SYNTHFIX: Adaptive Neuro-Symbolic Code Vulnerability Repair

Yifan Zhang[1] **Jieyu Li**[1] **Kexin Pei**[2] **Yu Huang**[1] **Kevin Leach**[1]
Vanderbilt University[1] University of Chicago[2]
{yifan.zhang.2,yu.huang,kevin.leach}@vanderbilt.edu
kpei@cs.uchicago.edu

Abstract

Large Language Models (LLMs) show promise for automated code repair but often struggle with the complex semantic and structural correctness required. We present SYNTHFIX, a hybrid neuro-symbolic framework that improves LLM-based vulnerability repair by unifying code synthesis with compiler-informed symbolic feedback. The core of our approach is an adaptive training strategy where a neural *Router Model* directs code samples to either Supervised Fine-Tuning (SFT) to learn common patterns or Reward Fine-Tuning (RFT) with symbolic rewards for complex, iterative refinement. On the FixJS (JavaScript) and CodeFlaws (C) benchmarks, SYNTHFIX achieves up to 18% relative improvement in CodeBLEU/CrystalBLEU and 32% in Exact Match over strong SFT and RFT baselines. Our results show that this adaptive combination of training strategies, which mirrors how developers alternate between pattern application and tool feedback, significantly improves the accuracy and efficiency of LLM-based vulnerability repair. Our code and data are available at https://github.com/CoderDoge1108/SynthFix[1].

1 Introduction

Large Language Models (LLMs) have demonstrated a remarkable ability to generate and modify source code by learning from vast open-source corpora (Chen et al., 2021; Allamanis et al., 2018). However, this proficiency is largely based on learning syntactic patterns and textual plausibility, often failing to grasp the strict semantic and structural constraints required for tasks like security vulnerability repair (Berabi et al., 2021; Jiang et al., 2023). This gap can lead to models proposing fixes that appear correct on the surface but remain functionally flawed or introduce new bugs, a critical failure when software security is at stake (Fu et al., 2022). Bridging this gap requires moving beyond purely neural approaches to integrate the symbolic reasoning inherent in compiler and program analysis tools, a challenge that the dominant LLM training paradigms are not equipped to handle (Shi et al., 2020; Wu et al., 2022).

The dominant paradigm for adapting LLMs to this task is Supervised Fine-Tuning (SFT) on datasets of bug-fix pairs (Dong et al., 2023). While effective for learning common, syntactic repair patterns, SFT struggles to impart the deeper reasoning needed to fix complex semantic vulnerabilities not explicitly seen in the training data (Berabi et al., 2021). To overcome this, Reward Fine-Tuning (RFT), often using reinforcement learning algorithms like PPO, has emerged as a promising alternative (Le et al., 2022; Islam and Najafirad, 2024; Fang et al., 2025). RFT allows models to learn from dynamic, task-specific feedback, such as whether a patch passes a security check, enabling iterative refinement toward a correct solution (Shojaee et al., 2023). However, RFT methods are notoriously sample-inefficient and computationally expensive, often suffering from slow convergence that limits their practical application (Schulman et al., 2017).

To resolve this tension, we present SYNTHFIX, a hybrid neuro-symbolic framework designed to harness the pattern-matching efficiency of SFT with the deep, feedback-driven refinement of RFT. Our approach enriches the learning process by integrating comprehensive, compiler-informed feedback, a concept explored in neuro-symbolic program synthesis (Parisotto et al., 2016). Specifically, SYNTHFIX enhances the RFT training phase by constructing a dense reward signal from multiple symbolic sources. This reward combines structural validation from Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs) (Wang et al., 2021a; Liang et al., 2019) with semantic correctness checks from vulnerability detection tools. By

[1]Our artifacts are publicly available under the MIT License.

Findings of the Association for Computational Linguistics: ACL 2026, pages 26857–26870

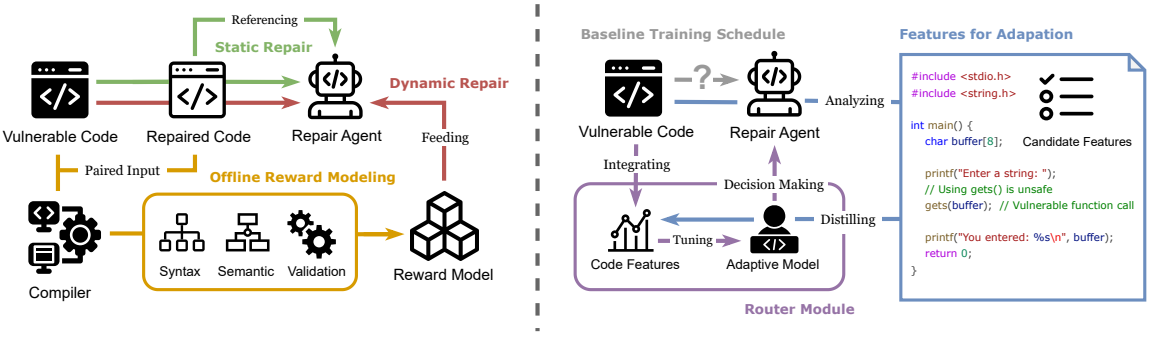


Figure 1: Overview of SYNTHFIX. Left: Candidate Patch Generation and Selective Validation (featuring the Repair Agent and static repair via SFT). Right: Adaptive Module (comprising the Router Model and the offline trained Reward Model) for dynamic strategy selection via RFT.

providing such multifaceted feedback, our framework guides the model toward repairs that are not only syntactically plausible but also structurally sound and semantically secure, addressing a key limitation of purely neural repair methods.

A key innovation of SYNTHFIX is its adaptive training mechanism, enabled by a neural **Router Model** that intelligently allocates computational resources. Instead of adhering to a fixed training strategy (Shen and Chen, 2020; de Fitero-Dominguez et al., 2024), our router dynamically selects the optimal learning path, either SFT or RFT, for each incoming batch. This decision is informed by real-time, compiler-derived metrics such as AST complexity and CFG depth. The underlying principle is to direct syntactically straightforward repairs to the efficient SFT pipeline, while reserving the more computationally expensive, feedback-driven RFT for complex cases requiring deeper semantic reasoning (Kulsum et al., 2024). This dynamic scheduling optimizes the trade-off between learning efficiency and repair quality, leading to faster convergence and a more robust final model.

