evaluate the method on the PRIMEVul dataset and compare it with baselines. The result Tab 5 shows that VRPO improves Qwen3-8B's performance in all important metrics: FUNC% improves from 69.74% to 74.31%, SEC% from 67.16% to 78.26%, and the complete patch generation rate increases from 21.22% to 24.76%. This highlights that VRPO improves the model's patch quality, both in terms of security and functionality preservation. Across the approaches, GPT-5.4 generates the most usable code (49.39%), with extremely high functionality and security scores.

The results also show that text-similarity metrics reveal little about the patch's usability. In particular, despite achieving high BLEU-4 and CodeBLEU scores, VulMaster and VulRepair's patch quality is extremely low compared to other methods such as Qwen-8B, VRPO, and especially GPT-5.4. The outputs generated are usually syntactically incorrect, resulting in fewer usable patches.

7 Discussions

What makes VRPO work? We attribute this improvement to VRPO's multi-objective design and the verifiable reward signals. Formal verification provides reliable, deterministic feedback on patch security, while LLM-as-a-Judge provides a useful approximation of functional similarity, guiding policy learning fixes that avoid regression. This observation is further evidenced in the training and validation reward curves in Fig. 2: as training progresses, both the security and functionality curves increase. These separate signals are also interpretable, enabling practitioners to understand the model's fixing properties dynamics during training. Leveraging this, we detected and addressed a reward-hacking attempt during training; more details are provided in Appx. D.

Vulnerability and Functionality Checker. We use ESBMC as our vulnerability scanner, though alternative tools such as static analyzers or LLM-as-a-Judge are compatible with our framework. ESBMC is chosen for its bounded model checking, which provides stronger correctness guarantees than pattern-matching static analyzers. Meanwhile, it remains practically efficient and deterministic, making it more suitable for benchmarking than LLM judgments. We leave the exploration of alternative verification backends to future work.

	COMP %↑	FUNC %↑	SEC %↑	FIX %↑
Base-8B	95.3%	**98.1%**	2.2%	1.9%
VRPO-8B	44.1%	77.7%	78.5%	17.8%

Table 6: VULNFUNC benchmark results for Codex agent across models.

We also consider the functionality similarity checker to be a code clone detector (Roy et al., 2009), where a higher clone score indicates greater semantic similarity. Since patches may differ in code style, variable naming, or implementation logic, we view this as a Type 4 Code Clone. We could not reproduce TAILOR (Liu et al., 2023a) because its codebase is outdated, and MRT-OAST (Yu et al., 2025b) does not provide a script to precompute the AST that the model can use. The only method we can run is DSFM (Xu et al., 2024), but we did not obtain usable results on unseen C programs from the benchmark the authors used.

Agentic System. We evaluate models' agentic capabilities on VULNFUNC by deploying them as backbones within the Codex framework (OpenAI, 2026). For each task, the agent receives the vulnerable source file, its corresponding test suite, and access to a vulnerability scanner. As shown in Tab. 6, VRPO-8B substantially improves the model's ability to produce complete patches compared to the base model, highlighting the importance of our training procedure. However, both models underperform their single-shot counterparts, as neither has been trained for agentic interaction; thus, they struggle to write tool commands and reason from feedback. We expect to see further performance gains as model sizes increase as their agentic capabilities improve.

8 Conclusions

In this paper, we construct VULNFUNC, a benchmark that contains verifiable security and functionality checks. Using this benchmark, we show that prior work and LLMs struggle to generate complete, usable patches. Based on a curated train dataset, we propose VRPO, a multi-objective RL training framework that leverages formal verification and LLM-as-a-Judge to significantly improve LLMs' capabilities to address software vulnerabilities while maintaining the correctness of generated patches, thereby enhancing overall AVR quality.

Limitations

Our dataset and evaluation are limited to C programs but not in other programming languages. Extending VRPO to cross-language settings remains an important direction for future work.

Due to the high cost of generating unit tests for the full training set, we rely on an LLM-as-a-Judge to approximate functional similarity during the RL training phase. While we take steps to improve its reliability through rubric-grading and position-swap calibration, LLM judges are not perfect proxies for ground-truth test execution, and may introduce noise into the training signal.

Our experiments are conducted on models up to 8B parameters due to computational constraints. The behavior of VRPO on larger models remains unexplored, and it is unclear whether the same training dynamics and reward trade-offs would hold at a greater scale.

