Leveraging Mixture-of-Experts Framework for Smart Contract Vulnerability Repair with Large Language Model

Hang Yuan[†‡], Xizhi Hou[†‡], Lei Yu[†‡], Li Yang[*†‡], Jiayue Tang [†‡],
Jiadong Xu[†‡], Yifei Liu[†‡], Fengjun Zhang[†], Chun Zuo[§]
[†]Laboratory of Precise Computing, Institute of Software, Chinese Academy of Sciences, China
[‡]University of Chinese Academy of Sciences, China
[§]Sinosoft Co., Ltd., China
{yuanhang2023, yulei2022, yangli2017, fengjun}@iscas.ac.cn,
{houxizhi24, tangjiayue23, xujiadong24, liuyifei241}@mails.ucas.ac.cn, zuochun@sinosoft.com.cn

***Abstract*—Smart contracts are a core component of blockchain ecosystems, but their transparency and immutability make them vulnerable to attacks, leading to significant financial losses. Thus, repairing vulnerabilities in smart contracts is crucial for establishing a trustworthy blockchain environment. Existing smart contract vulnerability repair methods suffer from a critical "one-for-all" design limitation, where a single model is tasked with fixing diverse vulnerability types, leading to suboptimal performance due to insufficient specialization. To address this, we propose MoEFix, a novel framework leveraging a Mixture-of-Experts (MoE) architecture tailored for smart contract characteristics. MoEFix partitions vulnerabilities into subspaces, trains specialized experts for each type (e.g., reentrancy, integer overflow), and employs a vulnerability-aware router to dynamically allocate repairs. We further redesign the repair workflow to align with large language models, enabling end-to-end secure contract generation instead of partial patches, and to achieve this, we curated a dataset of 1,391 contracts covering five critical vulnerability types.**

To validate our approach, we extend the benchmark PVD test suite. Experiments demonstrate that MoEFix outperforms state-of-the-art methods by 21.64% in overall accuracy, achieving improvements of 26.19% (reentrancy) and 23.08% (delegatecall) for specific vulnerabilities.

***Index Terms*—Smart Contract, Large Language Models, Mixture of Experts**

I. Introduction

Blockchain technology has been rapidly adopted across various domains due to its decentralized architecture [1]. This innovative technology enables the creation of secure, distributed digital ledgers for recording transactions [2]. Smart contracts are programs that operate on a blockchain, and their primary purpose is to automatically fulfill obligations between untrusting parties. Inheriting advantages such as the transparency of blockchain, smart contracts receive widespread attention in various fields, including digital assets and supply chains [2]. In this ecosystem, smart contracts function as self-executing programs on the blockchain, enabling the automated management of digital assets like cryptocurrencies. These contracts activate when specific conditions are met, and once deployed, they become a permanent component of the blockchain [3]. However, the immutability and inherent complexity of smart contracts pose significant security challenges [3]. For instance, the notorious DAO hack [4], [5] serves as a cautionary example, which resulted in the illegal transfer of $60 millionworth of Ethereum [6], demonstrating the potential severity of such vulnerabilities. Therefore, ensuring the security of smart contracts is crucial for a trustworthy blockchain ecosystem.

Vulnerability detection and repair are indispensable stages in the smart contract development lifecycle. In recent years, a plethora of research has emerged focusing on the detection of various vulnerabilities [7], with significant studies also addressing vulnerability repair [8]–[14]. SCRepair [10] stands as the first search-based Automatic Program Repair (APR) tool for smart contract source code. However, the search-based method employed by SCRepair has limited effectiveness when dealing with vulnerabilities that require complex code changes, and it cannot achieve full automation, requiring users to provide test cases [13]. Unlike SCRepair's repair approach, sGuard [12] adopts a source-code-based, semantic-driven method to ensure that contract vulnerabilities are correctly addressed. Nevertheless, sGuard's detection method, which is based on symbolic execution, is inaccurate and time-consuming. Consequently, the correctness of the repair is severely impacted, and patches inserted by sGuard may introduce new vulnerabilities and affect the original business logic. Therefore, Gao et al. [13] propose sGuard+, a machine-learning-based smart contract vulnerability remediation method. It rectifies the weak patches that fail to fully defend against vulnerabilities and enhances repair correctness. Subsequently, Guo et al. [11] propose RLRep, a reinforcement learning-based method for smart contract vulnerability remediation, which adopts an agent to provide repair action suggestions without any supervision.

However, these methods aim to repair multiple types of vulnerabilities using a single model. These techniques train a single model to handle all vulnerability types. Since each vulnerability possesses its unique characteristics and impact,

*Corresponding author

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00140

such approaches often fail to adequately capture the distinct patterns and features of different vulnerability types. A recent study [15] reveals a significant challenge that prevents deep learning models from being adopted in industry settings: the **one-for-all design limitation**. Yang et al. [16] suggests this limitation may prevent existing deep learning models from effectively handling the differences between various types of vulnerabilities. This represents a constraint on the model's overall capability, rather than being task-specific, which reveals an inherent limitation in the model's generalization ability. To further demonstrate this design limitation in the domain of smart contract vulnerability repair, we conducted an empirical study inspired by [16]. Our findings show that state-of-the-art methods exhibit performance discrepancies across different vulnerability types and also show a performance gap when compared to expert models specifically trained for each vulnerability type.