We conduct an extensive empirical evaluation on two widely-used vulnerability repair benchmarks: FixJS for JavaScript (Csuvik and Vidács, 2022) and CodeFlaws for C (Tan et al., 2017). Across a suite of state-of-the-art code language models, including CodeLLaMA (Roziere et al., 2023) and StarCoder2 (Lozhkov et al., 2024), our results demonstrate that SYNTHFIX substantially outperforms baselines trained with either SFT or RFT exclusively. Furthermore, through detailed ablation and qualitative analyses, we confirm that our adaptive mechanism, which leverages rich symbolic feedback, is the key driver of this success, robustly addressing diverse categories of vulnerabilities.

2 Approach

Our approach, SYNTHFIX, is a hybrid training framework designed to resolve the tension between the efficiency of Supervised Fine-Tuning (SFT) and the semantic depth of Reward Fine-Tuning (RFT). We provide detailed background on these core methodologies in Appendix A. The framework trains a single **Repair Agent** by dynamically alternating between these two learning pathways, guided by an adaptive scheduling mechanism. At its core, SYNTHFIX consists of three key components: (1) a symbolic **Reward Model** that provides dense, compiler-informed feedback on patch quality for RFT; (2) a neural **Router Model** that predicts the optimal training strategy (SFT or RFT) for a given batch of code; and (3) the **Repair Agent** itself, which is progressively updated using the selected strategy. An overview of this architecture is presented in Figure 1. We first detail the design of our symbolic reward function, followed by the adaptive routing mechanism, and conclude with the complete training algorithm.

2.1 Symbolic Reward Model for Patch Evaluation

The goal of our framework is to train a **Repair Agent**, a transformer-based model parameterized by θ, to map a vulnerable code snippet x to a corrected patch $\hat{y}$, a process defined as $\hat{y} = F(x;\theta)$. A core component of SYNTHFIX is its symbolic reward model, which provides a fine-grained evaluation of the generated patch $\hat{y}$ to effectively guide the RFT process (Appendix A.2). Instead of relying on a sparse, binary signal (e.g., pass/fail), we construct a dense, composite reward $r(\hat{y}) = \lambda_{\text{AST}} r_{\text{AST}}(\hat{y}) + \lambda_{\text{CFG}} r_{\text{CFG}}(\hat{y}) + \lambda_{\text{Semgrep}} r_{\text{Semgrep}}(\hat{y})$. As detailed in Appendix A.3, each component pro-

vides unique feedback from symbolic representations:

Syntactic Correctness (r_{AST}) Measures the structural soundness of a patch via its Abstract Syntax Tree (AST), encouraging well-formed code (Wang et al., 2021a).

Logical Fidelity (r_{CFG}) Uses the Control Flow Graph (CFG) to assess if a patch preserves the intended program logic (Liang et al., 2019).

Assessed Security (r_{Semgrep}) Leverages a static analysis tool to check if the vulnerability has been eliminated, providing a direct signal for security correctness (Ziems and Wu, 2021).

This multi-faceted reward guides the Repair Agent to generate fixes that are robust across multiple dimensions (Gao et al., 2023).

2.2 Adaptive Training with a Router Model

With the symbolic reward defined, we now detail the adaptive mechanism that intelligently schedules the SFT (Appendix A.1) and RFT (Appendix A.2) training strategies. The centerpiece of this mechanism is a neural **Router Model**, a lightweight classifier trained to select the most effective learning path for each training batch B (see Appendix D for training details). To inform its decision, the router analyzes a compiler-derived feature vector $\mathbf{f}_B = [f_1(B), \ldots, f_k(B)]$, where the components summarize properties of the samples in the batch such as AST complexity, CFG depth, and code length (Zhang et al., 2022b). The precise set of features (and thus the input dimensionality k) is implementation-specific; we describe the instantiation used in our experiments in Appendix B.

The underlying hypothesis is that these features serve as a proxy for repair complexity; syntactically simple fixes are best learned via efficient SFT, while semantically complex vulnerabilities require the iterative feedback of RFT (Kulsum et al., 2024). After normalizing the feature vector to $\tilde{\mathbf{f}}_B$, the router produces a binary decision $d_B = R(\tilde{\mathbf{f}}_B) \in \{0, 1\}$, where $d_B = 0$ directs the batch to the SFT pathway and $d_B = 1$ directs it to the RFT pathway. This data-driven scheduling allows SYNTHFIX to optimize the trade-off between computational efficiency and semantic correctness, a substantial departure from static, non-adaptive training strategies (Yang et al., 2024).

Algorithm 1 The Training Pipeline of SYNTHFIX

Require: Dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, initial parameters θ

for each batch $B \subset \mathcal{D}$ **do**
  $\mathbf{f}_B \leftarrow [f_1(B), \ldots, f_k(B)]$ {Extract compiler-derived code features}
  $\tilde{\mathbf{f}}_B \leftarrow T(\mathbf{f}_B)$ {Normalize features}
  $d_B \leftarrow R(\tilde{\mathbf{f}}_B)$ {Router selects path: 0 for SFT, 1 for RFT}
  if $d_B = 0$ **then**
    Compute SFT loss $\mathcal{L}_{\text{SFT}}(B)$
    Update θ using $\nabla \mathcal{L}_{\text{SFT}}(B)$ {Perform SFT update}
  else
    Generate candidate patches $\{\hat{y}_i\}$ for all $x_i \in B$
    Compute reward $r(\hat{y}_i)$ for each patch {See Section 2.1}
    Compute RFT loss $\mathcal{L}_{\text{RFT}}(B)$
    Update θ using $\nabla \mathcal{L}_{\text{RFT}}(B)$ {Perform RFT update}
  end if
end for

2.3 Integrated Training Algorithm

We combine the symbolic reward model and the adaptive router into a single, unified training pipeline. The complete process for updating the **Repair Agent**'s parameters θ for a given batch is detailed in Algorithm 1. For each batch, the Router Model first analyzes the code's features to select a training path. If SFT is chosen, a standard supervised update is performed by minimizing the $\mathcal{L}_{\text{SFT}}$ loss. If RFT is chosen, the agent generates candidate patches, which are then evaluated by our composite symbolic reward to calculate the $\mathcal{L}_{\text{RFT}}$ loss and perform the update. This integrated process allows SYNTHFIX to dynamically tailor its learning strategy to the complexity of the code it is repairing.

3 Experimental Design

This section details the experimental setup designed to rigorously evaluate SYNTHFIX. We assess our framework's performance on two standard vulnerability repair benchmarks across several state-of-the-art code language models. We first outline the models, datasets, and metrics used in our experiments. Then, we present the four key research questions that guide our comprehensive

analysis of SYNTHFIX's effectiveness and its core components.