Ethical Considerations

The paper introduces VULNFUNC, a highly granular, verifiable collection of C programs mapped explicitly to their exact security flaws. While the dataset is used to train better repair models, this could be used by malicious actors to identify precise patterns of code vulnerabilities or to benchmark automated exploit-generation tools. On the other hand, VRPO models are optimized based on ESBMC outputs, which may overlook specific classes of vulnerabilities. Therefore, the generated code should be verified under rigorous complementary security audits and human-in-the-loop validation before deploying the repaired code into production environments.

References

Anthropic. 2026. Claude sonnet 4.6 system card. Technical report, Anthropic PBC. Accessed: May 20, 2026.

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.

Forbes Technology Council. 2023. Zero-day vulnerabilities: 17 consequences and complications. Accessed: May 26, 2026.

Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 121–133. IEEE.

Josef Dai, Xuehai Pan, Ruiyang Sun, Jiaming Ji, Xinbo Xu, Mickel Liu, Yizhou Wang, and Yaodong Yang. 2023. Safe rlhf: Safe reinforcement learning from human feedback. *arXiv preprint arXiv:2310.12773*.

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*.

Zhiwei Fei, Jidong Ge, Chuanyi Li, Tianqi Wang, Yuning Li, Haodong Zhang, LiGuo Huang, and Bin Luo. 2025. Patch correctness assessment: A survey. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–50.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. 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*, ESEC/FSE 2022, page 935–947, New York, NY, USA. Association for Computing Machinery.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

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*.

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

Kai Huang, Su Yang, Hongyu Sun, Chengyi Sun, Xuejun Li, and Yuqing Zhang. 2022. Repairing security vulnerabilities using pre-trained programming language models. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 111–116. IEEE.

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th international conference on software engineering (ICSE)*, pages 802–811. IEEE.

Ummay Kulsum, Haotian Zhu, Bowen Xu, and Marcelo d'Amorim. 2024. A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pages 103–111.

Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215.

Ying Li, Faysal Hossain Shezan, Bomin Wei, Gang Wang, and Yuan Tian. 2025. Sok: Towards effective automated vulnerability repair. *arXiv preprint arXiv:2501.18820*.

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.

Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023a. Learning graph-based code representations for source-level functional similarity detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 345–357.

Peng Liu, He Wang, Chen Zheng, and Yuqing Zhang. 2024. Prompt fix: Vulnerability automatic repair technology based on prompt engineering. In *2024 International Conference on Computing, Networking and Communications (ICNC)*, pages 116–120. IEEE Computer Society.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023b. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 2511–2522.

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701.

Rafael Menezes, Mohannad Aldughaim, Bruno Farias, Xianzhiyu Li, Edoardo Manino, Fedor Shmarov, Kunjian Song, Franz Brauße, Mikhail R. Gadelha, Norbert Tihanyi, Konstantin Korovin, and Lucas C. Cordeiro. 2024. ESBMC 7.4: Harnessing the Power of Intervals. In 30th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'24)*, volume 14572 of *Lecture Notes in Computer Science*, page 376–380. Springer.

Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. 2024. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*.

Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. {APPATCH}: Automated adaptive prompting large language models for {real-world} software vulnerability patching. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4481–4500.

OpenAI. 2026. Codex: AI coding partner from OpenAI. https://openai.com/codex/. Accessed: 2026-05-25.

OpenAI. 2026. Gpt-5.4 thinking system card. Technical report, OpenAI.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297.

Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering*, 37(6):772–787.

Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 532–543.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C. Cordeiro. 2025. How secure is ai-generated code: a large-scale comparison of large language models. *Empirical Software Engineering*, 30(47).

Norbert Tihányi, Tamás Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C. Cordeiro, and Vasileios Mavroeidis. 2023. Formai dataset (v1 v2) – github repository. https://github.com/FormAI-Dataset. Release v2 contains 331,000 compilable C programs with vulnerability classification.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374. IEEE.

Xin-Cheng Wen, Zirui Lin, Yijun Yang, Cuiyun Gao, and Deheng Ye. 2025. Vul-r2: A reasoning llm for automated vulnerability repair. *arXiv preprint arXiv:2510.05480*.