To address this issue, we propose utilizing a Mixture-of-Experts (MoE) framework. The MoE framework [17], [18] represents a promising solution. This framework uses multiple specialized models (i.e., experts) to handle different parts of the input. A router directs each input to the most suitable expert, unlike a single model that processes all inputs. Building upon this, we design MoEFix, which comprises four steps: Splitting Input Space, Expert Training, Router Training, and Combiner. First, the input space is partitioned based on vulnerability categories, thereby defining distinct vulnerability repair experts. Next, these experts are individually trained according to their respective subspaces, forming specialized vulnerability repair experts. To train a routing mechanism capable of capturing vulnerability characteristics, other parameters are frozen, and only the router is updated. Finally, we combine these components to form the Mixture-of-Experts network.

In recent years, Large Language Models have developed rapidly. Owing to the effectiveness of transformers in scaling efficiently to large corpora, LLMs now excel in tasks such as question-answering, text summarization, and code generation [19]. This offers a new solution for smart contract vulnerability repair. However, recent research [20] has shown that conventional repair workflows are not well-suited for LLMs. In the process of smart contract vulnerability repair, existing workflows typically include three steps: vulnerability localization, patch generation, and patch validation. In the vulnerability localization phase, existing methods commonly use vulnerability detection tools [11] or static statistics [14] to provide localization information, or they directly use perfect vulnerability localization results as evaluation criteria. Nevertheless, in the current workflow, simply substituting LLMs into the patch generation step underestimates their pre-trained knowledge, as these models are capable of independently locating and repairing defective contracts. Furthermore, incorrect vulnerability detection results can negatively impact a LLMs' repair capabilities. To address the challenge and better align with the holistic capabilities of LLMs, we design an end-to-end workflow, utilizing the LLMs' inherent knowledge and capabilities to pinpoint vulnerabilities and directly perform the repair.

Recent studies [11], [13] highlight the scarcity of high-quality datasets tailored for smart contract repair, particularly those compatible with LLM training requirements. Existing datasets often suffer from limited scale (e.g., RLRep contains only 498 contracts in RE, IO and TX vulnerabilities) or lack diversity in vulnerability types (e.g., missing delegatecall and timestamp vulnerabilities). Furthermore, shifting LLMs' output from fixed patchs to the entire refined program can better align the inference objective with the training, thus significantly enhancing repair performance [20]. Therefore, it is necessary to build a smart contract dataset that contains various vulnerabilities. Referencing Guo et al.'s [11] strategy , we detect the collected smart contracts using existing state-of-the-art detection tools, retaining only those with vulnerabilities. Subsequently, by leveraging existing LLMs, thought processes and explanations can be generated to inform vulnerability repair. Finally, human experts manually repair the smart contracts based on the output of these LLMs. Ultimately, the resulting dataset consists of 1391 vulnerable contracts across five vulnerability categories. This includes 261 reentrancy vulnerability contracts, 481 integer overflow/underflow vulnerability contracts, 260 delegatecall vulnerability contracts, 187 timestamp vulnerability contracts, and 202 tx.origin vulnerability contracts.

To validate the effectiveness of our method, we expand the test set PVD [13] and evaluate MoEFix's performance across five different vulnerability categories. The results indicate that our method outperforms state-of-the-art methods on all vulnerabilities. Specifically, for reentrancy, delegatecall, timestamp, integer overflow/underflow, and tx.origin vulnerabilities, MoEFix surpasses current SOTA methods by 26.19%, 23.08%, 15.79%, 16.92%, and 14.76%, respectively. Furthermore, in terms of overall performance, MoEFix achieves a 21.64% higher accuracy rate than SOTA methods.

Our contributions are as follows:

- We propose the first MoE framework for smart contract vulnerability repair, addressing the "one-for-all" limitation by training specialized experts for distinct vulnerability subspaces (e.g., reentrancy, delegatecall). The router dynamically selects experts based on code features (e.g., external calls, arithmetic operations), ensuring targeted and efficient repairs.
- We redefine the repair pipeline to fully leverage LLMs' holistic understanding. Unlike existing patch-based approaches, MoEFix generates complete secure contracts by integrating vulnerability location and repair, mitigating error propagation from standalone location tools.
- We construct a dataset of 1,391 vulnerable contracts with five critical vulnerability types, annotated via a hybrid process combining static analysis, LLM-generated explanations, and expert validation, which aligns with the immutability of smart contracts and the specific codes of blockchains.