3.1 Models and Baselines

We evaluate SYNTHFIX by fine-tuning four representative code language models to demonstrate that our framework provides consistent benefits across different architectures and scales. The selected models are: (1) **CodeGen-220M** (Nijkamp et al., 2022), an open-source model for program synthesis; (2) **CodeT5-350M** (Wang et al., 2021b), an identifier-aware encoder-decoder model; (3) **CodeLLaMA-7B** (Roziere et al., 2023), a foundational generative code model; and (4) **StarCoder2-7B** (Lozhkov et al., 2024), a next-generation model targeting improved reasoning. For each of these base models, we establish two strong baselines for comparison: an **SFT-only** approach and an **RFT-only** approach. The detailed architectures and training procedures for our Reward Model and Router Model are provided in the Appendix for full reproducibility.

3.2 Benchmarks and Evaluation Metrics

We evaluate our framework on two benchmarks: **FixJS** for JavaScript (∼300k bug-fixing commits) (Csuvik and Vidács, 2022) and **CodeFlaws** for C (∼4k defects) (Tan et al., 2017). To ensure reproducibility, we use a fixed random seed for a standard 80/10/10 train/validation/test split across all experiments.

Repair quality is assessed using three standard metrics: **CodeBLEU** (Ren et al., 2020) for syntactic and semantic accuracy; **CrystalBLEU** (Eghbali and Pradel, 2022) for finer-grained structural alignment; and **Exact Match (EM)**, the percentage of patches identical to the ground-truth fix.

All experiments were conducted on two NVIDIA A6000 GPUs. For full reproducibility, we provide a complete breakdown of our experimental setup, including model architectures, training hyperparameters, and reward function implementations, in Appendix B.

3.3 Research Questions

Our evaluation is guided by four research questions (RQs) designed to systematically assess the performance and core components of SYNTHFIX:

RQ1: Efficacy of Hybrid Training. How does a hybrid SFT+RFT approach compare to using either SFT or RFT exclusively?

RQ2: Effectiveness of Adaptive Routing. How effective is the adaptive Router Model at improving performance over a non-adaptive, fixed-ratio hybrid approach?

RQ3: Generalization Across Defect Types. Does the performance of SYNTHFIX generalize across different types of vulnerabilities and code functionalities?

RQ4: Ablation of Reward Components. What is the individual contribution of each component (AST, CFG, and Semgrep) in our symbolic reward model?

4 Result Analysis

This section presents our empirical results, which validate the effectiveness of SYNTHFIX's adaptive, neuro-symbolic approach. We structure our analysis around the four research questions, presenting the main quantitative results, generalization analysis, ablation studies, and qualitative case studies.

4.1 Overall Performance (RQ1 & RQ2)

To answer our first two research questions, we conduct a two-step analysis of the models' overall performance.

First, to address **RQ1**, we establish the benefit of a hybrid approach over exclusive training. Table 1 compares a non-adaptive, fixed-ratio version of our hybrid framework against the SFT-only and RFT-only baselines. The results are unequivocal: the fixed-ratio hybrid model consistently and significantly outperforms both exclusive training methods across all metrics and models. This confirms that combining pattern-based learning with feedback-driven refinement is fundamentally superior to using either strategy in isolation.

Having confirmed that a hybrid strategy is beneficial, we then address **RQ2** by evaluating if an *adaptive* strategy is even better. Table 2 directly compares our full, router-enabled SYNTHFIX against the high-performing fixed-ratio hybrid. The results clearly demonstrate that the adaptive Router Model provides an additional, significant performance boost across nearly all configurations, confirming that dynamically allocating each batch to its optimal training strategy is the most effective approach.

Beyond final accuracy, the adaptive router also improves training efficiency. As shown in Figure 2, the version of SYNTHFIX with the router not only

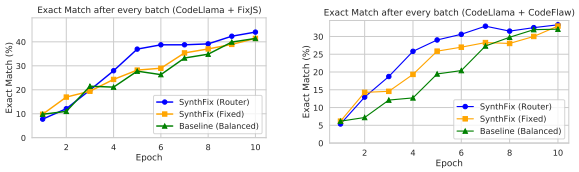
Table 1: Performance comparison of SFT-only, RFT-only, and our non-adaptive, fixed-ratio hybrid framework (SYNTHFIX). The results demonstrate the superiority of the hybrid approach, as it consistently achieves the highest scores (shown in **bold**) across all models and datasets. All methods were trained for 10 epochs.

Model	Training Method	FixJS (JavaScript)			CodeFlaws (C)		
		CodeBLEU (%)	CrystalBLEU (%)	Exact Match (%)	CodeBLEU (%)	CrystalBLEU (%)	Exact Match (%)
CodeT5	SFT	43.91	9.02	19.86	39.81	4.37	7.19
	RFT	44.72	13.23	11.55	40.34	8.67	9.65
	SYNTHFIX	**47.82**	**15.12**	**23.03**	**44.31**	**9.17**	**11.73**
CodeGen-350M	SFT	67.44	10.47	29.57	47.28	7.69	15.84
	RFT	67.11	15.80	18.39	49.74	16.85	18.34
	SYNTHFIX	**69.56**	**17.94**	**32.59**	**52.19**	**17.74**	**22.98**
CodeLLaMA-7B	SFT	76.51	18.32	30.83	55.19	9.93	18.68
	RFT	77.36	23.49	23.37	56.10	27.18	31.23
	SYNTHFIX	**83.71**	**25.72**	**40.71**	**58.07**	**29.93**	**33.14**
StarCoder2-7B	SFT	77.55	20.15	28.97	57.45	12.25	18.38
	RFT	78.79	25.95	20.11	60.37	29.13	30.40
	SYNTHFIX	**84.83**	**27.55**	**39.86**	**61.20**	**30.19**	**32.93**

Table 2: Evaluating the effectiveness of the adaptive Router Model (RQ2). Our full SYNTHFIX framework (Router) is compared against a non-adaptive baseline with a fixed SFT/RFT schedule (Fixed). The results confirm that dynamically scheduling training via the router provides a significant performance boost, achieving the highest scores (**bolded**) in nearly all cases.

Model	Training Method	FixJS (JavaScript)			CodeFlaws (C)		
		CodeBLEU (%)	CrystalBLEU (%)	Exact Match (%)	CodeBLEU (%)	CrystalBLEU (%)	Exact Match (%)
CodeT5	SYNTHFIX (Fixed)	47.82	15.12	23.03	44.31	9.17	11.73
	SYNTHFIX (Router)	**51.03**	**16.39**	**28.20**	**49.25**	**11.18**	**13.28**
CodeGen-350M	SYNTHFIX (Fixed)	69.56	17.94	32.59	52.19	17.74	22.98
	SYNTHFIX (Router)	**75.49**	**19.33**	**38.47**	**60.32**	**18.39**	**27.95**
CodeLLaMA-7B	SYNTHFIX (Fixed)	83.71	25.72	40.71	58.07	29.93	33.14
	SYNTHFIX (Router)	**86.94**	**27.72**	**42.03**	**60.28**	**30.44**	**35.64**
StarCoder2-7B	SYNTHFIX (Fixed)	84.83	27.55	39.86	**61.20**	30.19	32.93
	SYNTHFIX (Router)	**87.23**	**27.82**	**40.57**	61.13	**32.91**	**33.29**

reaches a higher peak accuracy but also converges faster than the fixed-ratio and balanced baselines, particularly in the later epochs. This demonstrates the tangible benefits of our adaptive training mechanism.