Zhiwei Xu, Shaohua Qiang, Dinghong Song, Min Zhou, Hai Wan, Xibin Zhao, Ping Luo, and Hongyu Zhang. 2024. Dsfm: Enhancing functional code clone detection with deep subtree interactions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2025a. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*.

Tianchen Yu, Li Yuan, Liannan Lin, and Hongkui He. 2025b. A multiple representation transformer with optimized abstract syntax tree for efficient code clone detection. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 281–293. IEEE.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623.

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*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

A Dataset Information and Statistics

A.1 Overall Statistics

We provide statistics for VULNFUNC dataset splits on Tab 1 and vulnerability distribution on Tab 7.

A.2 Dataset Curation and Filtering

We leverage FORMAI-v2 (Tihanyi et al., 2025), a large collection of compilable and independent C programs with vulnerability classification. The dataset license is MIT, which is free to use, and includes 331,000 compilable C programs of which 192,757 are vulnerable against 33 bug types. These programs were generated by LLMs using a dynamic zero-shot prompting technique and comprise programs with varying levels of complexity, consisting of different tasks and tones. Each data point is labeled according to the vulnerabilities present in the code using a formal verification method based on the Efficient SMT-based Bounded Model Checker (ESBMC) (Menezes et al., 2024). ESBMC works by translating a program into logical constraints and using an SMT solver to check whether a property violation is satisfiable; when such constraints are satisfiable, ESBMC reports a bug with a concrete counterexample, which reduces false positives and minimizes false negatives by performing exhaustive verification within a specified threshold. As a result, ESBMC provides a deterministic and verifiable on the existing bugs, making it a robust learning signal for RL training.

From the existing FORMAI-v2 dataset, we rescan for vulnerabilities using ESBMC. We relaxed the ESBMC scanning settings to reduce the vulnerability detection time, as (1) a single scan on the full dataset takes days and (2) Reinforcement Learning involves generating many training samples, and keeping the scanning settings as in (Tihanyi et al., 2025) makes the training unrealistic. In particular, we do not use k-induction and reduce the timeout for each check to 30 seconds as in the author's GitHub repository (Tihányi et al., 2023). After the rerun, we identified 156,538 vulnerable files, totaling 622,049 vulnerabilities. Our analysis indicates that all violation types persist in the new dataset, and the proportions of each type remain comparable to those in the original dataset.

To enable efficient RL training, we constructed two stratified subsets of 5,000 and 10,000 datapoints from the 156,538 vulnerable files, as training on the complete dataset is computationally impractical since RL requires generating numerous samples during training. Creating these stratified datasets presented three critical challenges: (1) extreme class imbalance, with the most common vulnerability type (NULL pointer) accounting for 265,570 instances while the rarest contain fewer than 10 instances; (2) multi-label complexity, as individual files can contain multiple vulnerability types simultaneously, preventing direct application of standard stratification techniques.

We addressed these challenges through a three-tier classification strategy combined with two-phase sampling. We classify vulnerability types