All the experimental data and source code is online available at https://zenodo.org/records/15544993

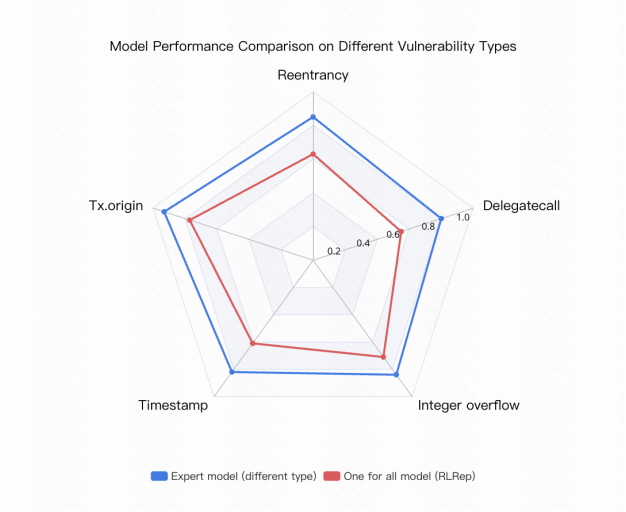

Fig. 1. Motivation. The performance of one-for-all Model (RLRep) compared with expert model trained on each specific Smart Contract vulnerability.

II. Background and Motivation

A. Typical Vulnerabilities of Smart Contract

In this paper, we focus on five key vulnerability types: Reentrancy, Timestamp Dependency, Integer Overflow/Underflow, Delegatecall and Tx.origin. Research [21] indicates that approximately 70% of financial losses in Ethereum smart contract attacks can be attributed to these vulnerabilities. Furthermore, existing studies demonstrate that these vulnerabilities occur more frequently in Ethereum smart contracts compared to other types of contracts [21]–[23]. Consequently, these vulnerabilities have significant negative impacts on both the security and functionality of the contracts, potentially leading to substantial economic losses. Despite their well-known status, these vulnerabilities are still frequently overlooked or misunderstood due to the inherent complexity and immutability of smart contracts.

B. Mixture of Experts

Mixture of Experts (MoE) [17] is a hybrid model consisting of multiple sub-models, known as experts, which are integrated together. The key concept of MoE is the use of a router to determine the token set that each expert handles, thereby reducing interference between different types of samples [24]. Fundamentally, the MoE represents a combinatorial approach that leverages individual learning techniques as experts within distinct subspaces of the input space [25]. The core idea behind MoE is to select suitable experts for a particular input using a router [16]. Typically, MoE contains the following three components: **Experts, Router and Combiner**. The **Experts** are individual models or neural networks trained specifically to address different parts of the input space or distinct tasks. These experts can take various forms, such as linear models, decision trees, or deep neural networks. The **Router** functions as a gating mechanism that determines which expert or combination of experts should process a given input. It is typically implemented as a neural network. The router selects the most relevant expert subset for each input, rather than invoking all experts for inference. The **Combiner** receives the outputs from the experts, weights them according to the probabilities or weights assigned by the router, and generates the final output.

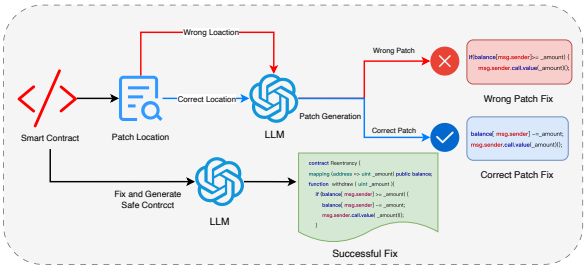

Fig. 2. Motivation. An example of two different workflow for Smart Contract Repair.

C. Motivation for Smart Contract Vulnerability Fix

*1) **Smart Contract Vulnerability Fix Workflow**:* The automated program repair process typically involves providing a faulty program and an artifact, whereupon automated program repair (APR) methods generate a fixed program that satisfies correctness criteria (e.g., passes all tests) [20]. Existing smart contract vulnerability repair methods [10]–[12], [14] adopt the traditional automated program repair workflow, which generally includes three steps: vulnerability localization, patch generation, and patch verification. Among these, the main research focus lies in the patch generation phase. In the smart contract vulnerability localization phase, existing methods commonly employ vulnerability detection tools [11] or static statistical analysis [10], [12] to provide localization information, or directly use perfect vulnerability localization results as an evaluation condition. However, these traditional vulnerability detection tools and static statistical methods exhibit issues with accuracy and efficiency, and directly adopting perfect localization results similarly does not align with real-world scenarios.

In recent years, interest in using large language models, such as GPT-4 [26], for Automated Program Repair (APR) [27]–[30], increasingly grows. Researchers typically follow the current workflow, positioning LLMs as novel patch generators to supersede previous models. However, within the current workflow, simply treating LLMs as a substitute for the patch generation step represents an under-utilization of their pre-trained knowledge, as these models possess the capability to independently locate and fix contracts [20]. As illustrated in Fig. 2, because vulnerability localization tools may not perfectly detect the precise location of vulnerabilities, the provided information can negatively impact LLMs, leading them to waste time on incorrect patches or even return erroneous results. To address these limitations and align with the holistic

capabilities of LLMs, we design an end-to-end workflow. This approach leverages the LLM's inherent knowledge and capabilities for vulnerability localization, directly performs vulnerability repair.

*2) **Vulnerability Fix with Mixture of Experts**:* Research [15] indicates the limitations inherent in the prevalent "one for all" design of existing deep learning-based vulnerability detection techniques. While this approach simplifies development and deployment processes, it concurrently fails to adequately accommodate the diverse demands of the real world [16]. A similar issue exists within the technical domain of vulnerability repair [11], [12], [14]. In practice, large organizations often encounter various types of vulnerabilities, each with unique characteristics and implications [31]. A one-for-all model may fail to adequately address the nuances of different vulnerability types and lack the capability of detecting and fixing certain types of vulnerabilities. These techniques operate by training a single model to address all vulnerability types. Consequently, they often fail to capture the unique patterns and characteristics inherent in different categories of vulnerabilities. This not only poses challenge for vulnerabilities that are underrepresented in the training dataset, but also leads to diminished performance and an increased risk of security breaches.