(a) FixJS (JavaScript) (b) CodeFlaws (C)

Figure 2: Exact Match accuracy per epoch on both datasets. The adaptive SYNTHFIX (Router) converges faster and achieves higher accuracy than fixed-schedule baselines.

4.2 RQ3: Generalization Across Defect Types

To answer **RQ3**, we investigate whether the performance gains of SYNTHFIX are consistent across different kinds of software bugs. We analyzed its performance by clustering the test data into fine-grained categories based on both vulnerability type (using their CWE classification (Christey et al., 2013)) and the code's functionality.

4.2.1 Performance by Vulnerability Type (CWE)

We first analyze performance across the most common vulnerability types found in the datasets. Tables 3 and 4 present the detailed performance breakdown for the top CWE categories in FixJS and CodeFlaws, respectively.

The results show that SYNTHFIX consistently outperforms both the SFT-only and RFT-only baselines across the vast majority of defect categories. For instance, in the case of CWE-79 (Cross-Site Scripting), SYNTHFIX achieves a 47.17% Exact Match score, substantially outperforming both baselines. While a baseline occasionally edges ahead on an individual metric in niche CodeFlaws categories (e.g., RFT on CrystalBLEU for CWE-190 and CWE-77, and SFT on Exact Match for CWE-20), SYNTHFIX demonstrates far greater overall robustness and achieves the highest Code-

BLEU across every CWE category and the highest Exact Match in the vast majority of them, confirming its ability to generalize across different vulnerability structures.

Table 3: Performance by CWE on the FixJS dataset. SYNTHFIX consistently achieves the highest scores (**bolded**) across all major vulnerability categories.

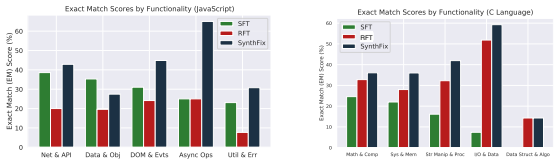
CWE ID	CodeBLEU (%)			CrystalBLEU (%)			Exact Match (%)		
	SFT	RFT	SYNTHFIX	SFT	RFT	SYNTHFIX	SFT	RFT	SYNTHFIX
CWE-79	76.51	76.46	**88.18**	21.88	22.17	**29.59**	33.96	18.87	**47.17**
CWE-20	83.28	72.01	**90.07**	25.37	19.52	**35.74**	31.11	25.64	**48.71**
CWE-22	78.70	75.64	**88.48**	19.82	20.90	**28.78**	32.26	22.58	**38.71**
CWE-502	68.93	76.25	**82.86**	9.10	21.33	**24.42**	24.14	27.59	**41.94**
CWE-200	74.04	75.32	**82.11**	17.32	19.12	**20.33**	26.47	25.00	**37.25**

Table 4: Performance by CWE on the CodeFlaws dataset. SYNTHFIX demonstrates robust generalization, achieving the best results (**bolded**) on most metrics across common C vulnerabilities.

CWE ID	CodeBLEU (%)			CrystalBLEU (%)			Exact (%)		
	SFT	RFT	SYNTHFIX	SFT	RFT	SYNTHFIX	SFT	RFT	SYNTHFIX
CWE-285	41.76	58.72	**60.28**	6.12	29.16	**30.44**	14.58	35.42	**35.64**
CWE-119	61.05	62.10	**65.40**	16.19	27.11	**31.77**	21.47	31.09	**40.12**
CWE-20	63.89	53.44	**68.07**	23.53	25.31	**35.74**	**51.28**	25.64	48.71
CWE-190	60.27	55.67	**63.52**	12.00	**20.05**	17.49	28.16	27.53	**52.63**
CWE-94	52.67	54.27	**60.45**	8.19	26.50	**33.06**	0.00	5.00	**12.50**
CWE-77	51.43	53.88	**55.20**	7.46	**25.37**	5.00	0.00	**19.47**	13.33
CWE-862	48.30	57.19	**60.38**	15.80	27.95	**30.19**	23.68	28.95	**34.21**

4.2.2 Performance by Code Functionality

We also analyze performance across different code functionalities to ensure our method is not biased towards a particular programming pattern. Figure 3 illustrates the Exact Match scores for the top functionality categories in both datasets. The charts clearly show SYNTHFIX's advantage, particularly in complex areas like *Asynchronous Operations* in JavaScript and *I/O & Data* in C, where it achieves substantial improvements over both baselines. This analysis further confirms that our adaptive approach generalizes effectively. The tables detailing the distribution of these categories are available in Appendix C.



(a) FixJS (JavaScript) (b) CodeFlaws (C)

Figure 3: Exact Match scores by code functionality. SYNTHFIX shows consistent and often substantial improvements over baselines across diverse categories in both languages.

4.3 Ablation of Reward Components (RQ4)

Finally, to answer **RQ4**, we conduct an ablation study to isolate the individual contribution of each component within our symbolic reward model. We evaluate the performance of SYNTHFIX after removing each of the three reward signals (AST, CFG, and Semgrep) one at a time.

Table 5: Ablation Study of Reward Components in SYNTHFIX (evaluated via CodeBLEU (CB), CrystalBLEU (CrB), and Exact Match (EM)).

Method	FixJS (JavaScript)			CodeFlaws (C)		
	CB (%)	CrB (%)	EM (%)	CB (%)	CrB (%)	EM (%)
CodeGen-350M						
All Rewards	**75.49**	**19.33**	**38.47**	**60.32**	**18.39**	**27.95**
Excluding AST	70.10	13.18	33.02	57.77	12.66	23.19
Excluding CFG	71.51	13.24	35.72	57.25	13.90	25.22
Excluding Semgrep	69.43	10.15	31.09	55.12	9.31	20.36
CodeLLaMA-7B						
All Rewards	**86.94**	**27.72**	**42.03**	**60.28**	**30.44**	**35.64**
Excluding AST	83.21	22.10	38.04	55.61	27.95	31.82
Excluding CFG	82.87	21.01	39.45	55.75	26.04	33.39
Excluding Semgrep	80.85	18.51	33.00	51.81	20.25	27.09

The results, presented in Table 5, lead to two clear conclusions. First, removing any single component degrades performance across all models and datasets, confirming that all three signals are necessary to achieve the best results. Second, while all components are important, the exclusion of the **Semgrep** signal consistently causes the most significant drop in performance. This finding highlights the critical impact of providing direct, semantic feedback on security correctness to the model. While the AST and CFG signals provide essential structural and logical guidance, the vulnerability-specific feedback acts as the strongest driver for the agent's repair capabilities. These results validate our design of the composite reward function, underscoring the effectiveness of combining multiple symbolic signals for nuanced code repair.