Vulnerability Type	Train	Val	Test	Total
Dereference Failures				
NULL pointer	17,508 (31.69%)	19 (33.93%)	551 (22.56%)	18,078 (31.31%)
Invalid pointer	2,896 (5.24%)	3 (5.36%)	107 (4.38%)	3,006 (5.21%)
Forgotten memory	1,471 (2.66%)	0 (0.00%)	44 (1.80%)	1,515 (2.62%)
Array bounds violated (general)	1,620 (2.93%)	3 (5.36%)	59 (2.42%)	1,682 (2.91%)
Invalidated dynamic object	436 (0.79%)	0 (0.00%)	18 (0.74%)	454 (0.79%)
Access to object out of bounds	419 (0.76%)	0 (0.00%)	15 (0.61%)	434 (0.75%)
Accessed expired variable pointer	58 (0.10%)	0 (0.00%)	2 (0.08%)	60 (0.10%)
Write access to string constant	30 (0.05%)	0 (0.00%)	0 (0.00%)	30 (0.05%)
Free of non-dynamic memory	44 (0.08%)	0 (0.00%)	1 (0.04%)	45 (0.08%)
Oversized field offset	28 (0.05%)	0 (0.00%)	0 (0.00%)	28 (0.05%)
IBTA	30 (0.05%)	0 (0.00%)	1 (0.04%)	31 (0.05%)
Data object accessed with code type	9 (0.02%)	0 (0.00%)	0 (0.00%)	9 (0.02%)
Arithmetic Overflows				
Arithmetic overflow on sub	1,983 (3.59%)	2 (3.57%)	114 (4.67%)	2,099 (3.64%)
Arithmetic overflow on add	1,988 (3.60%)	0 (0.00%)	102 (4.18%)	2,090 (3.62%)
Arithmetic overflow on mul	1,701 (3.08%)	0 (0.00%)	91 (3.73%)	1,792 (3.10%)
Arith. overflow on float `ieee_mul`	1,022 (1.85%)	0 (0.00%)	91 (3.73%)	1,113 (1.93%)
Arith. overflow on float `ieee_add`	618 (1.12%)	0 (0.00%)	50 (2.05%)	668 (1.16%)
Arith. overflow on float `ieee_div`	578 (1.05%)	0 (0.00%)	49 (2.01%)	627 (1.09%)
Arith. overflow on float `ieee_sub`	492 (0.89%)	0 (0.00%)	22 (0.90%)	514 (0.89%)
Arithmetic overflow on div	364 (0.66%)	0 (0.00%)	24 (0.98%)	388 (0.67%)
Arithmetic overflow on shl	307 (0.56%)	0 (0.00%)	27 (1.11%)	334 (0.58%)
Arithmetic overflow on modulus	260 (0.47%)	0 (0.00%)	15 (0.61%)	275 (0.48%)
Arithmetic overflow on neg	48 (0.09%)	0 (0.00%)	1 (0.04%)	49 (0.08%)
Buffer Overflows				
Buffer overflow on `scanf`	15,723 (28.46%)	27 (48.21%)	709 (29.03%)	16,459 (28.50%)
Buffer overflow on `fscanf`	627 (1.14%)	1 (1.79%)	36 (1.47%)	664 (1.15%)
Buffer overflow on `sscanf`	494 (0.89%)	1 (1.79%)	28 (1.15%)	523 (0.91%)
Array Bounds Violations				
Array upper bound	1,449 (2.62%)	0 (0.00%)	104 (4.26%)	1,553 (2.69%)
Array lower bound	1,187 (2.15%)	0 (0.00%)	64 (2.62%)	1,251 (2.17%)
VLA array size overflows	713 (1.29%)	0 (0.00%)	40 (1.64%)	753 (1.30%)
Miscellaneous Vulnerabilities				
Division by zero	826 (1.50%)	0 (0.00%)	57 (2.33%)	883 (1.53%)
Pointer to file must be valid	262 (0.47%)	0 (0.00%)	20 (0.82%)	282 (0.49%)
Same object violation	33 (0.06%)	0 (0.00%)	0 (0.00%)	33 (0.06%)
ZOFO	24 (0.04%)	0 (0.00%)	0 (0.00%)	24 (0.04%)
Total	**55,248 (100%)**	**56 (100%)**	**2,442 (100%)**	**57,746 (100%)**

Table 7: Distribution of vulnerability types across dataset splits with percentages.

into three tiers based on their file counts and projected representation: minority (fewer than 200 files), near-minority (at least 200 files but whose proportional allocation would drop below 200 in the target dataset size), and majority (sufficient files to maintain adequate representation under proportional sampling). To handle minority types, we apply oversampling via random sampling with replacement to ensure each receives 10-15 minimum samples, capped at three times of the original count to avoid extreme duplication. Near-minority types are protected by guaranteeing them at least 200 samples to prevent them from falling below the threshold. The majority types are sampled proportionally from the remaining budget after allocating samples to minority and near-minority classes. The two-phase sampling algorithm first performs seed sampling to ensure at least one file per vulnerability type, prioritizing files with exclusive vulnerabilities to minimize multi-label overlap. It then applies stratified filling according to the tier-specific allocation rules while tracking already-sampled files to control duplication.

This ensures 100% coverage of all 33 vulnerability types while maintaining the same distribution as the original dataset. We then create a test set consisting of 500 samples not in the training sets to prevent leakage.