For instance, Fig.1 illustrates the performance of RLRep, a state-of-the-art "one for all" approach. This method employs reinforcement learning and trains a single model to repair various types of vulnerabilities. While RLRep leverages reinforcement learning to provide the model with some repair capabilities, the underlying logic, code patterns, and repair strategies for different types of smart contract vulnerabilities—such as Reentrancy and Integer Overflow—are fundamentally distinct. For instance, repairing a Reentrancy vulnerability typically requires implementing the Checks-Effects-Interactions pattern or using a mutex lock. In contrast, fixing an Integer Overflow demands the use of a safe math library. The experiments show that the RLRep not only exhibits performance variations across different vulnerability types but also demonstrates a performance gap compared to expert models trained specifically for each vulnerability type. This suggests that a single-architecture model, like RLRep, which attempts to fix all types of vulnerabilities, struggles to simultaneously learn these vastly different repair paradigms.

To address the limitations of a single smart contract vulnerability repair framework, we'll design a solution using a Mixture of Experts (MoE) framework. The MoE framework enables specialization, with each expert focusing on a particular vulnerability type. Since vulnerability types naturally fall into distinct categories, this specialization enhances the accuracy and efficiency of the fix, making the solution more practical for real-world use.

III. Approach

In this section, the design and implementation of MoEFix are presented, which primarily comprises four key steps: Splitting Input Space, Expert Training, Router Training, and Combiner. The overall architecture of our approach for smart contract vulnerability fix is illustrated in Fig. 4.

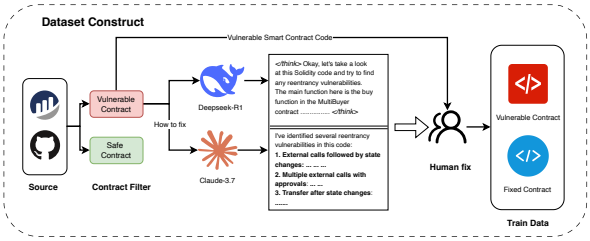

Fig. 3. The process of end-to-end workflow dataset construction.

A. Data construction

To train our experts and MOE model, we have curated a large-scale dataset of smart contract vulnerability repair. As shown in Fig. 3, we first collect smart contract source code data from Etherscan and GitHub which provide us with a broad range of smart contract sources. To ensure uniqueness, the Jaccard Index [32], a computationally efficient token-based similarity algorithm, was employed. To collect vulnerable smart contracts, we follow the methodology by [11]. Specifically, we leverage several vulnerability detection tools [33]–[36] to collect smart contracts exhibiting diverse vulnerabilities. The most suitable detection tool is meticulously selected for each vulnerability type, considering the unique characteristics of each tool. Furthermore, to account for the inherent accuracy limitations of these tools, we also employ state-of-the-art large language model-based detection methods for verification, such as GPTScan [37] and GPTLen [38], to filter out contracts flagged as false positives.

Manually repairing these vulnerabilities is a time-consuming endeavor that demands significant expertise from repair personnel. Therefore, in constructing our dataset of vulnerability fixes, we leverage two currently high-performing large language models: DeepSeek-R1 [39] and Claude-3.7-Sonnet [40]. To ensure the accuracy and relevance of the large language model outputs, we design specialized prompts for each vulnerability type. These prompts include detailed descriptions of the characteristics of each vulnerability and guide the models to provide relevant repairs for the corresponding smart contract flaws. We feed the collected vulnerable smart contracts as input to the large language models, prompting them to provide repair suggestions. Specifically, we extracted the chain-of-thought reasoning from DeepSeek-R1 and the repair suggestions generated by Claude-3.7-Sonnet as references for manual repair. Two researchers independently review the model's reasoning and suggestions for each smart contract vulnerability before manually applying the fixes. Subsequently, the two researchers discuss and integrate their independently generated fixes. In cases of disagreement, a third researcher is consulted to facilitate discussion and arrive at the final repaired code. A total of twelve researchers, each possessing over two

