SYNTHFIX: A Hybrid Neural-Compiler Framework for Code Vulnerability Repair

Anonymous ACL submission

Abstract

Addressing code vulnerabilities is crucial for software security and reliability. We present SYNTHFIX, an innovative framework for automated code repair that combines Supervised Fine-Tuning (SFT) with Proximal Policy Optimization (PPO) in an iterative training regime. Inspired by optimization strategies from statistical algorithms, SYNTHFIX balances the rapid pattern recognition of SFT with the adaptive learning of PPO. By incorporating compiler insights, such as Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and ESLint, SYNTHFIX enhances training dynamics, improving scalability and adaptability. Evaluation on the FixJS dataset with over 30k JavaScript code pairs, demonstrates that SYNTHFIX outperforms existing methods, achieving up to 7.78% improvement in CodeBLEU over SFT and 7.33% over PPO on the CodeT5 and CodeGen models. SYNTHFIX further shows substantial gains in Exact Match, achieving up to 2.16x improvement. This innovative training architecture outperforms traditional models and shows potential for advancing other software engineering tasks through feedback adjustments. The code for SYNTHFIX has been anonymized and can be found at https://github.com/iiiiiiii979/SynthFix

1 Introduction

In Software Engineering (SE), the automated repair of code vulnerabilities is essential for reducing extensive manual debugging efforts and enhancing system security and reliability. Traditional vulnerability repair, often rule-based and manual, lacks scalability, leading to the adoption of AI methods. Yet, these AI approaches, relying heavily on neural networks and natural language processing (NLP), primarily capture superficial code patterns, often missing deeper structural and semantic complexities crucial for effective repair (Mesbah et al., 2019; Jiang et al., 2018; An et al., 2018; Klieber et al., 2021).

Supervised Fine-Tuning (SFT) is quick to learn syntactic patterns but often fails to grasp the complexities of code, limiting its ability to address intricate vulnerabilities (Berabi et al., 2021; Huang et al., 2023). On the other hand, Proximal Policy Optimization (PPO) adopts a reinforcement learning strategy, enabling deeper analysis through iterative feedback-driven refinements. However, PPO's effectiveness is constrained by its resource intensity and slow convergence (Schulman et al., 2017). To address these limitations, enhancing PPO with an actor-critic architecture, which uses separate neural network models to distinctly assess the current policy (actor) and optimize the value function (critic), provides a more robust solution.

Inspired by iterative statistical models that refine parameter estimates to optimize performance (Wu, 1983; Gupta et al., 2011), SYNTHFIX effectively merges SFT with PPO in a hybrid framework specifically designed for software repair challenges. SFT sets the initial learning conditions, akin to an "expectation" step, quickly assimilating common coding patterns and addressing basic errors. PPO, enhanced by our specialized actor-critic architecture, acts as a "maximization" step, refining these foundational insights into deeper, more complex structural and logical aspects of code crucial for intricate software vulnerabilities.

This blend not only overcomes the limitations of each method when used alone—slow learning curves of PPO and superficiality of SFT—but also leverages their strengths to efficiently tackle sophisticated software engineering tasks. By integrating compiler constructs like Abstract Syntax Trees (AST), Control Flow Graphs (CFG), alongside ESLint[1], SYNTHFIX enhances the adaptability and scalability of our PPO's actor-critic architecture, key for software repair where adaptability to

[1]ESLint is a widely used linting tool that helps in identifying and reporting on patterns found in ECMAScript/JavaScript code, thus aiding in maintaining code quality and consistency.

various coding issues is essential.

Moreover, SYNTHFIX demonstrates substantial improvements, achieving up to a 7.78% increase in CodeBLEU over SFT and 7.33% over PPO. SYNTHFIX further shows notable gains in Exact Match, achieving up to 2.16x improvement, highlighting its precision and reliability in generating correct patches. These results underscore the potential of the proposed integrated approach of SFT and PPO for advancing automated code repair. Modifying the feedback in our PPO's critic model can inspire new methodologies, potentially setting new benchmarks in the field. Our contributions can be summarized as follows:

- We introduce SYNTHFIX, a novel hybrid framework that integrates SFT with PPO for automated code vulnerability repair, leveraging the strengths of both techniques to achieve superior results.
- We incorporate compiler constructs such as AST, CFG, and ESLint into the training process, improving the model's ability to address complex structural and semantic issues in code.
- Our evaluation on the FixJS dataset shows that SYNTHFIX outperforms existing methods, with up to a 7.78% improvement in CodeBLEU and a 2.16x gain in Exact Match over traditional SFT and PPO approaches.
- SYNTHFIX sets new standards in the field by achieving more accurate and reliable code patches, showcasing its potential to advance other software engineering tasks through adaptive feedback adjustments.