4.4 Qualitative Case Study

To provide a concrete, qualitative understanding of our model's capabilities, we examine two repair scenarios from the FixJS dataset. Figure 4 provides a visual comparison of the code diffs, and we present the final, superior patches generated by SYNTHFIX below for detailed analysis.

Case Study 1: Cross-Site Scripting (CWE-79). As shown in the left panel of Figure 4, a JavaScript web server is vulnerable to script injection. The **SFT** model applies a naive fix that only strips `<script>` tags, while the **RFT** model applies `escapeHtml(...)` but performs no other checks.

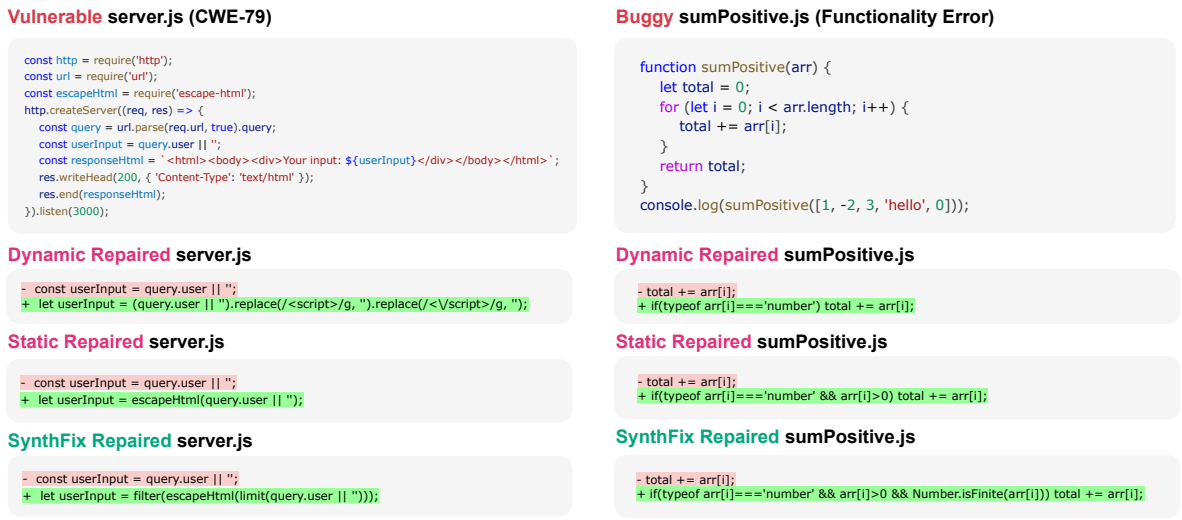


Figure 4: Comparison of code diffs generated by SFT, RFT, and SYNTHFIX for two representative cases: a Cross-Site Scripting (XSS) vulnerability and a logical error in an array summation function.

By synthesizing the pattern-matching of SFT with the semantic awareness learned through RFT, SYNTHFIX generates a superior, multi-layered defense:

```
let userInput = filter(
    escapeHtml(limit(query.user || ''))
);
```

This repair is more robust because it composes three distinct operations: `limit(...)` to enforce length, `escapeHtml(...)` to neutralize dangerous characters, and `filter(...)` to apply an allow list, showcasing a much deeper contextual understanding.

Case Study 2: Array Summation Logic Error. The right panel of Figure 4 shows a function that incorrectly sums array elements. The **SFT** model adds a partial fix (a type check) but still allows negative numbers. The **RFT** model produces a correct but verbose condition that includes a redundant check. SYNTHFIX synthesizes the ideal repair, which is both correct and concise:

```
if (typeof arr[i] === 'number' && arr[i] > 0)
    total += arr[i];
```

This succinct condition perfectly enforces the intended semantics without unnecessary checks, demonstrating the model's ability to generate minimal and efficient fixes. These cases illustrate how SYNTHFIX effectively synthesizes the strengths of both training paradigms to produce repairs that are more robust and precise.

5 Discussion

Our results provide strong evidence for the synergy between supervised and reinforcement learning in automated program repair. The success of the hybrid models suggests that SFT provides an essential foundation by efficiently learning common, syntactic repair patterns. This effectively "warms up" the model, allowing the more computationally expensive RFT process to focus its exploration on refining semantic and logical correctness, rather than learning basic syntax from scratch. This mirrors how human developers often apply known solutions to common problems while reserving deep, iterative debugging for novel or complex bugs.

Furthermore, the superior performance of the router-enabled SYNTHFIX highlights a key insight: program repair is not a monolithic task. Some vulnerabilities are simple and formulaic, while others are complex and nuanced. Our adaptive router effectively learns to distinguish between these cases based on static code features, acting as a learned curriculum scheduler. This finding suggests that future work on training LLMs for complex, multi-faceted tasks could benefit from similar adaptive mechanisms that allocate computational resources more intelligently.

Finally, the success of our composite reward function reinforces the value of neuro-symbolic approaches for tasks involving formal languages. While neural models excel at learning from large-scale data, our ablation study shows that injecting precise, symbolic feedback, especially domain-

specific semantic signals from tools like Semgrep, is the strongest driver of performance. This indicates that the most reliable LLMs for code will likely be those that can effectively ground their generative capabilities in the formal, verifiable logic of compilers and static analysis.

6 Related Work

Our work is situated at the intersection of automated program repair, neural code generation, and reinforcement learning. Prior research in this domain can be broadly categorized into three main streams, which we review in this section. We first discuss neural program repair methods that primarily rely on Supervised Fine-Tuning (SFT). We then cover the use of Reinforcement Learning (RL) for code generation and refinement. Finally, we review traditional symbolic and compiler-informed techniques for program analysis to contextualize our hybrid, neuro-symbolic approach.

