Hit The Bullseye On The First Shot: Improving LLMs Using Multi-Sample Self-Reward Feedback for Vulnerability Repair

Rui Jiao[1], Yue Zhang[2], Jinku Li[1,*], Jianfeng Ma[1]
[1]State Key Laboratory of Integrated Services Networks, School of Cyber Engineering, Xidian University
[2]Shandong University
jiaorui@stu.xidian.edu.cn, zyueinfosec@gmail.com, jkli@xidian.edu.cn, jfma@mail.xidian.edu.cn

***Abstract*—In recent years, large language models (LLMs) have emerged as powerful tools to assist developers in various coding tasks, including the challenging domain of vulnerability repair. While these models have demonstrated significant potential in generating patches for software vulnerabilities, current approaches often suffer from limitations in precision, requiring multiple attempts to produce accurate fixes. In this paper, we propose MUSSEL (*Mu*lti-*S*ample *Sel*f-Reward Feedback), a novel framework designed to address the issue of one-shot vulnerability patching. Inspired by insights from human learning mechanisms, our approach aims to enhance the efficiency and accuracy of LLMs in generating precise patches for software vulnerabilities. We introduce a multi-stage training process, beginning with supervised fine-tuning using domain-specific data to impart foundational knowledge in vulnerability repair to the LLM. Subsequently, we employ self-reward feedback learning to refine the model's patch generation capabilities, leveraging correct and incorrect patches iteratively to improve performance. We also introduce a novel prompt design tailored to better align with the capabilities of LLMs during inference. Our results demonstrate that MUSSEL consistently outperforms state-of-the-art solutions in one-shot queries. Notably, even with a small beam size, MUSSEL exhibits remarkable efficiency, requiring minimal GPU memory resources. Furthermore, MUSSEL's effectiveness across diverse CWEs underscores its significant security implications.**

***Index Terms*—Large Language Models, Vulnerability Repair.**

I. Introduction

Recently, a proliferation of Large Language Models (LLMs) has emerged to support coding tasks such as code generation and completion [16], [20], [29], [39], [43], [50], [61]. Among these, vulnerability repair has gained notable traction, with recent studies confirming LLMs' effectiveness in this domain. For example, Pearce et al. [37] evaluated multiple commercial and open-source LLMs across synthetic, hand-crafted, and real-world security bug scenarios, demonstrating their capability to successfully repair vulnerabilities.

However, LLMs' vulnerability repair capabilities remain constrained: unlike natural language tasks that accommodate varied outputs, program repair requires precise solutions. For instance, if generated patches alter specific variable names despite correct logic, compilation fails due to undefined variable errors. Consequently, the standard approach involves generating numerous potential patches that are then compiled, with success determined only by compilation outcomes. For example, Pearce et al. [37] exemplify this by generating up to 19,600 potential repair patches for 12 real-world project CVEs, with only 982 achieving successful repair.

Our approach aims to achieve one-shot vulnerability repair by drawing parallels between LLM training and human learning processes. We equip LLMs with repair capabilities through supervised fine-tuning (SFT, providing foundational knowledge) [11], then enhance performance via a special designed preference learning algorithm incorporating both correct and incorrect patches. This self-generated feedback loop improves LLMs' predict correct precision in complex vulnerability repairs, potentially enabling one-shot patch generation.

We design MUSSEL (**Mu**lti-**S**ample **Sel**f-Reward Feedback), a three-stage approach for vulnerability repair. First, we fine-tune the initial model on domain-specific data (vulnerable code and patches) through SFT, generating diverse candidate patch sequences of varying correctness. Second, we implement self-reward feedback fine-tuning using our designed novel Multiple Sample Reinforcement Direct Preference Optimization (MSR-DPO) algorithm, which optimizes the expected difference between correct and multiple erroneous generations from the first stage. We incorporate a vector of KL divergence [2] dynamic weights to prioritize specific non-preferred samples. Finally, we design an inference prompt that simplifies the repair task by consolidating various operations (ADD, $DELETE$, $MODIFY$) into a single $MODIFY$ operation, better aligning with LLM capabilities and addressing their classification limitations.

We evaluate our approach using a dataset of 5,800 C/C++ vulnerability and patch snippets from 1,754 large-scale projects, covering over 180 CWE types. Our experiments demonstrate MUSSEL's superior performance in one-shot vulnerability repair. With $ExtNum = 5$ (the number of extended non-preferred patches during training), MUSSEL achieves a BLEU score [35] of 53.66, substantially outperforming the state-of-the-art score of 29.32 [63]. On a deduplicated dataset with refined preprocessing, we reach a maximum BLEU score of 61.74. MUSSEL consistently surpasses baseline fine-tuning methods (SFT, PPO [44], and DPO [41]) across multiple met-

*Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00071

rics while requiring only 11.2GB of GPU memory—making it practical for consumer-grade GPUs with typical 24GB VRAM limitations. We further validate MUSSEL's effectiveness across diverse parameter settings and confirm the utility of our key techniques through comprehensive ablation studies.

In summary, our principal contributions are as follows:

- **Novel Problem of One-Shot Vulnerability Patching.** We address generating precise vulnerability patches in one shot, leveraging a training approach inspired by human learning to enable LLMs to fix vulnerabilities efficiently, unlike existing multi-attempt methods.
- **New Algorithm with Domain Insights.** We present the MSR-DPO algorithm within MUSSEL, an enhanced DPO algorithm tailored for vulnerability repair. The method advances one-shot vulnerability repair by leveraging MSR-DPO's dynamic KL-weighted optimization across multiple non-preferred samples, effectively overcoming the limitations of standard DPO in vulnerability repair scenarios, which rely on a single preference signal and thus lack the granularity required for precise vulnerability repair.
- **Superior Performance with Practical Implications.** Evaluations on C/C++ vulnerabilities show that MUSSEL outperforms state-of-the-art solutions in one-shot queries, achieving remarkable scores even with small $ExtNum$ size while using minimal GPU memory resources. MUSSEL proves effective across various CWEs, demonstrating significant security implications.

Code Availability. We release MUSSEL at: https://github.com/XDU-SysSec/Mussel.

II. Background and Related Work

A. Core Concepts of LLMs

Architecture of LLMs. LLMs primarily employ Transformer-based [48] architectures that can be classified as: (1) Encoder-only models (e.g., BERT [10], CodeBERT [15]), which excel at classification but struggle with generation; (2) Decoder-only models (e.g., GPT [54], Codex [40], LLaMA [47]), which perform autoregressive sequence generation [46]; and (3) Encoder-decoder models (e.g., CodeT5 [50], BART [26]), which leverage both encoder and decoder components. For security applications, particularly vulnerability repairing tasks, general-purpose LLMs require specialized training on code repositories to effectively analyze programming languages and identify vulnerability patterns.

Pre-Training and Fine-Tuning. LLM training involves two main phases: Pre-Training on vast datasets for general language understanding, and Fine-Tuning on instruction-response pairs for task-specific applications. LLMs with only pre-training typically show limited performance on complex tasks like code generation or bug-fixing, necessitating fine-tuning to enhance capabilities. Methods like Supervised Fine-Tuning (SFT) [11] and Reinforcement Learning from Human Feedback (RLHF) [34] are commonly employed to improve performance. Fine-tuning for code vulnerability fixes differs from general-purpose tuning in two key aspects: it requires task-specific adjustments prioritizing code syntax, semantics, and security vulnerability understanding, and employs code-specific evaluation metrics rather than general language understanding benchmarks.

TABLE I: Comparison of SFT, RLHF, and RLAIF.

Methods	Stage I	Stage II	
	SFT	RLHF	RLAIF
Training Data			
Human Annotation	✓	✗	✗
Human Feedback	✗	✓	✗
AI-generated Feedback	✗	✗	✓
Training Method			
Supervised Learning	✓	✗	✗
Reinforcement Learning	✗	✓	✓
Simplified RL Process	✗	✗	✓

B. Comparison of SFT, RLHF, and RLAIF

Supervised Fine-Tuning (SFT), Reinforcement Learning from Human Feedback (RLHF), and Reinforcement Learning from AI Feedback (RLAIF) [9] are three fine-tuning methods that enhance LLM performance, utilized in various works [27], [34], [45], [49]. These methods can be used in conjunction with each other.

SFT. This approach uses high-quality downstream supervision data to fine-tune model parameters, employing input-output pairs to minimize losses between generated and target sequences. Recent approaches like LoRA [21] maintain model quality without increasing inference latency, making SFT cost-effective. In the program repairing task, VRepair [7] employs a transformer model pre-trained on bug fixes and fine-tuned on vulnerability data, while Zhang et al. [59] demonstrate that SFT enables LLMs to transition from code generation to effective bug fixing.

RLHF. This method [8], [22] uses human-annotated preference data to train reward models, employing reinforcement learning to align outputs with human expectations. Ouyang et al. [34] utilize Proximal Policy Optimization (PPO) [44] to align LLMs by training an SFT policy model, evaluating it with human feedback, and adjusting outputs using PPO. CodeDPO [57] aims at code generation via external tests, using mutual verification where initial self-verification scores for code snippets and tests are refined iteratively, enhancing LLM code generation. Focused-DPO [58] aims to correct localized errors within otherwise functional code.