2 Approach

In this section, we introduce SYNTHFIX, a hybrid training method that combines SFT with PPO. By integrating compiler insights like AST, CFG, and ESLint scores, SYNTHFIX aims to improve code repair accuracy and adaptability. The following subsections outline the framework, actor-critic mechanism, and hybrid training strategy.

2.1 SYNTHFIX Framework

In this section, we outline SYNTHFIX, a framework that advances beyond traditional training methods.

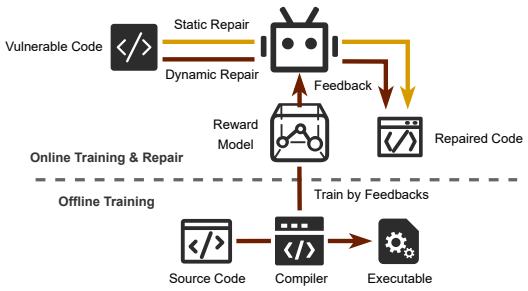

Figure 1: Overview of SYNTHFIX. This diagram shows the online and offline training processes in SYNTHFIX, highlighting how dynamic and static code repairs are informed by compiler insights and feedback loops to enhance model adaptability and effectiveness.

As depicted in Figure 1, our training process begins with *static repair* using SFT, as shown in Figure 2(b). This phase establishes basic code recognition and repair abilities.

The process then progresses to *dynamic repair* with PPO, detailed in Figure 2(a), which enhances these abilities into deeper repair skills using an actor-critic architecture. This dual-phase approach ensures thorough learning and effective vulnerability repair.

These two repairs cycle in a certain ratio, typically setting the one of smaller proportion as one epoch in a cycle. For example, in a 5:5 ratio, one SFT is followed by one PPO in a cycle; for a 7:3 ratio, two SFTs are followed by one PPO to complete one cycle, repeating three times with an additional SFT to adjust accordingly.

Figure 1 also highlights the critic component of our model, depicted in Figure 2(c), where the critic model pre-trained offline optimizes the *reward model* based on compiler feedback from AST, CFG, and ESLint scores. This comprehensive training strategy not only refines the repairs but also ensures high adaptability and precision, setting a new standard in automated code repair, further discussed in Subsection 2.4.

2.2 Actor-Critic Mechanism

Central to our approach is the actor-critic mechanism (Bahdanau et al., 2016; Grondman et al., 2012), depicted in Figure 2(c), which is crucial during the PPO training phase. The actor model proposes repair actions a_t, while the critic model evaluates these actions based on AST, CFG, and ESLint scores, generating a reward signal r_t.

The interaction is governed by the following equations:

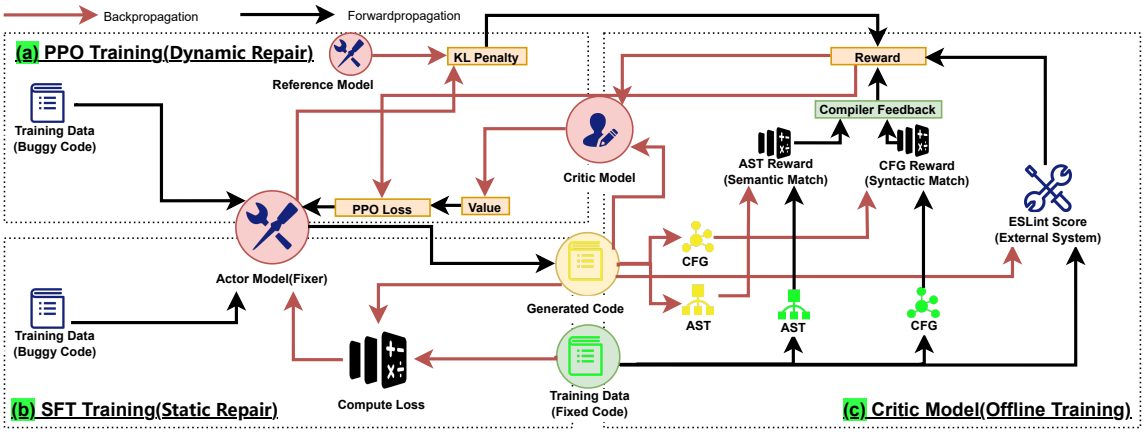

Figure 2: Detailed Workflow of SYNTHFIX. This diagram delineates the interactive processes among the dynamic and static repair training phases and the offline critic model training within SYNTHFIX. (a) PPO Training (Dynamic Repair) uses a reference model to guide the actor model through reward feedback enhanced by compiler metrics (AST and CFG rewards) and ESLint scores, facilitating syntactic and semantic improvements in generated code. (b) SFT Training (Static Repair) focuses on optimizing code fixes using static data to reduce bugs effectively. (c) The Critic Model (Offline Training) pre-trained on fixed and buggy code evaluates the actor's output using compiler insights, ensuring that the learning is grounded in both code quality metrics and practical functionality.

$$\phi^{(t+1)} = \arg\max_{\phi} \mathbb{E}_t \Big[\min \Big(r_t(\phi)\hat{A}_t,\ \text{clip}(r_t(\phi), 1-\epsilon,\ 1+\epsilon)\hat{A}_t \Big) \Big] \tag{1}$$

where $r_t(\phi) = \frac{\pi_\phi(a_t|s_t)}{\pi_{\phi_{\text{old}}}(a_t|s_t)}$ is the probability ratio, and ϵ is the clipping parameter.