6.1 Neural Program Repair with SFT

Neural network-driven approaches, particularly those based on large language models, have become a primary focus in automated code repair (Allamanis et al., 2018; Chen et al., 2021; Li et al., 2024; Bansal et al., 2023; Zhang, 2022; Zhang and Leach, 2025; Zhang et al., 2025a; Richter et al., 2022; Feng et al., 2020). The predominant training method is Supervised Fine-Tuning (SFT), where models learn to map buggy code to fixed code from large datasets of examples (Yin and Neubig, 2017; Hayati et al., 2018; Habib and Pradel, 2019; Li et al., 2019; Gupta et al., 2019; Allamanis et al., 2021; Jiang et al., 2023). While these SFT-based methods are effective at learning common syntactic patterns, they often lack a deep understanding of program semantics. This can result in proposed repairs that are syntactically plausible but functionally incorrect or fail to address the underlying vulnerability (Berabi et al., 2021; Huang et al., 2023; Pailoor et al., 2024; Zhang et al., 2022a; Kou et al., 2024; Ziems and Wu, 2021).

6.2 Reinforcement Learning for Code

To imbue models with a deeper semantic understanding, some research has turned to Reward Fine-Tuning (RFT) using reinforcement learning. This paradigm allows models to learn from dynamic feedback mechanisms, such as passing unit tests or security checks, which is crucial for addressing complex vulnerabilities (Le et al., 2022; Shojaee et al., 2023; Fang et al., 2025). Proximal Policy Optimization (PPO) is a common algorithm used for this iterative refinement (Schulman et al., 2017), and more recent work couples such refinement with multi-step symbolic verification signals (Zhang et al., 2025b). A significant challenge for these methods, however, is their high computational cost and slow convergence, which can limit their practical application in large-scale code repair scenarios (Huang et al., 2023; Shi et al., 2023).

6.3 Symbolic and Compiler-Informed Analysis

A separate line of research has long utilized traditional compiler techniques for code repair. These methods leverage symbolic intermediate representations like Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs) to gain precise structural and semantic insights into the code (Shi et al., 2020; Wu et al., 2022; Jiang et al., 2018; Klieber et al., 2021; Mandal et al., 2018; Wang et al., 2021a). While powerful, these approaches often depend on predefined heuristics or manually crafted repair templates. This reliance on fixed rules can make them less flexible and scalable when faced with the diverse and evolving nature of real-world software vulnerabilities (Zhang et al., 2022b).

Our work is positioned at the intersection of these three areas. While prior research has largely treated them as separate domains, we introduce a unified framework that adaptively leverages the pattern-matching strengths of SFT, the deep feedback of RFT, and the formal rigor of symbolic analysis.

7 Conclusion and Future Work

We introduced SYNTHFIX, a hybrid neuro-symbolic framework that improves automated vulnerability repair by adaptively scheduling SFT and RFT via a neural router and symbolic reward model, significantly outperforming standard baselines in both repair accuracy and training efficiency. Our results indicate that treating each batch as a routing decision, rather than committing to a single learning paradigm, lets compiler-informed rewards provide grounding that purely neural objectives lack. Natural next steps include incorporating **richer feedback** such as dynamic execution traces, improving **scalability** to large industrial codebases, and extending **cross-lingual transfer** from C and JavaScript to languages like Python.

Limitations

A primary limitation of our current approach is its reliance on static code features (e.g., AST complexity, CFG depth) for the Router Model. These features, while effective, do not capture dynamic runtime behavior, which could provide a richer signal for routing complex repairs that involve subtle execution logic. Additionally, our evaluation is conducted on established academic benchmarks. While standard for this type of research, these benchmarks may not fully represent the scale and intricate dependencies of large, industrial-grade software systems. Therefore, the performance of SYNTHFIX in such complex environments remains an open question for future validation.

Ethical Considerations

While the goal of SYNTHFIX is to improve software security, we acknowledge the potential risks associated with any automated program repair tool. A primary risk is inducing a false sense of security, where developers might over-rely on the system and reduce manual code audits. Our tool is not guaranteed to produce a perfect fix in all cases; a generated patch could be incomplete or introduce subtle runtime errors. Therefore, SYNTHFIX is intended to be used as an assistive tool to augment developer expertise, not to replace critical human oversight in the software security lifecycle. We advocate for its use in workflows that include mandatory testing and human review of all generated patches before deployment.

Use of AI Assistants. The authors used AI assistants (LLMs) for minor editorial assistance, including language polishing and LaTeX formatting. All research ideation, experimental design, implementation, analysis, and claims are the authors' own.

References

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876.

Aakash Bansal, Chia-Yi Su, Zachary Karas, Yifan Zhang, Yu Huang, Toby Jia-Jun Li, and Collin McMillan. 2023. Modeling programmer attention as scanpath prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1732–1736. IEEE.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*, pages 780–791. PMLR.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Steve Christey, J Kenderdine, J Mazella, and B Miles. 2013. Common weakness enumeration. *Mitre Corporation*.

Viktor Csuvik and László Vidács. 2022. Fixjs: A dataset of bug-fixing javascript commits. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 712–716.

David de Fitero-Dominguez, Eva Garcia-Lopez, Antonio Garcia-Cabot, and Jose-Javier Martinez-Herraiz. 2024. Enhanced automated code vulnerability repair using large language models. *Engineering Applications of Artificial Intelligence*, 138:109291.

Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang, Zheng Yuan, Chang Zhou, and Jingren Zhou. 2023. How abilities in large language models are affected by supervised fine-tuning data composition. *arXiv preprint arXiv:2310.05492*.

Aryaz Eghbali and Michael Pradel. 2022. Crystalbleu: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

Zihan Fang, Yifan Zhang, Yueke Zhang, Kevin Leach, and Yu Huang. 2025. Dpo-f+: Aligning code repair feedback with developers' preferences. *arXiv preprint arXiv:2511.01043*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and 1 others. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

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*, pages 935–947.

Leo Gao, John Schulman, and Jacob Hilton. 2023. Scaling laws for reward model overoptimization. In *International Conference on Machine Learning*, pages 10835–10866. PMLR.

Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Neural attribution for semantic bug-localization in student programs. *Advances in Neural Information Processing Systems*, 32.

Andrew Habib and Michael Pradel. 2019. Neural bug finding: A study of opportunities and challenges. *arXiv preprint arXiv:1906.00307*.

Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. *arXiv preprint arXiv:1808.10025*.

Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An empirical study on fine-tuning large language models of code for automated program repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1162–1174. IEEE.

Nafis Tanveer Islam and Peyman Najafirad. 2024. Code security vulnerability repair using reinforcement learning with large language models. *arXiv preprint arXiv:2401.07031*.

Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 298–309.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1430–1442. IEEE.

William Klieber, Ruben Martins, Ryan Steele, Matt Churilla, Mike McCall, and David Svoboda. 2021. Automated code repair to ensure spatial memory safety. In *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 23–30. IEEE.

Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. 2024. Do large language models pay similar attention like human programmers when generating code? *Proceedings of the ACM on Software Engineering*, 1(FSE):2261–2284.

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.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Jiliang Li, Yifan Zhang, Zachary Karas, Collin McMillan, Kevin Leach, and Yu Huang. 2024. Do machines and humans focus on similar code? exploring explainability of large language models in code summarization. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, pages 47–51.

Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30.

Hongliang Liang, Yuxing Yang, Lu Sun, and Lin Jiang. 2019. Jsac: A novel framework to detect malicious javascript via cnns over ast and cfg. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Avijit Mandal, Devina Mohan, Raoul Jetley, Sreeja Nair, and Meenakshi D'Souza. 2018. A generic static analysis framework for domain-specific languages. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 27–34. IEEE.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Shankara Pailoor, Yuepeng Wang, and Işıl Dillig. 2024. Semantic code refactoring for abstract data types. *Proceedings of the ACM on Programming Languages*, 8(POPL):816–847.

Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*.

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*.

Cedric Richter, Jan Haltermann, Marie-Christine Jakobs, Felix Pauck, Stefan Schott, and Heike Wehrheim. 2022. Are neural bug detectors comparable to software developers on variable misuse bugs? In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi,

Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Zhidong Shen and Si Chen. 2020. A survey of automatic software vulnerability detection, program repair, and defect prediction techniques. *Security and Communication Networks*, 2020(1):8858010.

Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 39–51.

Ke Shi, Yang Lu, Jingfei Chang, and Zhen Wei. 2020. Pathpair2vec: An ast path pair-based code representation method for defect prediction. *Journal of Computer Languages*, 59:100979.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. 2023. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*.

Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, and 1 others. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 180–182. IEEE.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021a. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*.

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

Bingting Wu, Bin Liang, and Xiaofang Zhang. 2022. Turn tree into graph: Automatic code review via simplified ast driven graph convolutional network. *Knowledge-Based Systems*, 252:109450.

Kai Yang, Jian Tao, Jiafei Lyu, Chunjiang Ge, Jiaxin Chen, Weihan Shen, Xiaolong Zhu, and Xiu Li. 2024. Using human feedback to fine-tune diffusion models without any reward model. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8941–8951.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*.

Yifan Zhang. 2022. Leveraging artificial intelligence on binary code comprehension. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–3.

Yifan Zhang, Chen Huang, Kevin Cao, Yueke Zhang, Scott Thomas Andersen, Huajie Shao, Kevin Leach, and Yu Huang. 2022a. Pre-training representations of binary code using contrastive learning. *arXiv preprint arXiv:2210.05102*.

Yifan Zhang, Chen Huang, Zachary Karas, Thuy Dung Nguyen, Kevin Leach, and Yu Huang. 2025a. Enhancing code llm training with programmer attention. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 616–620.

Yifan Zhang and Kevin Leach. 2025. Leveraging human insights for enhanced llm-based code repair. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 1536–1537.

Yifan Zhang, Junwen Yang, Haoyu Dong, Qingchen Wang, Huajie Shao, Kevin Leach, and Yu Huang. 2022b. Astro: An ast-assisted approach for generalizable neural clone detection. *arXiv preprint arXiv:2208.08067*.

Yueke Zhang, Yifan Zhang, Kevin Leach, and Yu Huang. 2025b. Codegrad: Integrating multi-step verification with gradient-based llm refinement. *arXiv preprint arXiv:2508.10059*.

Noah Ziems and Shaoen Wu. 2021. Security vulnerability detection using deep learning natural language processing. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1–6. IEEE.

A Technical Background for SYNTHFIX

This appendix provides technical details on the foundational methodologies adapted in SYNTHFIX. We briefly explain Supervised Fine-Tuning (SFT), Reward Fine-Tuning (RFT), and the compiler-informed code representations that are central to our framework's adaptive training and symbolic reward model.

A.1 Supervised Fine-Tuning (SFT)

Supervised Fine-Tuning is a standard technique for adapting pre-trained models to specific tasks. Given a dataset of buggy code snippets x_i and their corresponding ground-truth fixes y_i, SFT optimizes the model's parameters θ by minimizing the negative log-likelihood of the target sequences: $\mathcal{L}_{\text{SFT}}(\theta) = -\sum_i \log P_\theta(y_i \mid x_i)$.

In SYNTHFIX, **SFT serves as the efficient training pathway for learning common, syntactic repair patterns**. The Router Model directs less complex code samples to this path, allowing the Repair Agent to quickly learn from direct examples.

A.2 Reward Fine-Tuning (RFT)

Reward Fine-Tuning uses reinforcement learning to optimize a model based on feedback from a reward function, which is particularly useful when a ground-truth output is not available or when optimizing for complex criteria beyond surface-level similarity. We instantiate RFT using Proximal Policy Optimization (PPO), which maximizes a clipped surrogate objective function: $\mathcal{L}_{\text{RFT}}(\theta) = \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$. Here, $r_t(\theta)$ is the ratio of probabilities between the current and old policies, and $\hat{A}_t$ is the advantage estimate.

In SYNTHFIX, **RFT is the feedback-driven pathway for tackling semantically complex vulnerabilities**. It allows the Repair Agent to iteratively refine its patches by learning from our dense, multi-faceted symbolic reward signal, enabling it to find functionally correct solutions that SFT might miss.

A.3 Symbolic Representations for Reward and Routing

Our framework integrates symbolic reasoning by leveraging compiler-informed code representations, primarily Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs).

An **Abstract Syntax Tree (AST)** represents the hierarchical syntactic structure of code. A **Control Flow Graph (CFG)** models the program's execution paths, with nodes representing basic blocks of code and edges representing control flow transitions.

These representations serve a dual purpose in SYNTHFIX:

1. **For Adaptive Routing:** Compiler-derived metrics like AST complexity and CFG depth are extracted from each batch to form the feature vector f_B used by our Router Model to decide between the SFT and RFT pathways.

2. **For Reward Modeling:** They form the basis of our symbolic reward function used during RFT. We assess a candidate patch $\hat{y}$ by combining three signals into a composite reward: $r(\hat{y}) = \lambda_{\text{AST}} r_{\text{AST}}(\hat{y}) + \lambda_{\text{CFG}} r_{\text{CFG}}(\hat{y}) + \lambda_{\text{Semgrep}} r_{\text{Semgrep}}(\hat{y})$. Each component provides distinct feedback: r_{AST} measures syntactic correctness, r_{CFG} assesses logical fidelity, and r_{Semgrep} uses a static analysis tool to verify that the security vulnerability has been eliminated. Detailed implementation is provided in our open-source codebase.