RLAIF. This approach replaces human feedback with AI-generated feedback. Bai et al. [1] establishes the LLM-as-a-Judge paradigm, while Chen et al. [6] demonstrated an Iterative DPO-like framework that eliminates reward models. For Code-LLMs, Cui et al. [9] combined RLAIF with DPO to optimize models like CodeLlama-7B [43], showing RLAIF's effectiveness in generating code aligned with coding preferences.

We compare these methods in Table I. SFT uses human-annotated data under supervised learning with lower resource

requirements. RLHF employs human feedback through reinforcement learning, requiring significant resources including reward model training. RLAIF substitutes AI-generated responses for human feedback, typically using reinforcement learning while consuming fewer resources and enabling more precise control over output strategies for complex preferences. As mentioned earlier, some approaches (e.g., CodeDPO [57], Focused-DPO [58]) excel in code generation but differ from our method, which targets failures in security contexts. MUSSEL improves one-shot vulnerability repair, introducing MSR-DPO, which uses dynamic KL-weighted optimization with multiple samples, surpassing standard DPO's single-signal limitation for precise repairs.

C. Using LLMs for Vulnerability Repair

LLMs have demonstrated effectiveness in vulnerability repair. Fu et al. [17] use a T5 model [50] for C code vulnerability repairs, while Pearce et al. [37] show LLMs' efficacy on both synthetic and real-world vulnerability data. These models, based on RNN [18] or Transformer architectures, treat repair as a sequence generation task similar to machine translation. The repair process involves two key steps: (i) Training model for vector representation encodes vulnerable code geometrically within the vector space during training. Input code undergoes Natural Language Processing (NLP) techniques like word embedding [32] in a masked pre-training manner to acquire latent space representations, enhanced with positional encoding across progressive layers. (ii) Beam Search [54] for patches generation occurs during inference (with fixed model parameters). The decoder's final layer produces high-dimensional vectors, and beam Search sequentially identifies probable tokens to form repair patches, with candidate token quantity varying based on beam size.

III. Motivation and Problem Statement

A. Motivation

LLMs navigate through high-dimensional vectors to select tokens probabilistically, which enhances creativity but presents challenges for vulnerability repair. In Figure 1, we demonstrate how current models like GPT-4 [33] and CodeLlama [43] generate numerous candidate patches before producing a valid solution (Patch ❹). The alternatives often contain redundant self-references (Patch ❶), syntactic errors (Patch ❷), or inappropriate terminology (Patch ❸). Without error feedback mechanisms, models struggle to identify and correct these undesirable characteristics.

Increasing beam search parameters (*BeamNum*) to produce more candidate patches demands prohibitive computational resources. As shown in Figure 2, our experiments on an *NVIDIA RTX-3090 GPU* reveal that higher *BeamNum* values significantly increase both inference time and memory usage, with out-of-memory errors occurring beyond 20 returned sequences. Furthermore, zero-shot repair evaluations of GPT-3.5 and GPT-4o yield disappointing BLEU scores [35] of only 8.81 and 9.74, respectively (Table II), highlighting the inefficiency of current approaches to vulnerability repair. Therefore, it is crucial to develop methods that enhance the quality of one-shot patch generation.

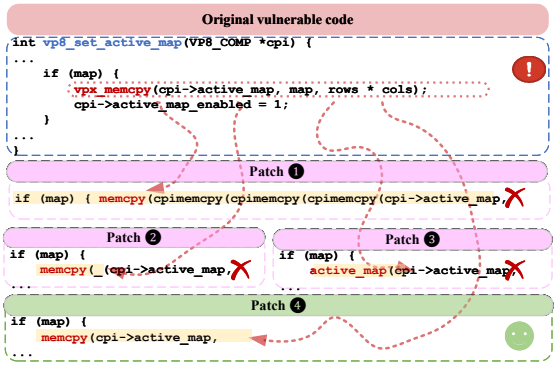

Fig. 1: The illustration of correct and incorrect patches.

B. Problem Statement

Formally, the task of vulnerability repair can be treated as crafting code sequence snippets based on input prompts (which can be a partially vulnerable program). Such a problem can be approached by framing it as a decision-making issue in a finite-horizon Markov Decision Process (MDP) [14]. By employing Deep Reinforcement Learning (DRL) techniques, the model can learn to generate sequences of code fixes to repair vulnerabilities. Specifically, the model takes an input code snippet containing vulnerabilities, denoted as X. The objective is to produce a repair sequence within a specified timestep, represented as $W = (w_1, w_2, ..., w_T)$ where each token $w_t \in \mathbb{V}$ is drawn from a predefined vocabulary $\mathbb{V}$ of code tokens. At each time step t, w_t is sampled from a softmax [4] distribution parameterized by $Linear(s_t)$ network, where s_t signifies the environmental state of the decoder at time step t. During the training process, the cross-entropy loss between the generated repair sequence and the target correct sequence serves as the optimization objective for model gradient descent. This iterative approach iteratively refines the generation policy to yield accurate repair patches. Define the set $W_{correct} = (w_1, w_2, \ldots, w_T)$ as the intended correct sequence for repair; the optimization target is expressed as Equation 1:

$$\begin{aligned} \mathcal{L}_{ce}(\theta) &= -\sum_{t} \log p_\theta(W_{\text{correct}}|X) \\ &= -\sum_{t} \log p_\theta(w_t|w_{1:t-1}, X) \end{aligned} \tag{1}$$

Where, θ indicates the parameters within the neural network model, while p_θ indicates the conditional probability of producing the desired sequence while given the input from the original code snippet.

IV. MUSSEL: Idea and Design

A. Key Idea

Our key idea draws from parallels between LLM training and human learning mechanisms. Just as students maintain

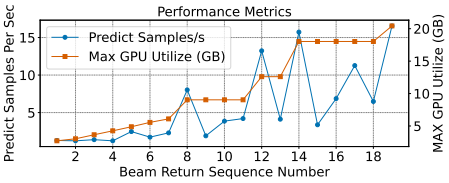

Fig. 2: Resource cost in LLM Beam Search.

an "error notebook" to review mistakes and correct solutions, LLMs can benefit from analyzing their own errors in vulnerability patching tasks. The process begins with SFT to establish baseline vulnerability-fixing capabilities. We then generate multiple candidate patch sequences using this model, which may be either correct or contain errors. We determine correctness using reference patches collected from GitHub fix commits, allowing us to label both correct and incorrect examples. These labeled patches form our second-phase learning corpus—analogous to an "error notebook." We apply a specially designed preference learning algorithm to this corpus, enabling the model to learn from its own mistakes. This self-generated feedback loop helps LLMs better understand complex vulnerability-repair tasks, identify generative weaknesses, and improve performance, ultimately enhancing the quality of our one-shot patch generation.

B. MUSSEL Design

As illustrated in Figure 3, MUSSEL employs a three-stage process. **Stage (I)** involves SFT for Multi-Sample Generation. In this stage, a model undergoes fine-tuning using domain-specific data to improve adaptability for vulnerability repair. We then use its improved abilities to set a baseline for our experiments. With the enhanced model, we generate a range of patch sequences, varying from flawless ones to those with errors or nonsensical outputs. In **Stage (II)**, Self-Reward Feedback Fine-Tuning is introduced to enhance the traditional reinforcement learning from AI feedback method. The goal is to align the output probability distribution of generated sequences with preferred data while moving away from non-preferred data. To address challenges, the Multiple Sample Reinforcement Direct Preference Optimization (MSR-DPO) algorithm is introduced. It optimizes the expected difference between correct and multiple erroneous generations, updating policy model parameters for improved generative performance. Meanwhile, the dynamic KL (Kullback-Leibler Divergence) [2] weight is continuously adjusted during training to modulate the emphasis on multiple non-preferred samples. In **Stage (III)**, Prompt Engineering is introduced. Rather than using three different types of operations for code repair (ADD, $DELETE$, $MODIFY$), the approach is refined to retain only one $MODIFY$ operation. This simplification aims to better suit the capabilities of generative language models, improving their ability to generate patches.

Stage (I) - SFT for Multi-Sample Generation. During the phase of supervised fine-tuning, a series of *"Incorrect-Correct Patch"* data pairs, denoted as $S_{sft} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, serve as the supervisory data (where x_i represents the original incorrect code patch, and y_i represents the labelled correct code patch). The model undergoes supervised fine-tuning using domain-specific data with known correct patch labels (gathered and labelled from sources such as GitHub fix commits), which aims to enhance the model's adaptability. In our context, adaptability refers to the model's capacity to address the task of C/C++ language repair, as well as to understand the design patterns used in data handling. This evolution enables the language model to transition from a merely conversational tool to a powerful instrument capable of generating patches for specific data structural flaws. To be more specific, the formal definition is as Equation 2:

$$\mathcal{L}_{SFT} = -\frac{1}{n}\sum_{i=1}^{n} \log P(y_i|x_i, \theta) \tag{2}$$

In Equation 2, $P(y_i|x_i, \theta)$ represents the conditional probability of the model outputting a patch under the model parameters θ, given an input of a vulnerable code snippet. The objective of training optimization is to adjust the model parameters to minimize the sum of the cross-entropy loss $\mathcal{L}_{SFT}$ for each data pair, ultimately leading to an optimal policy model π_{SFT}.