The value network is updated as:

$$V^{(t+1)} = \arg\min_{V} \mathbb{E}_t \left[(V(s_t) - R_t)^2 \right] \tag{2}$$

Here, R_t is the cumulative reward. The actor uses r_t to refine its policy, improving code repair effectiveness over time.

2.3 Hybrid Training Methodology

The training protocol of SYNTHFIX combines supervised learning from static datasets with dynamic, interactive learning via PPO, as depicted in Figure 2(a) and Figure 2(b). Initially, SFT trains the model on common code repair patterns, serving as an "expectation" step where the model learns to recognize and respond to prevalent issues. This foundational phase sets up a robust base of coding knowledge, akin to preparing the model with a comprehensive understanding of typical coding errors and their solutions.

Transitioning to PPO, the "maximization" step of the Expectation-Maximization (EM) algorithm, the model refines this knowledge through real-time interactions and feedback, focusing on complex, less common problems that require deeper insight. This phase dynamically enhances the model's decision-making processes and deepens its understanding of intricate code structures and semantics, enabling a sophisticated response to diverse and challenging coding scenarios.

PPO is a policy gradient method for reinforcement learning that alternates between sampling data through interaction with the environment and optimizing a surrogate objective function using stochastic gradient descent. The key advantage of PPO over other policy gradient methods is its ability to keep updates within a specified range, preventing large, destabilizing updates. This is achieved by using a clipping function in the objective, defined as:

$$L^{PPO}(\phi) = \mathbb{E}_t \Big[\min \Big(r_t(\phi)\hat{A}_t,\ \text{clip}(r_t(\phi), 1-\epsilon,\ 1+\epsilon)\hat{A}_t \Big) \Big] \tag{3}$$

where $r_t(\phi) = \frac{\pi_\phi(a_t|s_t)}{\pi_{\phi_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the new policy π_ϕ and the old policy $\pi_{\phi_{\text{old}}}$, $\hat{A}_t$ is the advantage function, and ϵ is a hyperparameter for clipping. This clipping mechanism ensures that policy updates remain stable by limiting the extent of each update.

To complement this, the value network, which estimates the expected return for a given state, is updated by minimizing the following loss function:

$$L^{value}(V) = \mathbb{E}_t \left[(V(s_t) - R_t)^2 \right] \tag{4}$$

In this equation, $V(s_t)$ represents the estimated value of the state s_t, and R_t denotes the cumulative reward. By minimizing this loss, the critic model provides a more accurate evaluation of the actions suggested by the actor model, thereby guiding the policy refinement in the PPO phase.

The EM-like algorithm alternates between the SFT and PPO phases, iteratively refining the model's parameters θ, ϕ, and V to enhance both semantic and structural understanding. The process can be summarized in the following steps:

1. **Initialize** the model parameters θ, policy parameters ϕ, and value parameters V.

2. **Expectation Step (SFT)**:

$$\theta^{(t+1)} = \arg\max_{\theta} \sum_{i=1}^{N} \log p(x_i|\theta) \quad (5)$$

   In this step, the model is trained on a static dataset to predict fixed code from buggy code, with the parameters θ updated to maximize the likelihood of correct predictions.

3. **Critic Evaluation and Feedback**:

$$V^{(t+1)} = \arg\min_{V} \mathbb{E}_t \left[(V(\hat{y}_i^{SFT}) - R_t)^2 \right] \quad (6)$$

   Here, the critic model evaluates the output $\hat{y}_i^{SFT}$ from the SFT phase based on its structural and semantic accuracy, providing a feedback signal R_t to guide further training.

4. **Maximization Step (PPO)**:

$$\phi^{(t+1)} = \arg\max_{\phi} \mathbb{E}_t \Big[\min \Big(r_t(\phi)\hat{A}_t, \text{clip}(r_t(\phi), 1-\epsilon, 1+\epsilon)\hat{A}_t \Big) \Big] \quad (7)$$

   The actor model then uses this feedback to adjust its policy ϕ, aiming to generate more effective code repairs.

5. **Repeat** steps 2 to 4 until convergence is achieved, ensuring a thorough refinement of the model parameters.