B Detailed Experimental Setup

This section provides a detailed overview of the model architectures, training hyperparameters, and reward function implementations used to evaluate SYNTHFIX, supplementing the information in our Experimental Design section.

B.1 Model Architectures

Our framework consists of three main neural components: the Repair Agent, the Router Model, and a Critic Model used during reward calculation.

Repair Agent For our experiments with CodeLLaMA, we used the `codellama/CodeLlama-7b-hf` model, loaded with 4-bit quantization to manage memory resources. We employed a partial fine-tuning strategy, where only the final four transformer layers (28-31) and the language model head (`lm_head`) were unfrozen for training. All other model parameters remained frozen.

Router Model The Router Model is a lightweight Multi-Layer Perceptron (MLP) designed to predict the optimal training strategy. It consists of an input layer that consumes the compiler-derived feature vector $\tilde{\mathbf{f}}_B$ described in Section 2 (so its input dimensionality is dictated by the set of features chosen at instantiation), two hidden layers with 64 units each using ReLU activation, and a final output neuron with a Sigmoid activation to produce a probability for selecting the RFT pathway.

Critic Model A Critic Model is trained to estimate the value function for the PPO algorithm. It is also an MLP, with an input dimension of 512, two hidden layers of 256 units with ReLU activation, and a single linear output neuron to predict the scalar state-value.

B.2 Training Hyperparameters

All models were trained using the hyperparameters listed below. The tokenizer was initialized

from `codellama/CodeLlama-7b-hf` with left-side padding.

Global Training We trained for a total of **10 epochs** with a **batch size of 16**.

Repair Agent (Actor) The agent's trainable parameters were optimized using the **AdamW** optimizer with a learning rate of 5×10^{-5}. A **StepLR** scheduler was used to decay the learning rate with a gamma of **0.95** at the end of each epoch.

Router Model (Controller) The router was optimized separately using the **Adam** optimizer with a learning rate of 1×10^{-3}.

PPO Configuration For RFT steps, we used a PPO clipping parameter ϵ of **0.2**. We also implemented a custom heuristic to limit the number of PPO-trained batches to a maximum of three per epoch to balance its computational cost.

B.3 Reward Function Implementation

Our composite reward signal $r(\hat{y})$ is calculated procedurally from multiple symbolic sources.

$r_{\textbf{Semgrep}}$ This component is derived from a code quality score calculated using the **Semgrep** static analysis tool with a custom YAML ruleset. A patch starts with a maximum score of 100, with points deducted for specific vulnerability patterns (e.g., -10 for 'find_eval') or a large penalty for code that fails to parse.

$r_{\textbf{CFG}}$ This reward is based on the structural similarity of Control Flow Graphs. The CFGs for the generated and target code are extracted using a custom **Node.js script**. The final score is a composite metric calculated from node similarity, edge similarity (based on operation type), path similarity, and the total structural difference (new, removed nodes and edges).

C Dataset Distribution for RQ3 Analysis

This section provides supplementary details on the composition of the test sets used for the generalization analysis in RQ3. The following tables show the distribution of the most common vulnerability types (CWE) and code functionalities in the FixJS and CodeFlaws datasets, which were the basis for the analysis in the main paper.

C.1 Vulnerability Type (CWE) Distribution

Tables 6 and 7 list the most frequent CWEs in the FixJS and CodeFlaws test sets, respectively.

Table 6: Top Defect Categories in the FixJS Evaluation Dataset

CWE ID	CWE Name	Percentage (%)
CWE-79	Cross-Site Scripting (XSS)	13.25
CWE-20	Improper Input Validation	11.25
CWE-22	Path Traversal	7.75
CWE-502	Deserialization of Untrusted Data	7.25
CWE-200	Information Exposure	5.75

Table 7: Top Defect Categories in the CodeFlaws Evaluation Dataset

CWE ID	CWE Name	%
CWE-285	Improper Access Control	12.00
CWE-119	Buffer Overflow	10.00
CWE-20	Improper Input Validation	9.75
CWE-190	Integer Overflow	5.75
CWE-94	Code Injection	4.50
CWE-77	Command Injection	4.25
CWE-862	Missing Authorization	2.25

C.2 Code Functionality Distribution

Tables 8 and 9 list the most frequent code functionalities in the FixJS and CodeFlaws test sets, respectively.

D Router Model Training Details

The Router Model is trained concurrently with the Repair Agent, acting as a controller that learns an optimal scheduling policy. We frame this as a simple reinforcement learning problem where the router is an agent that takes an action (choosing SFT or RFT) and receives a reward based on the outcome of that action on the main Repair Agent's performance.

The router's parameters are optimized using a policy gradient objective. For each batch, the router outputs a probability p for its chosen action (e.g., the probability of choosing RFT). The loss is then calculated as:

$$\mathcal{L}_{\text{Router}} = -\log(p) \times R_{\text{feedback}}$$

The feedback reward, R_{feedback}, is designed to encourage routing decisions that lead to better performance for the main Repair Agent. It is calculated as the difference between a moving-average baseline loss of the Repair Agent and the actual loss

Table 8: Top Functionality Categories in the FixJS Dataset

Functionality Category	**Description**	**%**
Network & API Communication	Handles HTTP requests, AJAX calls, and external API integration.	17.5
Data & Object Manipulation	Processes and transforms arrays, objects, and JSON data.	12.8
DOM Manipulation & Event Handling	Interacts with the DOM, handling user events like clicks and form submissions.	7.3
Asynchronous Operations & Callbacks	Manages asynchronous tasks using callbacks, promises, and async/await.	5.0
Utility & Error Handling Functions	Includes helper routines, logging, and robust error-handling mechanisms.	3.3

Table 9: Top Functionality Categories in the CodeFlaws Dataset

Functionality Category	**Description**	**%**
Mathematical Computations	Arithmetic operations, statistical calculations, geometry, and numerical analyses.	30.5
System & Memory Management	Memory allocation, file management, threading, networking, and low-level operations.	12.5
String Manipulation & Processing	Concatenation, searching, splitting, replacing, and formatting strings.	7.8
Input/Output & Data Parsing	Reading from input sources, parsing data (CSV, JSON, XML), output to files or console.	6.8
Data Structures & Algorithms	Implementations of data structures (lists, trees, graphs) and algorithms (sorting, searching).	1.8

achieved on the current batch. To ensure a fair comparison, the SFT and RFT losses are normalized to the same scale before being used to compute this reward. This process incentivizes the router to send each batch to the pathway that is expected to yield the greatest reduction in the Repair Agent's loss, thereby learning an efficient, data-driven training curriculum.