A.3 Verifiable Tests Generation Pipeline

The original FORMAI-v2 dataset includes vulnerability checks using ESBMC but lacks functionality

tests. We argue that an effective functionality test suite should: (1) achieve high code coverage to validate broad program behavior, (2) verify the correct behavior of the original code, and (3) detect whether patches inadvertently break existing functionality. Accordingly, we establish that test suites must achieve at least 90% line coverage of the original code while avoiding edge-case paths, as these are more likely to trigger vulnerabilities and may confound functionality verification. Additionally, we require all test cases to pass on the original code, establishing a clear baseline for detecting behavioral changes in patched versions, if all tests pass on the patched code, correct behaviors are demonstrably preserved. We opt against differential testing, which compares outputs from vulnerable and patched code given identical inputs, due to its complex setup.

Based on these criteria, we designed an agentic test case generation pipeline 1 that iteratively develops test suites through interaction with the environment. The agent receives a code file, generates an initial test suite with CMocka and CMake, and executes it in a Docker container environment. The container returns execution logs, test results, and coverage metrics, which the agent analyzes to identify gaps in coverage, failing tests, or build errors. Using this feedback, the agent refines the test suite, adding new test cases, fixing compilation issues, or adjusting assertions, and submits it for re-execution. This feedback loop continues iteratively, with each cycle improving the test suite's quality and coverage, until either all criteria are satisfied or the allocated time budget is exhausted. After the pipeline run, we check the test suite coverage and remove all the data points that do not meet the requirements.

We then run this test case generation pipeline to generate the test suite for the test set and the validation set. We do not run the pipeline on the training set due to its high cost. We finally obtained a validation set consisting of 19 data points and a test set of 426 data points. We would release the dataset in the future under the MIT license for vulnerability patching research.

B Training configurations

We trained our models using VerL (Sheng et al., 2025), which is under the Apache-2.0 license, on 4 A100 80 GB GPUs. Each run takes 3 days. The settings are provided in Tab 8.

Setting	**Value**
Training	
Epochs	3
Train batch size	64
Group size G	8
Gradient clipping	1.0
ε_{low}	0.2
$\varepsilon_{\text{high}}$	0.28
c	10.0
Learning rate	1×10^{-5}
LR schedule	Cosine
Warmup steps	5
Min LR ratio	0.0
Weight decay	0.1
Max prompt length	8,096 tokens
Max response length	10,240 tokens

Table 8: Training hyperparameters and configurations for VRPO.

C Dual Optimization Framework

C.1 Formulation

Since finding an optimal λ requires hyperparameter sweeping and does not guarantee the optimal trade-off between vulnerability patching and functionality retention, we leverage a primal-dual optimization approach as in Safe-RLHF (Dai et al., 2023).

Let the secure loss be defined as:

$$\mathcal{J}_{\text{sec}}(\theta) = \mathbb{E}_{\substack{x\sim\mathcal{D},\\ \{y_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(\cdot|x,V(x))}} \Bigg[\frac{1}{\sum_{i=1}^{G}|y_i|}\sum_{i=1}^{G}\sum_{t=1}^{|o_i|} \min\left(r_{i,t}(\theta)\,\hat{A}_{i,t},\ \text{clip}(r_{i,t}(\theta),\,1-\varepsilon_{\text{low}},\,1+\varepsilon_{\text{high}})\,\hat{A}^{sec}_{i,t}\right)\Bigg] \tag{6}$$

and the functionality loss is:

$$\mathcal{J}_{\text{func}}(\theta) = \mathbb{E}_{\substack{x\sim\mathcal{D},\\ \{y_i\}_{i=1}^{G}\sim\pi_{\theta_{\text{old}}}(\cdot|x,V(x))}} \Bigg[\frac{1}{\sum_{i=1}^{G}|y_i|}\sum_{i=1}^{G}\sum_{t=1}^{|o_i|} \min\left(r_{i,t}(\theta)\,\hat{A}_{i,t},\ \text{clip}(r_{i,t}(\theta),\,1-\varepsilon_{\text{low}},\,1+\varepsilon_{\text{high}})\,\hat{A}^{func}_{i,t}\right)\Bigg] \tag{7}$$

Note that $\hat{A}^{sec}$ and $\hat{A}^{func}$ are computed separately using secure and functionality reward. Then, we define the optimization objective as follows:

$$\mathcal{J}_{\text{VRPO}}(\theta,\lambda) = \frac{1}{1+\lambda}(\mathcal{J}_{\text{sec}}(\theta)+\lambda\mathcal{J}_{\text{func}}(\theta)) \tag{8}$$