In practice, for each data point x_i, the process begins with the SFT phase, establishing a baseline repair. The output from this phase, $\hat{y}_i^{SFT}$, is then evaluated by the critic model, which generates a feedback signal R_t. The PPO phase follows, where the actor model proposes a new repair a_i based on this feedback, and the critic model reassesses the repair. Through this iterative process, the model gradually improves its ability to perform robust and effective code repairs, leveraging the strengths of both static and dynamic learning methodologies.

2.4 Training the Critic Model

The critic model (D'Oro and Jaśkowski, 2020) in SYNTHFIX, illustrated in Figure 2(c), harnesses ASTs and CFGs to enhance code repair by ensuring both structural and behavioral correctness. ASTs capture the code's syntax and structure, ensuring alignment with language specifications and highlighting syntactic inaccuracies. CFGs complement this by capturing the execution paths within the code, validating the logical sequence of operations to maintain functional integrity.

The integration of ASTs and CFGs allows for a nuanced assessment that encompasses both the static aspects of code, such as its structure and syntax, and its dynamic behavior, including execution flow. This thorough evaluation ensures that repairs are not only syntactically correct but also functionally sound, leading to reliable and effective solutions.

Moreover, the model uses ESLint to enforce coding standards, where rule violations adjust the training rewards, thus promoting code that is not only correct but also clean and maintainable.

Compiler-informed rewards are specifically calculated based on: - **AST Reward:** Based on how closely the repaired code's AST aligns with that of the original, assessing structural and syntactic correctness. - **CFG Reward:** Focused on the integrity of control flows, ensuring that functional behaviors are preserved in the repaired code.

This integration of AST, CFG, and ESLint into the critic model's framework ensures a comprehensive evaluation of repairs, advancing the precision and efficacy of automated code repair processes.

The training algorithm, as outlined in Algorithm 1, exemplifies the structured approach adopted by SYNTHFIX to achieve this precision. By iteratively alternating between SFT and PPO, the model effectively balances the learning of static and dynamic code repair tasks. The Critic Evaluation phase serves as the linchpin, providing continuous feedback informed by syntactic and semantic metrics. This hybrid training process, characterized by its systematic refinement of model parameters, culminates in a robust framework capable of addressing complex code vulnerabilities with high

Algorithm 1 SYNTHFIX Hybrid Training

Require: Model parameters θ, policy parameters ϕ, value parameters V
Ensure: Convergence of model parameters
while not converged **do**
  SFT Phase:
  Train model on static dataset to predict fixed code from buggy code
  Update θ to maximize the likelihood of predicted code
  Obtain baseline repair $\hat{y}_i^{SFT}$
  Critic Evaluation:
  Evaluate $\hat{y}_i^{SFT}$ using AST, CFG, and ESLint metrics
  Generate feedback signal R_t
  PPO Phase:
  Actor model proposes repair a_i based on $\hat{y}_i^{SFT}$ and feedback R_t
  Critic model evaluates the proposed repair
  Update ϕ using PPO objective function to refine policy
  Iteration of Hybrid Training:
  Alternate between SFT and PPO phases according to the predefined ratio (e.g., 5:5 or 7:3)
  Repeat the cycle to further refine model parameters θ and ϕ
end while

accuracy.

3 Experimental Design

In this section, we outline the experimental setup used to evaluate SYNTHFIX's performance on automated code repair. We utilized the FixJS dataset, which includes over 30k JavaScript function pairs, and conducted experiments on two models, CodeGen-NL 350M and CodeT5, to demonstrate the framework's robustness across different architectures. The following subsections detail our evaluation criteria and research questions that guide the experimental analysis.

3.1 Experimental Setup

We use the FixJS dataset (Csuvik and Vidács, 2022), containing over 30,000 JavaScript function pairs from two million bug-fix commits, which is ideal for studying Automated Program Repair (APR) in JavaScript. The dataset is divided into training, validation, and testing in an 8:1:1 ratio. Our experiments were conducted on a workstation with dual NVIDIA A6000 GPUs.

We evaluated our framework using two models: the *CodeGen-NL 350M* model (Nijkamp et al., 2022), known for its proficiency with programming languages and diverse coding datasets, and the *CodeT5* model (Wang et al., 2021), which serves as a strong benchmark in code generation and repair tasks. By comparing results across both models, we aim to demonstrate the robustness and adaptability ofSYNTHFIX across different architectures.

3.2 Evaluation Criteria

CodeBLEU (Ren et al., 2020), used to assess our framework, combines traditional N-gram assessments with syntactic and dataflow evaluations. The overall CodeBLEU score is computed as a weighted average of the following metrics, aligning with standard practices:

- **N-gram Match Score:** Measures lexical similarity to the original code in the dataset.
- **Weighted N-gram Match Score:** Adjusts the importance of specific code elements based on their criticality.
- **Syntax Match Score:** Evaluates structural integrity through AST comparisons.
- **Dataflow Match Score:** Assesses logical integrity and functional correctness.