Subsequently, we leverage its enhanced capabilities to establish a foundational benchmark for our experimentation. Using this SFT-enhanced model, we embark on generating an array of candidate patch sequences. These sequences vary widely, ranging from potentially flawless renditions to those plagued with grammatical errors or producing nonsensical outputs. Determining the correctness of these generated patches is straightforward, as we can simply compare them with patches collected from various repositories, such as GitHub fix commits. Through this wealth of data, we are empowered not only to identify and label the correct patches but also to pinpoint and categorize the incorrect ones, thereby enriching our training dataset with a spectrum of examples for comprehensive learning.

Stage (II) - Self-Reward Feedback Fine-Tuning for One-Shot Patch Creation. To comprehend our Self-Reward Feedback Fine-Tuning, we first need to delve into common practices of traditional models, as our algorithm builds upon these models using our domain insights. Traditional models utilize Reinforcement Learning from Human Feedback (RLHF) after SFT to ensure expected output. Particularly, RLHF involves annotators selecting preferred (y_w) and non-preferred (y_l) outputs (in our context, they are the correct and incorrect patches) among multiple candidates (note that in the traditional model, the preferred and non-preferred outputs comprise just one pair). Given a collection of triplet data sets (x, y_w, y_l) denoted as **P**, a reference policy model π_{ref} (which is usually the fixed-parameter LLM itself), and a generative model π_θ refined via SFT, the parameters of the policy model π_θ are optimized through the Direct Preference Optimization (DPO) algorithm (as shown in Equation 3).

This optimization aims to align the output probability

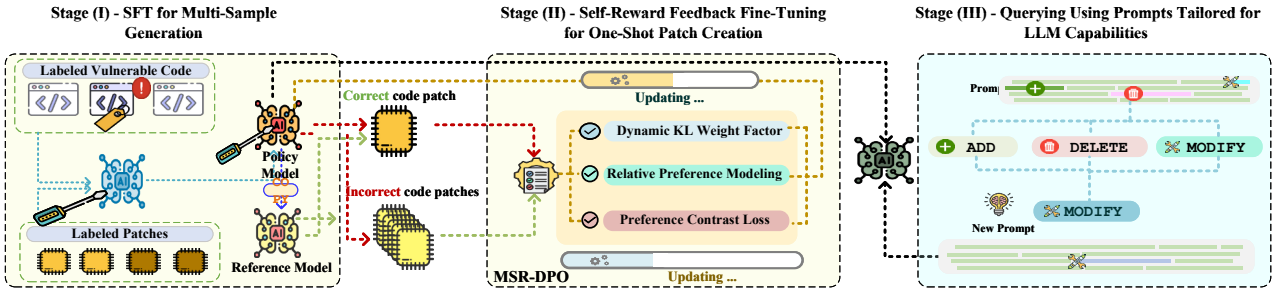

Fig. 3: The workflow of MUSSEL.

distribution of generated sequences (those generated by the generative model π_θ) with the preferred data (which is generated by the reference policy model π_{ref}) while moving away from the non-preferred data. The reference model prevents potential issues like overly long or repetitive sequences, which might meet reward optimization goals but deviate significantly from the original model's output distribution. It calculates the distance between preferred (y_w) and non-preferred (y_l) data conditional probabilities, with the denominator's ref model transforming this into a relative measure to prevent extreme optimization outcomes.

$$\mathcal{L}_{\text{DPO}}(\pi_\theta;\pi_{\text{ref}}) = -\,\mathbb{E}_{(x,y_w,y_l)\sim\mathbf{P}}\Big[\sigma\log\Big(\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)}\Big) - \sigma\log\Big(\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\Big)\Big] \tag{3}$$

The DPO algorithm (Equation 3) is effective for preference optimization but struggles with vulnerability repair. This algorithm accommodates only one non-preferred output per training instance alongside a single preferred data point. Yet, vulnerability repair tasks demand a considerable array of non-ideal outputs to serve as non-preferred data for strategic optimization (as the desired output is singular). Consequently, the algorithm proves inadequate for addressing these particular challenges. Therefore, building upon such an algorithm, we adapt and enhance the algorithm specifically for vulnerability repair tasks by introducing the MSR-DPO (Multiple Sample Reinforcement Direct Preference Optimization) algorithm. The $Loss$ definition for the optimization objective of the MSR-DPO algorithm is shown as Equation 4:

$$\mathcal{L}_{\text{MSR-DPO}}(\pi_\theta;\pi_{\text{ref}}) = -\,\mathbb{E}_{(x,y_w,y_l)\sim\mathbf{D}}\Big[\log\sigma\Big(\beta\cdot\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\cdot\sum_{i=1}^{k}\gamma_i\cdot\log\frac{\pi_\theta(y_{l_i}|x)}{\pi_{\text{ref}}(y_{l_i}|x)} + \lambda\cdot\mathcal{L}_{SFT}\Big)\Big] \tag{4}$$

γ_i is defined as Equation 5:

$$\gamma_i = \frac{\exp\left(D_{\text{KL}}\left(\left(\pi_\theta(y_{l_i}\mid x)\right)\parallel\left(\pi_\theta(y_w\mid x)\right)\right)\right)}{\sum_{i=1}^{k}\exp\left(D_{\text{KL}}\left(\left(\pi_\theta(y_{l_i}\mid x)\right)\parallel\left(\pi_\theta(y_w\mid x)\right)\right)\right)} \tag{5}$$

Note that Equation 4 requires two models: π_θ, which is the policy model, and π_{ref}, the reference model. The objective of the algorithm is to optimize the expected difference between correct generation and multiple erroneous generations, thereby updating the parameters of the policy model to achieve improved generative performance. Within the expectation notation, y_{l_i} represents multiple non-preferred samples, while y_w denotes the anticipated correct samples, both of which are sampled from the generation of π_θ. The term σ in the equation refers to the activation function, β is a hyper-parameter controlling the extent of optimization. γ_i is a dynamic non-preferred weight, used to control the importance of multiple non-preferred data points during the training process. λ represents the weight parameters, denoting the SFT fusion adaptation weight, respectively. The first term of Equation 4 calculates the relative distance in the probability distribution of preferred data outputs between the policy model and the reference model (i.e., relative reference modeling. This is similar to that of the original DPO). The second term of the equation calculates the relative distance of outputs for multiple non-preferred data (i.e., preference contrast loss). The final term of the equation is introduced to incorporate the SFT fusion adaptation loss, which also serves to prevent overfitting due to preference optimization. We next explain our design intuition:

Dynamic KL Weight Factor: During the preference optimization training phase, the model is simultaneously provided with a desired preferred sample and multiple non-preferred samples. As the training progress, the probability distribution of the sequences generated by the model and the discrepancies with the multiple non-preferred samples continually evolve. To enable the model to fully exploit these non-preferred samples and dynamically recognize these discrepancies during training, we introduce a *"Dynamic KL Weight Factor"* for the non-preferred samples. This factor γ_i continuously computes the KL divergence between the preferred sample and multiple non-preferred samples throughout the training process, thereby adjusting the importance of the losses associated with different non-preferred samples. This approach aims to fully leverage the multiple non-preferred samples for optimal learning.

Relative Preference Modeling: Relative Preference Modeling is essential because it introduces a mechanism to prevent models from exploiting reward systems in unintended ways,

by focusing on the relative probabilities of different outcomes rather than solely on maximizing rewards. To be more specific, the binary preference features an optimal preference conditional probability expressed as follows: $p^*(y_1 \succ y_2 \mid x) = \sigma(r^*(x, y_1) - r^*(x, y_2))$, where σ denotes the logistic sigmoid function, and r^* represents the ground-truth reward linked to the optimal policy π^*. Solely focusing on the reward model within the policy framework might lead to the reward hacking issue (i.e., models exploit the reward system in unintended ways to maximize its reward without actually accomplishing the intended task). Consequently, we incorporate the relative preference modeling approach referenced in the DPO algorithm [41]. For y_w and y_{l_i} , we have established relative deviations. By dividing by the reference model and then taking the logarithm, we obtained relative probabilities in the form of $\log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)}$ and $\log \frac{\pi_\theta(y_{l_i}|x)}{\pi_{\text{ref}}(y_{l_i}|x)}$. This relative probability constrains the policy optimization of the policy model using the reference model, preventing it from generating sequences whose probability distributions deviate significantly from the expected distributions.