Specifically, in one optimization step, we jointly optimize the policy parameters θ and λ via gradient ascent.

$$\theta_{k+1} \leftarrow \theta_k + \eta_\theta \nabla_\theta \mathcal{J}_{\mathrm{VRPO}}(\theta_k, \lambda_k), \quad (9)$$

$$\ln \lambda_{k+1} \leftarrow \ln \lambda_k + \alpha \cdot \lambda_k \cdot (\mathcal{J}_C(\theta_k) - b) \quad (10)$$

where η_θ and α are learning rates of the policy and the multiplier λ, respectively. Note that the update must keep the Lagrange multiplier strictly positive, because a negative constraint weight would break the meaning of the penalty. Therefore, we apply a log-reparameterization to guarantee this while keeping the optimization stable.

C.2 Training Process

We initially set $\lambda = 1$, so the security and functionality loss terms contribute equally to the final loss. The learning rate of λ is set to 0.01, and there is a 5-step warm-up before updating λ.

Training under the primal-dual framework is notably less stable than standard VRPO. As shown in Fig. 4, the functionality reward exhibits abrupt fluctuations throughout training, rather than converging smoothly. In the early phase, the gradient clipping ratio is high, reflecting large policy updates, before dropping near zero following the warm-up period. Response length similarly oscillates without settling, introducing variate training step length.

D Reward Hacking

During the earlier training run, with a simpler LLM-as-a-Judge prompt, we observed a reward hacking run from a VRPO-trained Qwen3-4B model. Specifically, the model achieved a superior performance when pairing with LLM-as-a-Judge functionality checking: 89.80% COMP%, 92.07% FUNC%, 99.94% SEC %, and 99.55% FIX %. Despite the positive result, the patches are mostly empty main functions that contain comments such as "// old code" (shown in Listing D). Since there is no code in the main function, the vulnerability checker detects no vulnerabilities, and the LLM-as-a-Judge score is extremely high because it treats the comments as placeholders for real code. This motivated us to design the test case generation pipeline and to rework the LLM-as-a-Judge prompt by adding rubric-based grading and random position swapping for before/after code pairs.

Listing 1: Example of an incomplete patch generated by an LLM.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// ... [rest of the code remains the same] ...

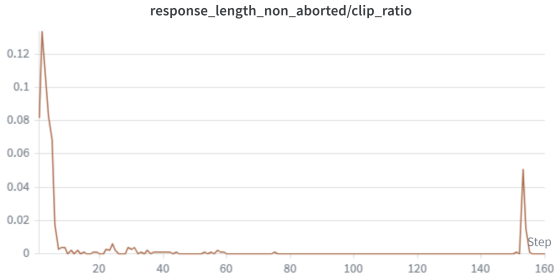
int main()
// ... [rest of the code remains the same] ...

return 0;

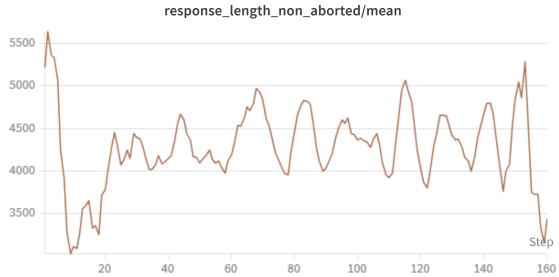
// ... [the rest of the functions and includes are as original] ...
```

E Prompts

We provide the core prompts for the LLM vulnerability patcher and LLM-as-a-Judge system.

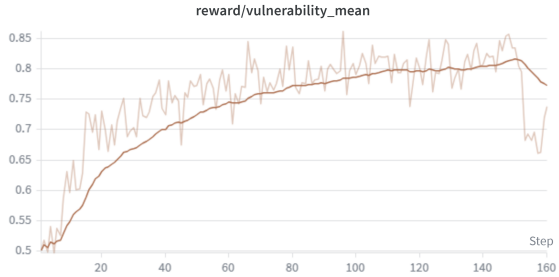


(a) Gradient clipping over training steps.

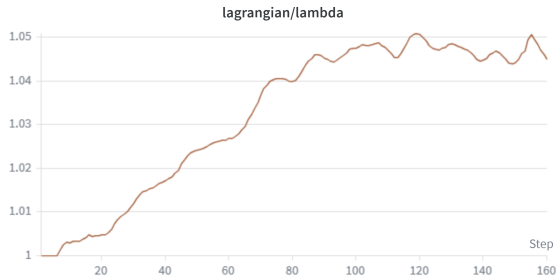


(b) Response length over training steps.

(c) Functionality reward over training steps.



(d) Security reward over training steps.



(e) Lagrange multiplier λ over training steps.

Figure 4: Training dynamics of VRPO under the primal-dual optimization framework. λ adapts dynamically to balance the security and functionality objectives, starting from $\lambda = 1$ and evolving as the policy improves.

Listing 2: LLM Patcher prompt.