In addition to CodeBLEU, we also use **Exact Match** as a key evaluation metric. Exact Match measures the percentage of generated patches that are identical to the ground truth patches, serving as a robust indicator of the model's precision in code repair. This metric is crucial for assessing the effectiveness of SYNTHFIX in generating correct and reliable patches.

These metrics are calculated to compare the code generated by our model against the ground truth provided in the FixJS dataset, ensuring a rigorous and comparable evaluation with established benchmarks in automated code repair.

3.3 Research Questions

The uniformity of training duration and resource allocation is critical for a balanced experimental design. **SFT sessions last approximately 1 hour and 22 minutes, while PPO sessions are about 1 hour and 41 minutes.** This consistency allows us to implement equivalent epoch settings across both training methods, ensuring a fair comparison.

With a consistent experimental setup, we organized our experiments to systematically answer the following research questions:

- **RQ1:** How does the hybrid training approach, integrating SFT and PPO, compare against using SFT or PPO exclusively in terms of efficacy in code vulnerability repair?
- **RQ2:** What is the optimal balance between SFT and PPO epochs that maximizes repair performance?

- **RQ3:** How does the SYNTHFIX framework perform in a real-world application scenario for code repair, specifically focusing on JavaScript vulnerabilities?
- **RQ4:** What are the contributions of different components (AST, CFG, ESLint) in the reward model to the overall effectiveness of the PPO training?

4 Results

In this section, we present the outcomes of our experiments to evaluate the effectiveness of SYNTHFIX in repairing code vulnerabilities. We compare various training paradigms, optimize the balance between SFT and PPO epochs, and analyze the performance through case studies and ablation tests. These results highlight the strengths of our hybrid approach in achieving superior code repair accuracy and robustness.

4.1 RQ1: Analysis of Training Paradigms

Table 1 presents a detailed performance comparison of different training paradigms used for JavaScript code repair, evaluated on both *CodeT5* and *CodeGen-NL 350M* models. Each method is analyzed based on improvements over the SFT (Supervised Fine-Tuning) approach, which serves as a baseline for measuring the efficacy of more advanced training configurations.

The SFT-only approach demonstrates significant gains across most metrics, particularly in the N-gram and CodeBLEU scores, showcasing its efficiency in rapidly acquiring syntactic patterns. However, the PPO-only model, while slightly lagging in terms of quick pattern recognition, shows better performance in Syntax and Dataflow Match scores, emphasizing its ability to improve the deeper structural integrity of the code.

The hybrid SFT-PPO model, SYNTHFIX, which strategically intersperses three epochs of PPO with seven epochs of SFT, achieves the highest overall performance. For instance, on the *CodeGen-NL 350M* model, SYNTHFIX improves the CodeBLEU score by 2.97% over the SFT approach and by 2.56% over the PPO approach. On the *CodeT5* model, SYNTHFIX yields even more significant improvements, with an increase of 7.78% in CodeBLEU over SFT and 7.33% over PPO. This indicates that the combined approach effectively leverages the strengths of both SFT's quick learning and PPO's deep structural analysis, resulting in superior code repair outcomes.

4.2 RQ2: Optimizing the SFT-PPO Epoch Mix

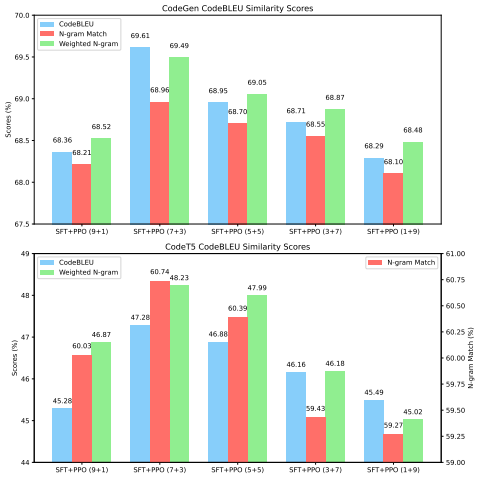

Figure 3: CodeBLEU similarity scores for CodeGen and CodeT5 across different SFT-PPO ratios. The 7:3 ratio shows optimal performance.

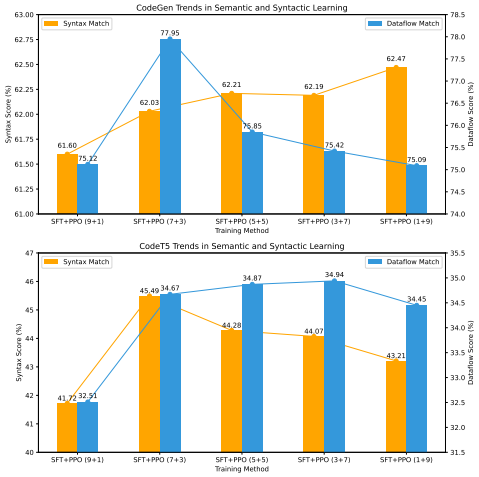

Figure 4: Trends in Syntax and Dataflow match scores for CodeGen and CodeT5 across different SFT-PPO ratios.