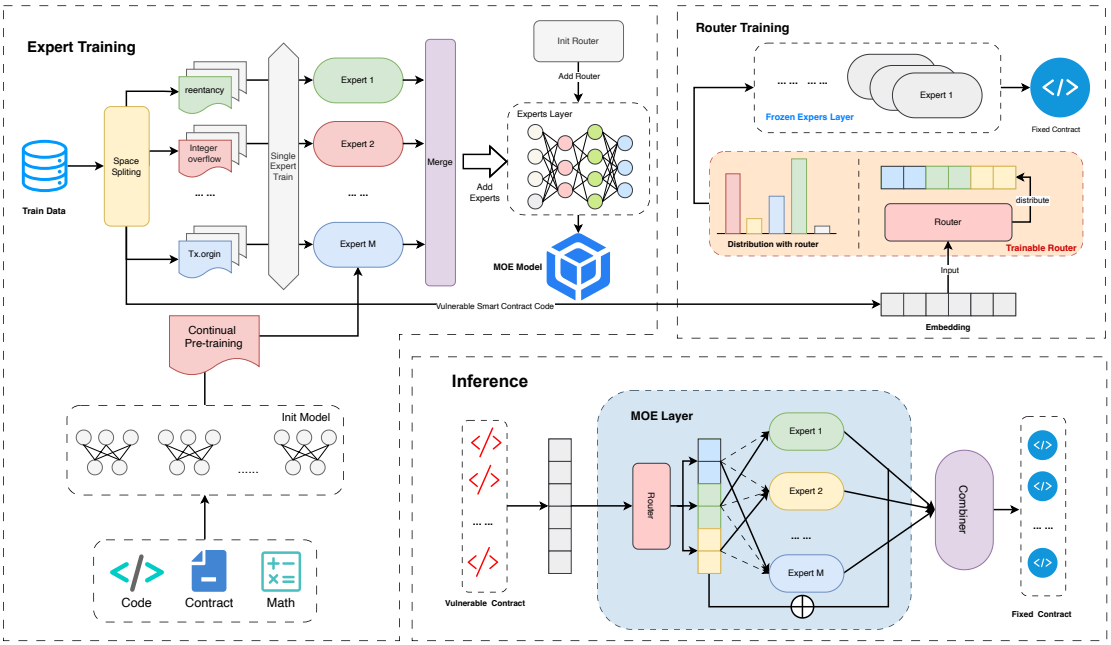

Fig. 4. The Overview of the MoEFix.

years of experience in smart contract research, participate in this repair process.

Ultimately, we obtain a dataset comprising 1391 vulnerable contracts across five vulnerability categories. This dataset includes 261 contracts with reentrancy vulnerabilities, 481 contracts with integer overflow/underflow vulnerabilities, 260 contracts with delegatecall vulnerabilities, 187 contracts with timestamp vulnerabilities, and 202 contracts with tx.origin vulnerabilities.

B. Model Selection

We employ LLaMA-3.2-3B [41] as the base model for Continual Pre-training and Expert Training, based on the following considerations: 1) Its open-source nature ensures transparency and accessibility; 2) The smaller parameter scale (3B) significantly reduces training and inference costs, making it particularly suitable for validating the effectiveness of MOE model; 3) It shows strong fine-tuning potential for better task adaptation; 4) Previous research [42]–[44] has validated its reliable performance across various scenarios.

C. Splitting Input Space

To address the limitations of smart contract vulnerability repair posed by a singular framework, we aim to design a solution using the Mixture of Experts (MoE) framework. First, to effectively leverage the MoE framework, it is crucial to split the input space into sub-spaces, allowing each expert to handle a specific sub-space [16]. Subsequent training enables the experts to specialize, forming distinct experts that correspond to the partitioned sub-spaces. During training, each expert undergoes differentiated training, focusing on and mastering the characteristics of smart contract vulnerabilities associated with its assigned sub-space, thereby enhancing its vulnerability repair capabilities within that domain. Extensive research indicates that many existing insecure smart contracts still contain numerous common patterns. Several studies and technical documents categorize different types of vulnerabilities based on these patterns and propose relevant repair recommendations [45], [46]. In this paper, we focus on the following five vulnerabilities: Reentrancy, Timestamp Dependency, Integer Overflow/Underflow, Delegatecall, and Tx.origin. We use the types of smart contract vulnerabilities as the basis for sub-space partitioning.

D. Experts Training

To specialize our model for smart contract tasks, we first implement continual pre-training on the expert model. This stage enables the LLM to develop a deep understanding of specific contract patterns and concepts. The weights obtained from this phase serve as the foundation for subsequent MOE expert training, facilitating better training and differentiation of the experts. We based our Continual Pre-training dataset on research from [47], which underwent extensive filtering and quality checks. Furthermore, to maintain the model's versatility while preventing knowledge degradation, we add diverse content such as mathematics, general contracts, and programming to the training data. This balanced approach ensures that the model retains its broad knowledge base and simultaneously develops expertise in smart contract analysis, thereby enhancing generalization and robust performance

across different contract scenarios. Based on the splitting of the input space, we categorize experts into distinct groups. For each type of vulnerability, a specific expert model is trained, tailored to address that particular class of flaw. The reentrancy expert specializes in analyzing features such as the order of state updates and the placement of external calls within the contract. Integer Overflow expert focuses on inspecting the safety of numerical operations, including mathematical calculations and type conversions. Timestamp Dependency expert primarily examines how block timestamps are utilized and their potential impact on the contract's logic. Delegatecall expert verifies the access control and context inheritance associated with the use of delegatecall.

First, based on the subspace division, for each vulnerability type, we have a sub-dataset $D_t = (x^{(i)}, y^{(i)})_{i=1}^{|D_t|}$, where t represents the vulnerability type, $x^{(i)}$ is the i-th input code sample containing the specific type of vulnerability, and $y^{(i)}$ is its corresponding i-th repaired target code sample. For a given vulnerability type, our objective is to train a specialized expert model that repairs the specific type of vulnerability present in the input $x^{(i)}$ and outputs the repaired code $y^{(i)}$.