Preference Contrast Loss: Preference Contrast Loss measures the difference between relative preference values of preferred and non-preferred data. It guides the policy model π_θ toward expected preferences by maximizing this difference, with varying loss magnitudes providing dynamic weights per training iteration—effectively revisiting an "error notebook" multiple times. During training, π_θ maximizes relative reward differences while adapting to evolving $RewardMargin$. Training epochs generate non-preference losses of different magnitudes, with the model computing discrepancies between preference and non-preference outputs, prioritizing closer probability distributions. This creates varied attention to non-preferred samples each epoch, similar to focusing on different error types iteratively. The $Loss$ formulation enables multiple non-preferred objectives in a single optimization step, and it is critical for tasks requiring extensive repair attempts by simultaneously optimizing against several undesired outcomes.

Stage (III) - Querying Using Prompts Tailored for LLM Capabilities. When fixing vulnerable code, LLM needs to modify the vulnerable parts based on existing code snippets. However, this modification process may involve multiple actions such as adding, replacing, or removing elements in the code snippets. The previous approach [7], the authors initially defined three types of operations: ADD, $DELETE$, and $MODIFY$. Although this method performs well with previous encoder-based language models, it is inherently more suitable for classification tasks than for autoregressive tasks. With three types of operation, it adds barriers for LLMs, and the overly complex original labeling hampers the model's ability to generate better repair sequences. Therefore, we refine the data preprocessing strategy to suit LLMs better. Instead of using three different operations, we modify the repair modes to retain only one $MODIFY$ operation. We leverage Figure 4 as an example to illustrate the methodology employed in our study. The "*Original Vulnerable Code*" indicates that our approach

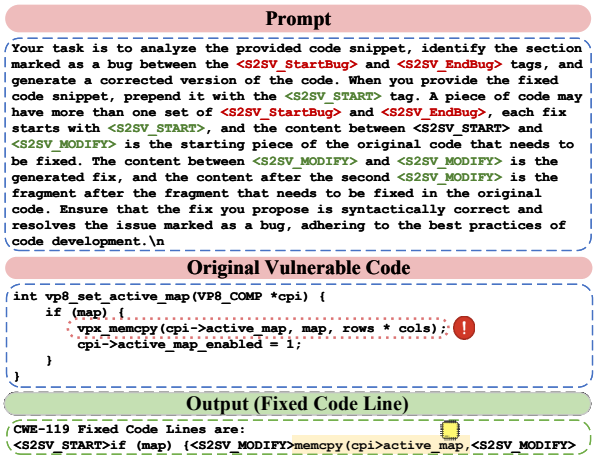

Fig. 4: An example of fixing a vulnerability in a MUSSEL designed prompt: the input consists of the prompt text, symbol explanations, and the vulnerable code snippet; the output is the corrected code snippet without the vulnerability.

utilized <S2SV_StartBug> and <S2SV_EndBug> tags to denote the locations of vulnerabilities. The "*Output*" shows model output, <S2SV_START> marks the beginning label of the repair, and the content between <S2SV_MODIFY> includes the context of the repair sequence, typically containing several tokens preceding the fragment that needs fixing. The correct repair sequence is placed between two <S2SV_MODIFY> tags. Additionally, the task of the model is to generate the entire method's repair sequence with a one-shot query, which usually contains multiple repair sub-fragments. Note that the above-crafted prompt with the vulnerable code and its fixes is also used during training.

V. Experiment

A. Experimental Setup

Dataset. We use the dataset from VulMaster [63], specifically CVEFixes [3] and Big-Vul [12], consisting of C/C++ vulnerabilities with corresponding fixes from 1,754 real-world projects. The dataset contains 5,800 entries covering over 180 CWE types, split into 3,872 training, 316 validation, and 1,612 test pairs. Additionally, for the pretraining dataset used in prior works [7], [63], we rigorously address potential data leakage issues (corresponding to RQ2). Specifically, we applied a string-matching Python script to the 500,000-sample pretraining dataset to identify overlaps with the final test set. During comparison, we removed non-semantic characters such as whitespace, newlines, and special symbols to retain only meaningful content, ensuring comprehensive detection and mitigating the influence of formatting discrepancies. As a result, we identified and excluded 243 leaked samples from the pretraining dataset.

Experiment Environment. Our approach utilizes *Llama-Factory* [62] with *Pytorch* [36] and *Transformers* [51] as core packages. For reinforcement learning, we employed the *Trans-*

TABLE II: Performance comparison of different approaches.

Type	Model	BLEU	EM
Pre-Trained	CodeBERT	3.96	3.97
	PolyCoder	4.35	3.52
	CodeT5	3.92	10.23
Task-Specific	VRepair	24.29	8.91
	VulMaster (SOTA)	**29.32**	**20.00**
LLM (Closed-Source)	GPT-3.5	8.81	3.64
	GPT-4o	9.74	5.31
LLM-SFT	StableCode-3B	35.82	4.21
	CodeGemma-3B	38.87	6.93
	DeepSeek-Coder-1.3B	47.12	16.09
	DeepSeek-R1-1.5B	48.91	19.66
	Qwen3-1.7B	48.18	20.16
MSR-DPO	MUSSEL-DeepSeek-Coder-1.3B	**51.31**	**24.25**
	MUSSEL-DeepSeek-R1-1.5B	**53.19**	**26.67**
	MUSSEL-Qwen3-1.7B	**53.66**	**27.66**

former Reinforcement Learning (TRL) module. Experiments were conducted on two *NVIDIA A100-80G* GPUs and four *RTX 3090-24G* machines, with fine-tuning on *A100s* using LoRA [21] and inference on *RTX 3090* devices.

Experiment Parameters. In stage (I), we use the LoRA adapter with $lora\_target$ set to "all," learning rate of 5e-5, and $lora\_rank$ of 10. Training typically exceeded 8 epochs, based on validation results. For stage (II), we perform full-parameter adapter fine-tuning with $Beta\_factor$ at 0.2, Ftx_factor at 0.5, and typically over 14 training epochs. We implement FP-16 [25] precision to conserve GPU resources.

B. Performance Evaluation

RQ1. How does MUSSEL compare to prior works in terms of performance?

We first evaluated the patch correctness related to vulnerability repair using the previously mentioned dataset, employing the same preprocessing method as in prior works. In this scenario, our primary evaluation metrics are the BLEU and EM scores, which measure the degree of alignment between the generated and target ground-truth sequences. A higher BLEU/EM score signifies that the generated repair sequences more closely resemble the target sequences (i.e., labeled correct patches). Several studies [7], [63] have proposed methods for vulnerability repair, achieving commendable results. These pioneering approaches have established a robust foundation for subsequent research in automated vulnerability repair. However, we identified that due to human negligence, data leakage issues were discovered in training datasets used in prior studies. The 243 instances leaked from the 500,000 pre-training dataset (accounting for 15.07% of the 1,612 test set) and 126 instances leaked from the 3,872 SFT-Training dataset (accounting for 7.82% of the 1,612 test set), culminating in a total leakage of 22.89%. This introduces the potential for test set label overfitting during continuous training, which may lead to discrepancies between BLEU and EM results (high EM with low BLEU). The issue of dataset leakage results in a diminished reference significance of the experimental

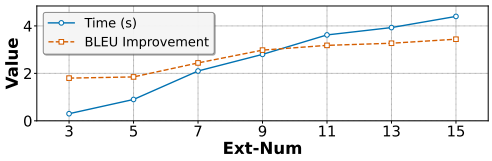

Fig. 5: Line chart of the relationship between the patch generation time and the number of non-preferred samples (Ext-Num) with original data format.

results. However, to illustrate its impact on the method we proposed, we conduct experiments under the same conditions. The *DeepSeek-Coder-1.3B* [20], *DeepSeek-R1-1.5B* [19], and *Qwen3-1.7B* [52] models served as the foundational models for executing MUSSEL training.

As shown in Table II, MUSSEL with MSR-DPO algorithm markedly improved the quality of the generated repair sequences, achieving the highest BLEU/EM scores. For example, *MUSSEL-DeepSeek-Qwen3-1.7B* yielded a BLEU score of 53.66, and an EM score of 27.66. Compared to the state-of-the-art (SOTA) [63], BLEU improved by 24.34, and EM improved by 7.66. It is also evident that the performance of Pre-Trained Models (e.g., *CodeT5*) without fine-tuning is suboptimal, with BLEU scores ranging from 3-5. Task-Specific LLMs without fine-tuning perform better (achieving a BLEU score of 29.32 for the generated repair sequences), yet remain inferior to those LLMs that have undergone fine-tuning. Closed-source LLMs (e.g., *GPTs*) achieve BLEU scores of 8-10. After SFT, *StableCode-3B* [39] reached a BLEU score of 35.82. The *CodeGemma-3B* model [60], achieved a BLEU score of 38.87. *Deepseek-Coder-1.3B* [20] with SFT attained a BLEU score of 47.12. The latest models, *DeepSeek-R1-1.5B* [19] and *Qwen3-1.7B* [52], demonstrate significant improvements in EM scores over previous models.