Figure 3 compares the impact of various SFT-PPO training ratios on *CodeGen* and *CodeT5* models. The 7:3 ratio achieves the highest CodeBLEU scores, 69.61% for *CodeGen* and 47.28% for *CodeT5*, effectively combining SFT for strong initial learning with PPO for targeted optimization. This balance is crucial for establishing a solid foundation while allowing for refinement through feedback-driven adjustments.

Figure 4 highlights trends in Syntax and Dataflow match scores across different SFT-PPO ratios. The 7:3 ratio also maintains the highest dataflow match scores, 77.95% for *CodeGen* and 34.67% for *CodeT5*, indicating well-preserved log-

Table 1: Performance Comparison of Different Training Paradigms. The training configuration for SYNTHFIX strategically distributed the epochs to ensure a balanced approach, interspersing three epochs of PPO evenly with seven epochs of SFT across the total training sessions.

Training Method	CodeBLEU (%)		N-gram (%)		Weighted N-gram (%)		Syntax (%)		Dataflow (%)		Exact Match (%)	
	CodeGen	CodeT5	CodeGen	CodeT5	CodeGen	CodeT5	CodeGen	CodeT5	CodeGen	CodeT5	CodeGen	CodeT5
Baseline	50.26	30.84	49.26	33.74	50.69	26.17	54.95	39.86	46.14	23.58	7.03	3.85
SFT (10 epochs)	67.60	43.87	67.43	59.19	67.98	44.62	60.04	40.73	74.97	30.94	29.12	18.24
PPO (10 epochs)	67.87	44.05	67.03	57.51	67.55	42.73	60.97	42.39	75.92	33.56	18.36	11.02
SYNTHFIX (10 epochs)	**69.61**	**47.28**	**68.96**	**60.74**	**69.49**	**48.23**	**62.03**	**45.49**	**77.95**	**34.67**	**32.97**	**23.84**
Improvement over SFT (%)	(+2.97)	(+7.78)	(+2.26)	(+2.62)	(+2.22)	(+8.09)	(+3.32)	(+11.68)	(+3.97)	(+12.04)	(+13.23)	(+30.73)
Improvement over PPO (%)	(+2.56)	(+7.33)	(+2.88)	(+5.61)	(+2.87)	(+12.88)	(+1.74)	(+7.31)	(+2.67)	(+3.31)	(+79.59)	(+116.31)

Table 2: Exact Match Score Comparison Across Different SFT-PPO Training Ratios for *CodeT5* and *CodeGen Models*.

Training Method	CodeGen Exact Match (%)	CodeT5 Exact Match (%)
SFT+PPO (9+1)	30.72	20.03
SFT+PPO (7+3)	**32.97**	**23.84**
SFT+PPO (5+5)	29.37	22.76
SFT+PPO (3+7)	28.11	20.38
SFT+PPO (1+9)	26.93	17.94

ical flow and functionality. Although higher PPO ratios improve syntax match scores, they may reduce semantic coherence, emphasizing the importance of finding the right balance between syntactic accuracy and overall code quality.

Table 2 confirms these results, with the 7:3 configuration achieving the best Exact Match scores for both models. This further supports the idea that a strategic mix of SFT and PPO epochs can effectively balance rapid pattern recognition with detailed structural refinement, making it the most effective approach for complex software repair tasks.

Overall, the 7:3 SFT-PPO mix proves most effective for *CodeGen* and *CodeT5*, ensuring both syntactic and functional integrity without compromising one for the other.

4.3 RQ3: Case Study

Figure 5 presents a case study of repairing a JavaScript function using the PPO, SFT approaches, and their combination in SYNTHFIX (SFT+PPO), all trained with the same computational resources on the *CodeGen-NL 350M* model. The original code snippet contains a syntax error in the conditional statement. Specifically, the condition "if (arr[i] =! null)" is syntactically incorrect and leads to compilation failure.

The Dynamic Repair (PPO) method corrects the syntax by replacing "=!" with "!=", resulting in a syntactically valid code. However, the fix does not fully address potential logical issues, as the strict inequality "!==" would be a more robust choice in JavaScript.