For a single sample $(x^{(i)}, y^{(i)})_i$, in the dataset, the loss $L_t^{(i)}(\theta_t)$ can be defined as:

$$L_t^{(i)}(\theta_t) = -\sum_{k=1}^{M_i} \log P(t_k^{(i)} | t_1^{(i)}, \ldots, t_{k-1}^{(i)}, x^{(i)}; \theta_t) \quad (1)$$

where M_i is the length of the code sequence, and θ_t represents the parameters of the expert specialized in repairing this type of vulnerability. The total loss $L_t(\theta_t)$ for the expert model on the subspace is then:

$$L_t(\theta_t) = \frac{1}{|D_t|} \sum_{i=1}^{|D_t|} L_t^{(i)}(\theta_t) \quad (2)$$

Following separate training on the distinct subspaces, we obtain different experts $E_1, E_2, \ldots, E_N$, each focused on repairing specific vulnerabilities. We provide a strong initialization for each expert through continual pre-training and subsequently fine-tune the individual experts on their designated subspaces. This process enables the experts to specialize and form a diverse network dedicated to particular vulnerabilities. By combining these experts, we ultimately achieve an expert network for smart contract vulnerability repair.

E. Router Training

A specific type of vulnerability relates to certain distinctive markers and contexts; however, traditional "one-for-all" models exhibit an imbalanced processing of crucial information during their operation [16].

This is a multi-vulnerability classification neural network for input examples. It takes an instance as input and outputs a probability distribution over all experts. This router analyzes key features within the code, such as external calls, mathematical operations, and timestamp usage, to determine which experts should be activated. This approach effectively handles the pertinent vulnerability features, thereby facilitating a more accurate matching of inputs and preparing them for subsequent expert processing. Regarding the routing selection, we adopt

$$G(x,k)_i = \frac{exp(f(x,k)_i)}{\sum_j^E exp(f(\mathbf{x},\mathbf{k})_j)}, \quad i = 1, \ldots, E \quad (3)$$

$$f(x,k)_i = (x \cdot W_v)_i \cdot I(i \in \text{top } k \text{ elements of } E) \quad (4)$$

where W_v is a trainable parameter matrix used to evaluate the alignment between code features and the expertise of each specialist. and k is the selected experts and $f(x,k)_i$ represents the routing output for each expert. I represents an indicator function, where its value is 1 when the condition is true, and 0 otherwise. The router's output consists of corresponding output weights for each expert. Through this routing mechanism, only the top-k experts are activated during each inference. Following normalization, a sparse weighted combination of these k experts constitutes the final output.

The objective is to train a router capable of discerning the distinctive features of various vulnerabilities, thereby enabling the selection of appropriate experts during the routing process. During the router training phase, we elect to train a designated set of parameters while freezing the remaining ones. In the Expert Training phase, we have already trained the experts, resulting in specialized entities focused on the remediation of specific vulnerabilities. To prevent any adverse effects on the experts during router training, we consequently choose to freeze their parameters and train only the router.

Therefore, we define the model's loss function as follows:

$$L_{\text{moe}} = -\mathbb{E}(x,y) \sim \mathcal{D}[\log P(y|x, \theta_{\text{router}})] + \alpha \cdot L_{\text{balance}} \quad (5)$$

Here, the parameter θ_{router} encompasses the MOE layer parameters of the model, and all other parameters remain frozen. $L_{balance}$ is an auxiliary loss used to balance the expert load, which is scaled by the balancing coefficient α. Through the above process, we train the router while preserving the efficacy of the Expert Training. We achieve router training while maintaining the effectiveness of the Expert Training phase. The result is a routing allocation mechanism tailored to different vulnerability types, which, when combined with the specialized experts, ultimately constitutes the MOE layer.

F. Combiner

We integrate the experts with the router's output through a weighted summation mechanism, which ultimately serves as the output of the MOE layer. Essentially, the combiner is an instance-level network that uses a weighted aggregation mechanism.

The output of the MoE layers can be formulated as

$$\text{MoE}(x) = \sum_{i=1}^{E} G(x)_i E_i(x), \quad i = 1, \ldots, E \quad (6)$$

Instead of using all experts with different weights, using only the top$k$ experts reduces the inference cost and reduces the

impact caused by irrelevant experts. To realize sparse selection for experts, we only compute on the selected experts based on the output of routing. Specifically we select the first k experts, and the selected experts will participate in the computation with different weights, while the experts not selected for are not involved in the computation. And the associated weights are the result of a normalization process. We opt to maintain consistency with existing practices regarding the selection of k [48], setting k=2. Subsequent experiments in Section V-B further demonstrate that our model achieves a balance between performance and inference efficiency under this parameter configuration.

IV. EVALUATION DESIGN

We evaluate MoEFix to answer the following research questions.

RQ1: How does MoEFix perform compared to baselines in Smart Contract Vulnerability Repair?

RQ2: How do the total experts and activations affect the effectiveness of MoEFix?

RQ3: How effective is the MoE framework and Continual Pre-Training within MoEFix?

RQ4: How efficient is the MoEFix regarding its computational costs and time?