Notably, as mentioned before, due to issues of dataset leakage, the results show a deviation in two metrics compared to prior works when not applied with deduplication. The prior SOTA achieved a relatively high EM score of 20.00, but the BLEU score for the generated response sequences was notably low, at only 29.32. It is important to note that the previously mentioned dataset leakage percentage was 22.89%, which closely aligns with the results observed during actual testing (20.00 EM score). This renders experiments on this dataset lacking in reliable reference value. Consequently, we have performed deduplication on the dataset and conducted experimental comparisons with mainstream fine-tuning methods for LLMs in subsequent sections, to further illustrate the improvements of our method within the fine-tuning paradigm.

RQ2. What are the performance differences of MUSSEL under various data preprocessing methods?

In this experiment, we evaluated our method against alternatives using different preprocessing approaches. We removed leaked samples from the training set and compared perfor-

TABLE III: Experiment results of original data format.

Type	ExtNum	BLEU	Rougle-L	CodeBLEU	SyntaxMatchScore	DataflowMatchScore	EM
ORI-Model	/	10.34	16.75	18.07	20.76	49.00	0.00
SFT (BASE)	/	38.76	49.07	27.34	39.21	42.29	12.71
SFT+PPO	/	33.18 (−5.58)	38.70 (−10.37)	24.53 (−2.81)	33.95 (−5.26)	45.34 (-3.05)	8.79 (−3.92)
SFT+DPO	/	37.37 (−1.39)	44.69 (−4.38)	27.3 (−0.04)	37.46 (−1.75)	43.82 (+1.53)	12.13 (−0.58)
MUSSEL	ext-3	40.56 (+1.8)	49.25 (+0.18)	28.78 (+1.44)	39.84 (+0.63)	45.8 (+3.51)	12.15 (−0.56)
	ext-5	40.61 (+1.85)	49.37 (+0.3)	29.74 (+2.4)	41.03 (+1.82)	46.31 (+4.02)	12.28 (−0.43)
	ext-7	41.2 (+2.44)	49.62 (+0.55)	29.23 (+1.89)	40.18 (+0.97)	46.56 (+4.27)	12.15 (−0.56)
	ext-9	41.74 (+2.98)	50.07 (+1.00)	29.79 (+2.45)	41.08 (+1.87)	47.01 (+4.72)	12.90 (+0.19)
△ MUSSEL	ext-9	(+4.37)	(+5.38)	(+2.49)	(+3.62)	(+3.19)	(+0.77)

mance improvements over mainstream fine-tuning techniques (SFT, PPO, DPO). Since VulMaster [63] only provides the model without pretraining code, we couldn't retrain it after removing leaked data and thus excluded it from comparisons. Additionally, we tested a preprocessing method specifically designed for Decoder-Only LLMs using both Full Parameters and LoRA Fine-Tuning.

(I) Comparison with Alternative Preprocessing. We addressed the issue of dataset leakage by deduplication and validated the improvements our method offers in fine-tuning LLMs compared to mainstream baseline methods. As shown in Table III, we present the comparative results of our method. The baselines we used are the widely adopted SFT method and the two-stage standard method of SFT followed by PPO/DPO. In the table, ORI-Model represents the original model, where we used *Deepseek-Coder-1.3B* [20] as the base model. MUSSEL represents our method, and *ExtNum* denotes the number of non-preferred samples used in training (corresponding to the non-preferred term in Equation 4). Both SyntaxMatchScore and DataflowMatchScore were computed using the corresponding modules in the CodeBLEU [42] package, reflecting the syntactic and semantic effectiveness of the output results. It is evident that compared to the original model ORI-Model, SFT significantly enhances its performance in specific vulnerability repair tasks. Metrics reflecting sequence similarity, such as BLEU, rising from 10.34 to 38.76, and the EM score rising from 0 to 12.71. This is attributed to SFT's ability to train LLMs to output in specific formats, aligning more closely with the label's format (similar to instruction fine-tuning in QA tasks [38], i.e., Question-Answering). In traditional natural language processing tasks, the PPO/DPO fine-tuning method following SFT can align the output of LLMs to better meet expected styles. But in the vulnerability repair task, it is observed that after PPO/DPO, there is a slight decrease in most metrics. Our MSR-DPO method incorporates domain-specific improvements for vulnerability repair, effectively balancing alignment and self-feedback. Performance improves as we increase non-preferred samples (*ExtNum*), with significant gains at ext-9 compared to standard DPO. The syntax matching score notably increases by 3.62, confirming our expectation that the model implicitly learns syntactic knowledge through negative examples.

Moreover, as non-preferred sampling incurs a non-negligible time cost—approximately 0.3 seconds per sample on *RTX-3090*. As shown in Figure 5, we evaluate our method using 3 to 15 non-preferred samples to balance sampling time and repair effectiveness. The trade-off analysis shows that the optimal convergence point between sampling time and improvement performance (△BLEU) occurs at around 9 samples. Based on this, we select a maximum of 9 non-preferred samples (ext-9) as the optimal configuration.

Additionally, our comprehensive evaluation reveals other insights into model training trade-offs. While our MUSSEL achieves superior BLEU/EM scores, we observed that standard SFT can inadvertently degrade certain code understanding capabilities preserved during pre-training, an effect intrinsic to SFT itself rather than a consequence of MUSSEL. Specifically, SFT may prioritize syntactic correctness over preserving underlying computational logic, leading to semantically different but syntactically correct code [13]. Our approach demonstrates meaningful progress in mitigating such capability losses: while standard SFT shows DataflowMatchScore degradation, our method achieves +3.19 improvement, representing a favorable trade-off where substantial gains in instruction following (0.00→12.90 EM) and code quality (10.34→41.74 BLEU) come with better preservation of semantic understanding.

(II) Our Preprocessing Method. Considering that the previous data preprocessing methods were not specifically designed for Decoder-Only architecture LLMs, we conducted comparative experiments with specially designed data processing approaches. In this section, we compare Full Parameter Fine-Tuning [31] with LoRA Parameter Fine-Tuning [21], setting the non-preferred samples ext from 3 to 9. The baselines used for comparison remain the widely adopted SFT, PPO, and DPO. As shown in Table IV, the improvements achieved by our method are presented, with MUSSEL representing the results of our approach. In Full Parameters Fine-Tuning, our method achieved the best BLEU score of 63.96 (+5.29), EM score of 13.17 (+2.96); even in LoRA Fine-Tuning, our method brought about significant improvements over the baseline methods. Similarly, both fine-tuning approaches enhanced grammatical metrics compared to the baselines, further corroborating the previously mentioned concept of implicitly learning syntactic knowledge.

TABLE IV: Experiment results of MUSSEL designed data format.

	ExtNum	BLEU	Rouge-L	CodeBLEU	SyntaxMatchScore	DataflowMatchScore	EM
ORI-Model	/	9.47	16.94	22.73	29.37	45.71	0.00
**** Full Params Fine-Tuning***							
SFT (BASE)	/	58.67	61.74	27.19	40.45	39.77	10.21
SFT+PPO	/	56.26 (-2.41)	58.38 (-3.36)	24.93 (-2.26)	35.28 (-5.17)	36.62 (-3.15)	7.71 (-2.50)
SFT+DPO	/	59.75 (+1.08)	62.91 (+1.17)	29.20 (+2.01)	42.33 (+1.88)	40.83 (+1.06)	11.29 (+1.08)
	ext-3	60.25 (+1.58)	63.13 (+1.39)	29.47 (+2.28)	42.60 (+2.15)	40.84 (+1.07)	11.34 (+1.13)
MUSSEL	ext-5	61.74 (+3.07)	64.44 (+2.70)	29.97 (+2.78)	43.14 (+2.69)	40.98 (+1.21)	11.64 (+1.43)
(MSR-DPO)	ext-7	62.87 (+4.20)	65.21 (+3.47)	30.72 (+3.53)	44.29 (+3.84)	42.84 (+3.07)	12.86 (+2.65)
	ext-9	63.96 (+5.29)	66.50 (+4.76)	31.24 (+4.05)	45.27 (+4.82)	43.26 (+3.49)	13.17 (+2.96)
**** LoRA Fine-Tuning***							
SFT (BASE)	/	48.96	53.67	22.11	33.84	36.64	1.25
SFT+PPO	/	46.22 (-2.74)	49.98 (-3.69)	18.65 (-3.46)	27.89 (-5.95)	34.55 (-2.09)	0.91 (-0.34)
SFT+DPO	/	52.63 (+3.67)	56.11 (+2.44)	22.60 (+0.49)	33.92 (+0.08)	36.09 (-0.55)	4.03 (+2.78)
	ext-3	53.84 (+4.88)	56.92 (+3.25)	22.97 (+0.86)	34.47 (+0.63)	37.25 (+0.61)	4.27 (+3.02)
MUSSEL	ext-5	54.75 (+5.79)	58.06 (+4.39)	23.40 (+1.29)	35.08 (+1.24)	38.51 (+1.87)	4.65 (+3.40)
(MSR-DPO)	ext-7	56.28 (+7.32)	59.82 (+6.15)	24.15 (+2.04)	36.70 (+2.86)	40.61 (+3.97)	5.73 (+4.48)
	ext-9	57.84 (+8.88)	61.29 (+7.62)	24.96 (+2.85)	37.79 (+3.95)	41.85 (+5.21)	6.34 (+5.09)