The Static Repair (SFT) method, on the other

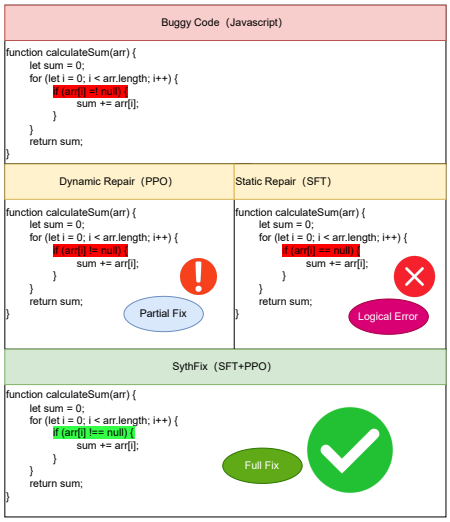

Figure 5: Case study example for JavaScript code repair. The erroneous snippets are highlighted in red.

hand, introduces a logical error by using the equality operator "==" instead of correcting the syntax with the appropriate operator. This results in a condition that incorrectly processes "null" values, leading to unintended behavior.

In contrast, SYNTHFIX (SFT+PPO) effectively combines the strengths of both methods, correcting the syntax and preserving the intended logic of the original code by using the strict equality operator "!==". This ensures that the generated code is both compilable and logically consistent with the original intent.

This case study demonstrates the effectiveness of SYNTHFIX in handling complex code repairs, where both syntax and logical correctness are critical. By leveraging compiler-level feedback and combining dynamic and static approaches, SYNTHFIX provides a more reliable solution for automated code repair, ensuring that the final code is both functional and adheres to best practices in software development.

4.4 RQ4: Ablation Study of Reward Model

Continuing from the optimal SFT-PPO mix identified in RQ2, this ablation study, detailed in Table

Table 3: Performance of Different Reward Model Configurations in Ablation Study. This table reflects the impact of including or excluding scores (AST, CFG, ESLint) on training outcomes.

Training Method	CodeBLEU (%)		N-gram (%)		Weighted N-gram (%)		Syntax (%)		Dataflow (%)		Exact Match (%)	
	CodeGen	CodeT5	CodeGen	CodeT5	CodeGen	CodeT5	CodeGen	CodeT5	CodeGen	CodeT5	CodeGen	CodeT5
Using all three scores	69.61	47.28	68.96	60.74	69.49	48.23	62.03	45.49	77.95	34.67	32.97	23.84
Excluding AST	67.10	45.54	66.17	58.34	67.22	46.65	60.09	44.08	74.91	33.29	30.15	21.80
Excluding CFG	66.92	45.39	66.35	58.50	66.87	46.40	60.01	44.00	74.43	33.06	30.04	21.77
Excluding ESLint	65.32	44.32	61.47	54.21	66.13	45.92	59.27	43.50	74.40	33.03	29.91	21.66
Excluding AST and CFG	64.97	44.07	63.79	56.27	65.06	45.20	58.79	43.18	72.22	32.08	28.14	20.38
Excluding AST and ESLint	64.59	43.76	61.59	54.23	65.74	45.68	58.93	43.30	72.08	32.02	28.64	20.76
Excluding CFG and ESLint	64.03	43.39	61.75	54.40	65.12	45.11	57.47	42.23	71.76	31.85	27.19	19.70

3, assesses the impact of omitting AST, CFG, and ESLint components.

Excluding CFG and ESLint scores particularly affects performance, with substantial reductions in CodeBLEU and dataflow scores, emphasizing their crucial roles in structural and semantic accuracy. This streamlined analysis reinforces the need for a comprehensive approach in neural models for effective code repair, ensuring deep semantic understanding alongside syntactic precision.

5 Related Work

This section reviews advancements in machine learning applied to code vulnerability repair, focusing on the integration of SFT, PPO, and compiler technologies within our framework, SYNTHFIX.

Neural Code Repair: Advancements in AI, particularly with large language models (LLMs) and code language models (CLMs), have substantially improved tasks like code synthesis, bug detection, and vulnerability repair through SFT (Yin and Neubig, 2017; Hayati et al., 2018; Parisotto et al., 2016; Habib and Pradel, 2019; Li et al., 2019; Gupta et al., 2019; Allamanis et al., 2021; Ziems and Wu, 2021; Thapa et al., 2022; Gupta et al., 2020). These models are often adapted to specific coding tasks using domain-specific datasets, enhancing their understanding and rectification capabilities (Xia and Zhang, 2022; Shi et al., 2023; Jiang et al., 2023). However, traditional approaches sometimes simplify code repair to mere content generation, missing deeper semantic and structural intricacies (Berabi et al., 2021; Huang et al., 2023). SYNTHFIX addresses these limitations by enriching SFT with compiler-level insights, thus improving both syntactic and semantic repair accuracy.

Reinforcement Learning in Software Development: While PPO is extensively used across various domains for adaptability and iterative learning, its application in code repair is less explored (Le et al., 2022; Shojaee et al., 2023). Reinforcement Learning (RL) has proven effective in managing dynamic adjustments based on feedback, essential in unpredictable environments like code optimization and software testing (Bagherzadeh et al., 2021; Wang et al., 2022). SYNTHFIX capitalizes on these attributes, applying PPO to enhance the adaptability and iterative improvement in vulnerability repair.