RQ5: How does the accuracy of router selection affect MoEFix?

A. Data Preparation

MoE Model Train: For Continual Pre-training, the dataset we employ is derived from the research findings of Storhaug et al [47]. Inspired by the work of [49], we further expand the dataset with an additional 50,000 instances from various domains, such as general code and mathematics. For Experts and Routers, since there is a lack of public datasets for smart contract repair, we build a dataset of vulnerable smart contracts with five types of vulnerabilities. The final dataset comprises 1,391 vulnerable smart contracts, categorized into five types of intelligent contract vulnerabilities.

Evaluation: Regarding the evaluation test set selection, we opted for the publicly available dataset (PVD) proposed by Gao et al. [13]. This dataset comprises 234 unique contract source code files, which are labeled with 247 vulnerable functions. However, it only includes 31 reentrancy vulnerabilities and 18 tx.origin vulnerabilities, and does not encompass delegatecall and timestamp vulnerabilities. To fully evaluate our method, following the test set partitioning approach of [11], we expanded the PVD to adequately cover the scope of our trained model while also achieving a more balanced distribution across different vulnerability types. In total, the benchmark dataset comprises 365 vulnerable smart contracts.

B. Baseline

To provide the comprehensive evaluation of our method, we selected a range of current baselines encompassing different types of approaches, including reinforcement learning-based, machine learning-based, statistical model-based, and static analysis methods. RLRep [11] is a reinforcement learning-based method. Nguyen et al. [12] propose sGuard, which performs static analysis on symbolic execution traces to detect vulnerabilities and subsequently applies verified fixes. Building upon this work, Gao et al. [13] introduce sGuard+, a machine learning-based smart contract vulnerability repair method designed to enhance the sGuard. SmartFix [14] leverages a statistical model derived from validation feedback to accelerate the generation and verification of repairs. For LLM-based techniques, we selected D4C [20], a program repair technique based on large language models which enables the LLM to repair the entire program without first identifying the faulty statements. We also used Claude-3.7-Sonnet and LLaMA3.2-3B as baselines. For ensemble methodology, we chose P-EPR [50], a repair tool integration method based on repair patterns and preferences.

For a more equitable comparison between our method and the baselines, we augment the action space of the RLRep method by incorporating actions related to delegatecall and timestamp fixing. Furthermore, we train this enhanced RLRep method using a newly curated dataset. Similarly, we expand the dataset used for sGuard+. We also train and evaluate sGuard+ following its three-step paradigm of vulnerability detection, localization, and repair.

C. Evaluating Correctness

Following the approach in [11], [12], [14], we manually verify the generated vulnerability repair suggestions. Assuring patch correctness remains an open problem in automatic program repair techniques [51]. To verify the correctness of contract repair operations, two volunteer students with two years of Solidity development experience are invited to conduct an evaluation. Based on the SWC Registry and contract repair suggestions from prior work, they determine whether the repaired contracts are fixed according to the suggestions while ensuring that the semantics of the original contracts are not altered. If the two students have differing opinions, they jointly discuss and decide on the final result. For the issue of lacking test cases, we refer to the method in [11] and utilize its defined paradigm for testing. Notably, in contrast to Guo et al.'s [11] approach of setting beamsize=5 and considering a vulnerability correctly fixed if at least one suggestion is correct, we adopt a more rigorous majority voting method for evaluation. That is, for five repair suggestions for a single vulnerability, the vulnerability is considered successfully repaired only if a majority of the suggestions are deemed correct.

D. Implementation Details

We perform Continual Pre-training using LlamaFactory [52], and DeepSpeed [53] with bf16 enabled. The loss is calculated using cross-entropy, and parameters are optimized using AdamW [54] with β=$(0.9, 0.99)$ and ϵ=1e-8. During Continual Pre-training, we set the batch size to 64 per device, gradient accumulation steps to 16, epochs to 2, learning rate to 1e-5 with cosine decay, warmup steps to 0, cutoff length to 2048, and save steps to 500. For the training of different

experts, we fine-tune them on specific subspaces. We set the batch size to 2 per device, gradient accumulation steps to 8, epochs to 3, learning rate to 1e-5 with cosine decay, warmup steps to 0, cutoff length to 2048, and save steps to 200. The training of the router is essentially the training of the entire MoE model, except that we freeze some parameters. For evaluating our method, if not specified, the hyperparameters for the number of experts and the number of activations were set to: total experts to 10 and activated experts to 2. All models were trained on a server equipped with 2 NVIDIA H800 GPUs, each with 80GB memory.

V. EVALUATION RESULT

A. RQ1: Performance of MoEFix

In this RQ, we aim to investigate the effectiveness of MoEFix compared to baseline methods in smart contract vulnerability repair. Table I presents the performance comparison between our approach and studied baselines.