(III) Comparison with Full-Code Generation Repair. To further demonstrate the necessity of fault localization and the generalizability of MUSSEL, we perform additional experiments from three aspects: efficiency, effectiveness, and generalizability. The experimental results indicate that MUSSEL with fault localization enhances repair effectiveness while reducing repair time compared to MUSSEL for full-code generation, resulting in faster and more reliable software repair.

a) Efficiency: We conduct an average time evaluation over 100 generations on MUSSEL for vulnerability code generation (which requires fault localization) and full-code generation, respectively. The results shown in Figure 6 indicate that MUSSEL with fault localization demonstrates a growing performance advantage as code line size increases. Specifically, MUSSEL for vulnerability code generation (*Vul-Code Gen*) consistently completes generation within 1 second for outputs ranging from 1 to 500 lines, whereas the full-code generation time (*Full-Code Gen*) increases from 0.34s to 35.08s. For code segments of tens to hundreds of lines (common in real-world development), this results in a time increase of up to several dozen or even over a hundred times.

b) Effectiveness: We perform performance evaluation experiments on MUSSEL for vulnerability code generation (which requires fault localization) and full-code generation, respectively. The results in Table V indicate that compared to full-code generation, the fault localization-supported strategy yields higher effectiveness. For example, for MUSSEL with ext-5, BLEU improves from 16.12 (in Table V) to 61.74 (as shown in Table IV), and EM improves from 10.19 to 11.64.

c) Generalizability: To demonstrate the generalizability of our approach, we perform additional performance evaluation experiments on SFT, SFT+PPO, and SFT+DPO for full-code generation. The results in Table V indicate that MUSSEL yields higher effectiveness compared to all three methods. For example, BLEU improves from 9.46 to 18.29, and EM improves from 5.56 to 12.26, respectively. To get the results, we perform a difference comparison between the fully generated code and the original code to extract the modified

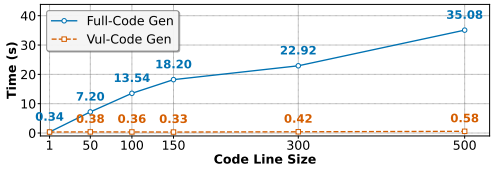

Fig. 6: Generation time comparison across different code lengths.

TABLE V: Performance comparison for full-code generation.

Method	Ext	BLEU	Rouge-L	CodeBLEU	EM
SFT	/	9.46	11.55	12.00	5.56
PPO	/	7.74	8.28	8.96	5.32
DPO	/	12.26	14.58	16.44	6.25
	ext-3	14.72	17.04	26.73	8.56
MUSSEL	ext-5	16.12	18.67	27.49	10.19
(MSR-DPO)	ext-7	17.34	20.03	28.20	11.34
	ext-9	18.29	21.08	28.75	12.26

code patches, which are then compared against the ground-truth patches. For the degrading of BLEU and ROUGE-L scores compared to the fault localization-supported strategy, we think the reason lies in the difficulty for LLMs to accurately localize the target fix in full-code generation, and the models often introduce erroneous modifications at irrelevant locations.

C. Case Studies

Case Study I (Disorganized Patches). The LLM may generate disorganized patches, making code restoration unfeasible. To be more specific, we use the patches produced by our model and a model fine-tuned with SFT for CWE-264 as an example. CWE-264 (*Issues Involving Improper Management of User Permissions*) can lead to security risks like unauthorized disclosure, data tampering, or denial of service. As shown in Figure 7, this example demonstrates a vulnerability in code from the Linux network device driver. The function of this code is to initialize various properties and operations of a bonded device. The operation "`bond_dev->priv_flags &= ~IFF_XMIT_DST_RELEASE;`" modifies the private flags "priv_flags" of the network device. Here, "`&=`" and "`~`"

are bitwise operators used to clear (or reset) specific bits. This handling method could lead to resource leaks or performance bottlenecks, which could be mitigated by adjusting these flags.

```
1       bond_dev->priv_flags |= IFF_BONDING;
2       bond_dev->priv_flags &= ~IFF_XMIT_DST_RELEASE;
3       bond_dev->features |= NETIF_F_VLAN_CHALLENGED;
```

Fig. 7: Vulnerable code snippet of CWE-264.

The modifications provided by SFT resulted in excessively long outputs and the content was disorganized, making code restoration unfeasible, as shown in Figure 8.

```
1       <S2SV_START>|= IFF_BONDING ; bond_dev
        ↪  -><S2SV_MODIFY> priv_flags |=
        ↪  IFF_TX_SKB_SHARING ; bond_dev -> priv_flags
2       <S2SV_MODIFY>&= ~ IFF_XMIT_DST_RELEASE ; (
        ↪  IFF_XMIT_DST_RELEASE | IFF_TX_SKB_SH
```

Figure 8: Patch generated by model with SFT for CWE-264.

To address this issue, MUSSEL employs the "IFF_XMIT_DST_ RELEASE" flag in conjunction with the "IFF_TX_SKB_SHARING" flag, facilitating SKB sharing which enhances performance by reducing the necessity for data duplication. The solution addresses potential resource leaks or performance bottlenecks. Moreover, the sequence lengths produced by MUSSEL are fully consistent with the target. The modified output is presented in Figure 9.

```
1       <S2SV_START>bond_dev -> priv_flags &=
        ↪  ~<S2SV_MODIFY>
2       ( IFF_XMIT_DST_RELEASE | IFF_TX_SKB_SHARING )
3       <S2SV_MODIFY>; bond_dev -> features
```

Figure 9: Patch generated by MUSSEL for CWE-264.

Case Study II (Patches with Incorrect Variable Types). Type mismatches can halt compilation. We contrast patches from our model and a CWE-119 tuned SFT model. CWE-119 (*Buffer overflow vulnerability*) covers buffer overflows leading to crashes or code execution due to poor boundary handling. As shown in Figure 10, "twopass_rc" should be "TWO_PASS". This discrepancy could pose a security risk if the memory layout of "twopass_rc" differs from that of "TWO_PASS". Such a difference may lead to incorrect references to the "stats_in" member, potentially accessing erroneous memory regions, and resulting in data corruption or program crashes (e.g., segmentation faults). Incorrect memory access may also lead to the leakage of sensitive information, particularly when memory regions that should not be accessed are erroneously read.

```
1       static void reset_fpf_position ( struct
        ↪  twopass_rc * p ,  const FIRSTPASS_STATS *
        ↪  position )
2       {
3         p -> stats_in = position;
4       }
```

Figure 10: Vulnerable code snippet of CWE-119.

As shown in Figure 11, after being handled by the SFT, the LLM provided repair suggestions. However, it is evident that, although the suggestions specified the contents to be fixed, the choice of type was still incorrect, specifically "two_pass_rc_t". Additionally, the superfluous pointer "p", when restored to the source code, would lead to a compilation failure.

```
1       <S2SV_START><S2SV_null> static void
        ↪  reset_fpf_position (
2       <S2SV_MODIFY> two_pass_rc_t * p,
3       <S2SV_MODIFY>const FIRSTPASS_STATS * position ) {
```

Figure 11: Patch generated by model w/ SFT for CWE-119.

As shown in Figure 12, our MUSSEL model demonstrated precise repair capabilities, accurately restoring the "TWO_PASS" type without incorporating extraneous pointer content. This indirectly suggests that through further preference learning, the model implicitly acquired some knowledge of code syntax.

```
1       <S2SV_START>static void reset_fpf_position (
2       <S2SV_MODIFY> TWO_PASS
3       <S2SV_MODIFY>* p, const FIRSTPASS_STATS
```

Figure 12: Patch generated by MUSSEL for CWE-119.

D. Ablation Results

a) Impact of different parameter settings on the final results: To investigate the individual contributions of each component in MUSSEL, we conduct a comprehensive ablation study as shown in Table VI. The *FTX* denotes the SFT fusion loss (final term in Equation 4), *ext* represents the non-preferred sample number, *KL-Weight* indicates the weight of the KL divergence term (Equation 5). For instance, removing the KL weight mechanism (w/o KL Weight) leads to a significant decrease across all metrics, with the BLEU score dropping by 3.45 points (61.74→58.29) and EM by 1.82 points (11.64→9.82), demonstrating the importance of dynamically prioritizing different non-preferred samples during training. These results validate our design decisions and highlight the complementary nature of MUSSEL's components in achieving state-of-the-art performance. Additionally, we conducted an *Optimal Hyperparameter Study*, and the results indicate that the optimal values for λ and β are 0.5 and 0.2 (Equation 4).