```
// File: file_name
{source_code}
// Violation i: {violation_i_information}
Given the vulnerable code and the list of violations found, provide a corrected
version of the code that addresses all the violations. You FIRST think about the
reasoning process as an internal monologue and then provide the final answer.
The reasoning process MUST BE enclosed within <think> </think> tags. The final
answer MUST be enclosed within ```c ``` tags.  If the code cannot be fixed,
explain why within the <think> </think> tags and provide the original code
within the ```c ``` tags.
```

Listing 3: LLM-as-a-Judge Functionality Prompt.

```
You are a differential testing agent. Determine if two C code versions behave
identically for valid inputs.

**Original Code:**
```c
{original_code}
```
Modified Code:
```c
{patched_code}
```

Task

Mentally test both versions with various inputs and compare their behavior.

Methodology

STEP 1: Completeness Check
If modified code has '...', placeholders, or incomplete functions → Score 0.0
and STOP.

STEP 2: Identify Changes
List all modifications between original and modified code.

STEP 3: Differential Testing
Think through different input scenarios:
- **Valid inputs**: Normal cases, edge values, boundary conditions
- **Invalid inputs**: NULL pointers, overflows, bad data

For each scenario, mentally trace execution in both versions:
- Do they return the same values?
- Do they produce the same side effects?
- Do they have the same behavior?
```

```
**STEP 4: Classify Differences**
- **No impact**: Valid inputs behave identically → High score
- **Acceptable**: Only differs on invalid/crash inputs (modified is safer) →
Don't penalize
- **Breaking**: Different behavior on valid inputs → Lower score

## Scoring

- **1.0**: Perfect - identical for all valid inputs
- **0.8-0.9**: Cosmetic only - no behavioral differences
- **0.6-0.7**: Minor edge case differences
- **0.4-0.5**: Some valid input differences
- **0.2-0.3**: Major functional differences
- **0.0-0.1**: Incomplete code or completely different

## Output

<score>X.X</score>

## Rules
- Incomplete code = automatic 0.0
- Valid inputs must behave identically for high scores
- Improved crash handling doesn't lower score
- Focus on runtime behavior, not style
```

Listing 3: LLM-as-a-Judge Security Prompt.

```
You are a C security patch reviewer. Determine whether a generated patch
implements the same vulnerability remediation as the ground-truth patch.

**Vulnerable Code:**
```c
{vulnerable_code}
```

Ground Truth Patch:
```c
{ground_truth}
```

Generated Patch:
```c
{generated_patch}
```

CWE: {cwe}
CVE: {cve}
CVE Description: {cve_desc}
```

```
## Task

Compare the generated patch to the ground-truth patch and judge whether it fixes
the same vulnerability without introducing an equivalent security weakness.

## Methodology

**STEP 1: Completeness Check**
If the generated patch has `...`, placeholders, missing functions, or non-C
explanatory text instead of code -> Score 0.0 and STOP.

**STEP 2: Vulnerability Fix Check**
Identify the security-relevant change in the ground-truth patch, then inspect
whether the generated patch provides an equivalent mitigation.

**STEP 3: Regression Check**
Consider whether the generated patch leaves the original vulnerability
exploitable or introduces a closely related issue.

## Scoring

- 1.0: Fully equivalent security fix
- 0.8-0.9: Fixes the vulnerability with minor non-security differences
- 0.6-0.7: Mostly fixes the issue but misses a narrow edge case
- 0.4-0.5: Partial mitigation, important security gaps remain
- 0.2-0.3: Weak or mostly unrelated mitigation
- 0.0-0.1: Incomplete code or does not address the vulnerability

## Output

<score>X.X</score>

## Rules
- Judge security remediation, not formatting
- Prefer semantic equivalence over textual similarity
- Only output the score tag
```