Compiler-Informed Insights in Code Repair: Compiler intermediate representations such as AST and CFG have been pivotal in advancing code vulnerability repair, aiding in software defect prediction and complex repair tasks (Shi et al., 2020; Wu et al., 2022; Mesbah et al., 2019; Jiang et al., 2018; An et al., 2018; Klieber et al., 2021; Chen et al., 2002; Mandal et al., 2018; Jiang et al., 2006). These insights help in understanding code dependencies and flows, crucial for addressing intricate vulnerabilities. In SYNTHFIX, these intermediate representations are seamlessly integrated with PPO, enhancing the model's learning efficacy and repair precision.

6 Conclusion

We presented SYNTHFIX, a hybrid neural-compiler framework that substantially enhances automated code vulnerability repair by integrating SFT and PPO. Our results demonstrate that SYNTHFIX outperforms traditional SFT or PPO methods, effectively combining syntactic learning with dynamic, feedback-driven adjustments.

The incorporation of compiler intermediate representations like AST and CFG within SYNTHFIX not only improves semantic robustness and structural integrity but also sets new benchmarks for reliability in software systems.

Future efforts will explore refining this integration and expanding its application to broader programming contexts, underscoring SYNTHFIX's potential to influence further advancements in automated code repair.

7 Limitations

While SYNTHFIX has demonstrated promising results in automated code vulnerability repair, several limitations remain that warrant attention.

7.1 Epoch-Level Transitions

Currently, the transitions between SFT and PPO phases are implemented at the epoch level. Although this method has shown effectiveness, it may overlook the potential benefits of finer-grained control. Transitioning at the batch level could allow for more dynamic adjustments during training, potentially enhancing the overall performance and convergence speed of the model.

7.2 Lack of Mixture of Experts (MoE)

The current version of SYNTHFIX does not leverage a MoE framework, which could provide a more specialized approach to different aspects of code repair. By integrating RL and NLP as separate experts, each trained to handle distinct types of vulnerabilities, the framework may more effectively address the diverse challenges presented by various code structures and patterns.

7.3 Absence of Rigorous Mathematical Justification

Although SYNTHFIX employs an EM-like algorithm, the underlying mathematical justification is not as rigorously developed as it could be. A more formal mathematical explanation using EM principles would help in better understanding and potentially improving the model's ability to generalize across different codebases and types of vulnerabilities.

In conclusion, while SYNTHFIX provides a robust framework for automated code repair, these limitations should be addressed to further improve its performance and applicability.

References

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

Gabin An, Jinhan Kim, and Shin Yoo. 2018. Comparing line and ast granularity level for program repair using pyggi. In *Proceedings of the 4th International Workshop on Genetic Improvement Workshop*, pages 19–26.

Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel Briand. 2021. Reinforcement learning for test case prioritization. *IEEE Transactions on Software Engineering*, 48(8):2836–2856.

Dzmitry Bahdanau, Philemon Brakel, Kelvin Xu, Anirudh Goyal, Ryan Lowe, Joelle Pineau, Aaron Courville, and Yoshua Bengio. 2016. An actor-critic algorithm for sequence prediction. *arXiv preprint arXiv:1607.07086*.

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.

Zhenqiang Chen, Baowen Xu, and Jianjun Zhao. 2002. An overview of methods for dependence analysis of concurrent programs. *ACM Sigplan Notices*, 37(8):45–52.

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.

Pierluca D'Oro and Wojciech Jaśkowski. 2020. How to learn a useful critic? model-based action-gradient-estimator policy optimization. *Advances in Neural Information Processing Systems*, 33:313–324.

Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. 2012. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, part C (applications and reviews)*, 42(6):1291–1307.

Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. 2020. Synthesize, execute and debug: Learning to repair for neural program synthesis. *Advances in Neural Information Processing Systems*, 33:17685–17695.

Maya R Gupta, Yihua Chen, et al. 2011. Theory and use of the em algorithm. *Foundations and Trends® in Signal Processing*, 4(3):223–296.

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.

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.

Shujuan Jiang, Shengwu Zhou, Yuqin Shi, and Yuanpeng Jiang. 2006. Improving the preciseness of dependence analysis using exception analysis. In *2006 15th International Conference on Computing*, pages 277–282. 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.

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.

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.

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.

Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. Deepdelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 925–936.

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*.

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*.

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

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*.

Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal. 2022. Transformer-based language models for software vulnerability detection. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 481–496.

Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. 2022. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 129–143.

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

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.

CF Jeff Wu. 1983. On the convergence properties of the em algorithm. *The Annals of statistics*, pages 95–103.

Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 959–971.

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

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.