Figure 13 illustrates the impact of different components on the training loss and reward margin (i.e., the difference between preferred and non-preferred samples rewards). The shaded regions represent unsmoothed value fluctuations, while the solid lines indicate smoothed trends. As shown in the left plot, MUSSEL achieves the fastest loss convergence and the lowest final loss. The right plot demonstrates that MUSSEL, with all components integrated, attains the largest reward margin, indicating more effective training.

b) Analysis of the discrepancy between full-output evaluation and code-only repair evaluation: To examine the potential impact of the output description on model-generated fixed code, we conduct additional calculations for the performance metrics on the fixed code only.

As shown in Table VII, *Full-Eval* denotes the evaluation of the complete model output, while *FixCode-Eval* represents the results after removing the descriptive prefix and

TABLE VI: Ablation study of MUSSEL components on evaluation metrics with ext-5.

Method	BLEU	Rouge-L	CodeBLEU	EM
MUSSEL	61.74	64.44	29.97	11.64
w/o FTX	60.41	63.02	28.63	10.35
w/o ext	59.75	62.91	29.2	11.29
w/o KL-Weight	58.29	62.84	27.36	9.82

TABLE VII: Performance comparison on different output format metrics.

Method	BLEU	Rouge-L	CodeBLEU	EM
Full-Eval	61.74	64.44	29.97	11.64
FixCode-Eval	61.81	64.47	29.97	11.64

vulnerability localization marks preceding the code snippet. The results indicate that BLEU and ROUGE-L scores remain nearly identical after removing the descriptive prefix. This consistency can be attributed to our fine-tuning that enhances the model's instruction-following capability. For example, for the output "CWE-119, The Fixed Code is: <S2SV_START> reset_fpf_postion...," where "CWE-119, The Fixed Code is: <S2SV_START>" serves as the prefix and "reset_fpf_postion..." denotes the fixed code. Across different fixes, the prefixes are largely consistent, with minor variations only in the CWE number identifiers. Moreover, CodeBLEU and EM scores remain unchanged between Full-Eval and FixCode-Eval, as both metrics require reinserting the patch code (after removing the description) into the original code before evaluating the complete code.

VI. Discussion

First, dataset leakage may introduce bias into the evaluation results. To mitigate this, we conduct experiments under two distinct conditions: one allowing potential dataset leakage (RQ1), consistent with settings used in prior work, and the other strictly eliminating any leakage (RQ2). In both scenarios, our method demonstrates consistent improvements over baseline models, validating its robustness.

Second, in real-world scenarios, due to the inherent generation mechanism of LLMs, repeated or highly similar outputs may occur, which potentially reduces the diversity of negative samples and thereby degrades the performance of the intended repair. For example, when using a low temperature and a small top-p value, the model tends to generate nearly identical negative samples, compromising sampling quality. To address this issue, we propose a parameterized sampling strategy that adjusts generation settings based on the number of "ext" samples. Specifically, we select a corresponding set of temperature values within the range of 0.3 to 1.0 for separate sampling, while introducing randomness through varying top-p values. This design ensures both the diversity and effectiveness of negative sample generation. We leave it as future work.

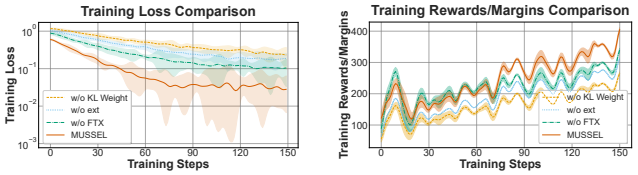

Figure 13: Variation of training loss and reward margins under different conditions.

Third, although our work focuses on vulnerability repair rather than fault localization, the accuracy of upstream localization directly impacts the quality of vulnerability repair. In our experiments, we use precisely localized vulnerabilities obtained from the difference comparison results of GitHub commit histories. However, in practical settings where repair has not yet occurred, localization may rely on LLM-based methods. Fortunately, existing solutions for fault localization [53], [55], [56], [64] can be naturally integrated into the vulnerability repair pipeline.

For accurate fault localization in practice, it requires a strategic combination of modern and conventional approaches to ensure reliable identification of vulnerable code regions. LLM-based localization methods can leverage contextual information from CVE/CWE knowledge bases to guide the identification of potential vulnerability sites, with recent studies demonstrating the effectiveness of incorporating external vulnerability documentation and fine-tuned models for this purpose [5], [23], [53]. Complementarily, conventional program analysis techniques utilizing abstract syntax tree (AST), control-flow graph (CFG), and data-flow graph (DFG) provide robust structural analysis capabilities for vulnerability detection [24], [28], [30]. By integrating these LLM-based and conventional analysis approaches, practitioners can establish a comprehensive localization pipeline that enhances the accuracy of upstream vulnerability identification, thereby improving the overall effectiveness of our proposed repair methodology. This multifaceted localization strategy addresses the practical deployment challenges while maintaining compatibility with existing vulnerability repair frameworks.

VII. Conclusion

In this paper, we propose a framework to address vulnerability repair, called MUSSEL, which is inspired by human learning mechanisms. Through a multi-stage training process involving supervised fine-tuning and self-reward feedback learning, MUSSEL equips LLMs with the foundational knowledge and refinement techniques necessary to generate precise patches efficiently. Our results show MUSSEL outperforms state-of-the-art solutions in One-Shot query with limited GPU memory, effective across diverse CWE types, highlighting major security benefits.

Acknowledgment

We thank all reviewers. This work was partially supported by the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (62121001), by the National Natural Science Foundation of China (62232013), by the Key Research and Development Program of Shaanxi (2025CYYBXM-066), and by the '111 Center' (B16037).

REFERENCES

[1] BAI, Y., JONES, A., AND NDOUSSE, K. Training a helpful and harmless assistant with reinforcement learning from human feedback. *CoRR abs/2204.05862* (2022).

[2] BERGER, A. L., PIETRA, S. D., AND PIETRA, V. J. D. A maximum entropy approach to natural language processing. *Comput. Linguistics 22*, 1 (1996), 39–71.

[3] BHANDARI, G. P., NASEER, A., AND MOONEN, L. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *PROMISE '21: 17th International Conference on Predictive Models and Data Analytics in Software Engineering, Athens Greece, August 19-20, 2021* (2021), ACM, pp. 30–39.

[4] BRIDLE, J. S. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing - Algorithms, Architectures and Applications, Proceedings of the NATO Advanced Research Workshop on Neurocomputing Algorithms, Architectures and Applications, Les Arcs, France, February 27 - March 3, 1989* (1989), vol. 68 of *NATO ASI Series*, Springer, pp. 227–236.

[5] CAO, D., AND JUN, W. Llm-cloudsec: Large language model empowered automatic and deep vulnerability analysis for intelligent clouds. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications Workshops, Vancouver, BC, Canada, May 20, 2024* (2024), IEEE, pp. 1–6.

[6] CHEN, Z., DENG, Y., AND YUAN, H. Self-play fine-tuning converts weak language models to strong language models. *CoRR abs/2401.01335* (2024).

[7] CHEN, Z., KOMMRUSCH, S., AND MONPERRUS, M. Neural transfer learning for repairing security vulnerabilities in C code. *IEEE Trans. Software Eng. 49*, 1 (2023), 147–165.

[8] CHRISTIANO, P. F., LEIKE, J., BROWN, T. B., MARTIC, M., LEGG, S., AND AMODEI, D. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (2017), pp. 4299–4307.

[9] CUI, G., YUAN, L., AND DING, N. Ultrafeedback: Boosting language models with high-quality feedback. *CoRR abs/2310.01377* (2023).

[10] DEVLIN, J., CHANG, M., LEE, K., AND TOUTANOVA, K. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)* (2019), Association for Computational Linguistics, pp. 4171–4186.

[11] DONG, G., YUAN, H., LU, K., AND LI, C. How abilities in large language models are affected by supervised fine-tuning data composition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024* (2024), Association for Computational Linguistics, pp. 177–198.

[12] FAN, J., LI, Y., WANG, S., AND NGUYEN, T. N. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020* (2020), ACM, pp. 508–512.

[13] FAN, L., LIU, Z., WANG, H., BAO, L., XIA, X., AND LI, S. FAIT: fault-aware fine-tuning for better code generation. *CoRR abs/2503.16913* (2025).

[14] FEINBERG, A. Markov decision processes: Discrete stochastic dynamic programming (martin l. puterman). *SIAM Rev. 38*, 4 (1996), 689.

[15] FENG, Z., GUO, D., AND TANG, D. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020* (2020), vol. EMNLP 2020 of *Findings of ACL*, Association for Computational Linguistics, pp. 1536–1547.

[16] FRIED, D., AGHAJANYAN, A., LIN, J., WANG, S., AND WALLACE, E. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, May 1-5, 2023* (2023).

[17] FU, M., TANTITHAMTHAVORN, C., LE, T., NGUYEN, V., AND PHUNG, D. Q. Vulrepair: a t5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022* (2022), ACM, pp. 935–947.

[18] GRUM, M. Learning representations by crystallized back-propagating errors. In *Artificial Intelligence and Soft Computing - 22nd International Conference, ICAISC 2023, Zakopane, Poland, June 18-22, 2023, Proceedings, Part I* (2023), vol. 14125 of *Lecture Notes in Computer Science*, Springer, pp. 78–100.

[19] GUO, D., YANG, D., AND ZHANG, H. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *CoRR abs/2501.12948* (2025).

[20] GUO, D., ZHU, Q., YANG, D., XIE, Z., AND DONG, K. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR abs/2401.14196* (2024).

[21] HU, E. J., SHEN, Y., WALLIS, P., AND ALLEN-ZHU, Z. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022* (2022), OpenReview.net.

[22] IBARZ, B., LEIKE, J., POHLEN, T., IRVING, G., AND LEGG, S. Reward learning from human preferences and demonstrations in atari. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada* (2018), pp. 8022–8034.

[23] KELTEK, M., HU, R., SANI, M. F., AND LI, Z. LSAST: enhancing cybersecurity through llm-supported static application security testing. In *ICT Systems Security and Privacy Protection - 40th IFIP International Conference, SEC 2025, Maribor, Slovenia, May 21-23, 2025, Proceedings, Part I* (2025), vol. 745 of *IFIP Advances in Information and Communication Technology*, Springer, pp. 166–179.

[24] KHARE, A., DUTTA, S., LI, Z., SOLKO-BRESLIN, A., ALUR, R., AND NAIK, M. Understanding the effectiveness of large language models in detecting security vulnerabilities. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025* (2025), IEEE, pp. 103–114.

[25] LEE, W., SHARMA, R., AND AIKEN, A. Training with mixed-precision floating-point assignments. *Trans. Mach. Learn. Res. 2023* (2023).

[26] LEWIS, M., LIU, Y., GOYAL, N., GHAZVININEJAD, M., MOHAMED, A., LEVY, O., STOYANOV, V., AND ZETTLEMOYER, L. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020* (2020), Association for Computational Linguistics, pp. 7871–7880.

[27] LI, X. L., AND LIANG, P. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021* (2021), Association for Computational Linguistics, pp. 4582–4597.

[28] LI, Z., DUTTA, S., AND NAIK, M. IRIS: llm-assisted static analysis for detecting security vulnerabilities. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025* (2025), OpenReview.net.

[29] LIU, J., XIA, C. S., WANG, Y., AND ZHANG, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023* (2023).

[30] LIU, X., WU, J., TAO, Z., MA, Y., WEI, Y., AND CHUA, T. Harnessing large language models for multimodal product bundling. *CoRR abs/2407.11712* (2024).

[31] LV, K., YANG, Y., LIU, T., GUO, Q., AND QIU, X. Full parameter fine-tuning for large language models with limited resources. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024* (2024), Association for Computational Linguistics, pp. 8187–8198.

[32] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings* (2013).

[33] OPENAI. Gpt-4: Generative pre-trained transformer 4. https://openai.com/research/gpt-4, 2023. Accessed: 2024-04-28.

[34] OUYANG, L., WU, J., JIANG, X., ALMEIDA, D., WAINWRIGHT, C. L., MISHKIN, P., AND ZHANG, C. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022* (2022).

[35] PAPINENI, K., ROUKOS, S., WARD, T., AND ZHU, W. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA* (2002), ACL, pp. 311–318.

[36] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., AND ANTIGA, L. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada* (2019), pp. 8024–8035.

[37] PEARCE, H., TAN, B., AHMAD, B., KARRI, R., AND DOLAN-GAVITT, B. Examining zero-shot vulnerability repair with large language models. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023* (2023), IEEE, pp. 2339–2356.

[38] PENG, B., LI, C., HE, P., GALLEY, M., AND GAO, J. Instruction tuning with GPT-4. *CoRR abs/2304.03277* (2023).

[39] PINNAPARAJU, N., ADITHYAN, R., AND PHUNG, D. Stable code technical report. *CoRR abs/2404.01226* (2024).

[40] PRENNER, J. A., BABII, H., AND ROBBES, R. Can openai's codex fix bugs?: An evaluation on quixbugs. In *3rd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2022, Pittsburgh, PA, USA, May 19, 2022* (2022), IEEE, pp. 69–75.

[41] RAFAILOV, R., S. A. M. E. M. Direct preference optimization: Your language model is secretly a reward model. In *Advances in Neural Information Processing Systems* (2024), IEEE, p. 36.

[42] REN, S., GUO, D., LU, S., ZHOU, L., AND LIU, S. Codebleu: a method for automatic evaluation of code synthesis. *CoRR abs/2009.10297* (2020).

[43] ROZIÈRE, B., GEHRING, J., GLOECKLE, F., SOOTLA, S., GAT, I., AND ELLEN, X. Code llama: Open foundation models for code. *CoRR abs/2308.12950* (2023).

[44] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., RADFORD, A., AND KLIMOV, O. Proximal policy optimization algorithms. *CoRR abs/1707.06347* (2017).

[45] SONG, F., YU, B., LI, M., YU, H., AND HUANG, F. Preference ranking optimization for human alignment. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada* (2024), AAAI Press, pp. 18990–18998.

[46] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada* (2014), pp. 3104–3112.

[47] TOUVRON, H., LAVRIL, T., IZACARD, G., MARTINET, X., AND LACHAUX, M. Llama: Open and efficient foundation language models. *CoRR abs/2302.13971* (2023).

[48] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (2017), pp. 5998–6008.

[49] WANG, Y., KORDI, Y., MISHRA, S., LIU, A., AND SMITH, N. A. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023* (2023), Association for Computational Linguistics, pp. 13484–13508.

[50] WANG, Y., LE, H., GOTMARE, A., BUI, N. D. Q., AND LI, J. Codet5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023* (2023), Association for Computational Linguistics, pp. 1069–1088.

[51] WOLF, T., DEBUT, L., SANH, V., CHAUMOND, J., DELANGUE, C., MOI, A., CISTAC, P., RAULT, T., LOUF, R., FUNTOWICZ, M., AND DAVISON, J. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, EMNLP 2020 - Demos, Online, November 16-20, 2020* (2020), Association for Computational Linguistics, pp. 38–45.

[52] YANG, A., LI, A., YANG, B., ZHANG, B., AND HUI, B. Qwen3 technical report. *CoRR abs/2505.09388* (2025).

[53] YANG, A. Z. H., LE GOUES, C., MARTINS, R., AND HELLENDOORN, V. J. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024* (2024), ACM, pp. 17:1–17:12.

[54] YENDURI, G., RAMALINGAM, M., SELVI, G. C., AND SUPRIYA, Y. Gpt (generative pre-trained transformer)— a comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions. *IEEE Access 12* (2023), 54608–54649.

[55] YIN, X., NI, C., AND WANG, S. Multitask-based evaluation of open-source LLM on software vulnerability. *IEEE Trans. Software Eng. 50*, 11 (2024), 3071–3087.

[56] ZHANG, J., WANG, C., LI, A., SUN, W., AND ZHANG, C. An empirical study of automated vulnerability localization with large language models. *CoRR abs/2404.00287* (2024).

[57] ZHANG, K., LI, G., DONG, Y., XU, J., ZHANG, J., SU, J., LIU, Y., AND JIN, Z. Codedpo: Aligning code models with self generated and verified source code. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025* (2025), Association for Computational Linguistics, pp. 15854–15871.

[58] ZHANG, K., LI, G., LI, J., DONG, Y., AND JIN, Z. Focused-dpo: Enhancing code generation through focused preference optimization on error-prone points. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025* (2025), Association for Computational Linguistics, pp. 9578–9591.

[59] ZHANG, Q., FANG, C., YU, B., SUN, W., ZHANG, T., AND CHEN, Z. Pre-trained model-based automated software vulnerability repair: How far are we? *CoRR abs/2308.12533* (2023).

[60] ZHAO, H., HUI, J., HOWLAND, J., AND NGUYEN, N. Codegemma: Open code models based on gemma. *CoRR abs/2406.11409* (2024).

[61] ZHENG, Q., XIA, X., ZOU, X., DONG, Y., WANG, S., XUE, Y., AND ET.AL, L. S. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023* (2023), ACM, pp. 5673–5684.

[62] ZHENG, Y., ZHANG, R., ZHANG, J., YE, Y., LUO, Z., AND MA, Y. Llamafactory: Unified efficient fine-tuning of 100+ language models. *CoRR abs/2403.13372* (2024).

[63] ZHOU, X., KIM, K., XU, B., HAN, D., AND LO, D. Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024* (2024), ACM, pp. 88:1–88:13.

[64] ZHOU, X., TRAN, D., LE-CONG, T., ZHANG, T., IRSAN, I. C., SUMARLIN, J., LE, B., AND LO, D. Comparison of static application security testing tools and large language models for repo-level vulnerability detection. *CoRR abs/2407.16235* (2024).