The experimental results reveal that MoEFix consistently outperforms the baselines across all vulnerability types. Specifically, compared to the best baseline, our method achieves a 26.19% higher repair accuracy for Reentrancy vulnerabilities (improving from 63.09% to 89.28%). For Delegatecall and Timestamp vulnerabilities, MoEFix demonstrates accuracy increases of 23.08% and 15.79%, respectively. Furthermore, our approach shows a 16.92% improvement in accuracy for Integer overflow vulnerabilities when compared to sGuard+. Notably, for Tx.origin vulnerabilities, MoEFix achieves an accuracy of 91.80%, significantly surpassing other baselines. In addition, the experimental results show that MoEFix outperforms the ensemble method P-EPR. Specifically, in the repair of Delegatecall and Reentrancy vulnerabilities, MoEFix's performance is higher than P-EPR by 48.07% and 53.57%, respectively. MoEFix can achieve a deeper understanding of contract context and the root causes of vulnerabilities through learning. Consequently, compared with fixed repair patterns, it is able to generate more targeted repair solutions. Compared to the LLM-based program repair method D4C and the currently popular large language models, MoEFix also demonstrates better performance, surpassing them in overall performance by 31.50%, 32.82% and 45.47%, respectively.

Furthermore, the one-for-all model often fails to adequately address the unique characteristics and impacts of different vulnerabilities. This limitation is specifically evident in the notable disparities observed in vulnerability repair correctness rates. Even with modifications to RLRep — such as incorporating specific handling mechanisms for relevant vulnerabilities — its repair capabilities for Delegatecall vulnerabilities remain insufficient compared to its performance in fixing other types of vulnerabilities. MoEFix trained with the MoE architecture not only achieve improved performance in vulnerability repair but also enable each expert to specialize in the unique characteristics of different vulnerabilities.

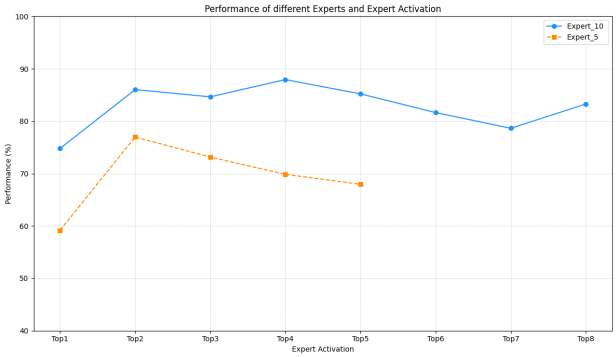

Fig. 5. Performance of different Experts and Expert Activation. Expert-10(5) means the total number of experts is 10(5). The x-axis represents the number of experts activated during each inference of the model. The y-axis indicates the performance of different variant models.

Answer to RQ1: MoEFix consistently outperforms all baselines in terms of vulnerability repair performance and achieves a vulnerability overall repair accuracy of 86.02%, which surpasses current state-of-the-art methods in overall repair performance.

B. RQ2: Impact of the Number of Selected Experts

To systematically evaluate the impact of the number of experts on MoEFix performance, we conducted experimental analysis exploring various combinations of total experts and activation patterns. Combination experiments are conducted with varying activation quantities, considering two main configurations: a total of 10 experts and a total of 5 experts. Specifically, the setting with a total of 10 experts implies that the MoE model incorporates two experts for each type of vulnerability repair. As illustrated in the Fig. 5, the results demonstrate the performance variations in vulnerability repair tasks under different expert-activation configurations.

When the total number of experts remains constant, increasing the number of activated experts does not lead to a significant performance improvement. Specifically, in the scenario with a total of 10 experts, performance is lowest (74.79%) when only one expert is activated. As the number of experts increases, performance reaches its optimum (87.94%) when four experts are chosen. Subsequently, further increases in the number of experts do not yield a significant performance gain; in fact, performance may even decline slightly. Similarly, when the total number of experts is 5, performance is lowest when only one expert is activated. The model achieves its optimal performance when two experts are activated. We analyze that when the number of activated experts is too high, the presence of the routing mechanism can perturb the overall model capability if there isn't effective collaboration among the experts. When the number of experts is one, the model has only a single choice. This selection mode is too restrictive and can even lead to negative consequences. This

TABLE I
CORRECT REPAIR RESULTS FOR EACH VULNERABILITY TYPE .

Vulnerability Type	MoEFix	RLRep	sGuard+	D4C	P-EPR	Claude-3.7-Sonnet	LLaMA3.2-3B	sGuard	SmartFix
	Corr/Total (%)	Corr/Total (%)	Corr/Total (%)	Corr/Total(%)	Corr/Total(%)	Corr/Total (%)	Corr/Total (%)	Corr/Total (%)	Corr/Total (%)
Reentrancy	**75/84 (89.28%)**	53/84 (63.09%)	43/84 (51.19%)	47/84(55.95%)	30/84(35.71%)	35/84(41.67%)	26/84(30.95%)	15/84 (17.85%)	27/84 (32.14%)
Delegatecall	**43/52 (82.69%)**	29/52 (55.76%)	31/52 (59.61%)	29/52(55.76%)	18/52(34.62%)	27/52(51.92%)	23/52(44.23%)	-	-
Timestamp	**33/38 (86.84%)**	27/38 (71.05%)	22/38 (57.89%)	24/38(63.16%)	14/38(36.84%)	21/38(55.26%)	14/38(36.84%)	-	-
Integer overflow	**107/130 (82.30%)**	79/130 (60.76%)	85/130 (65.38%)	57/130(43.85%)	53/130(40.77%)	74/130(56.92%)	55/130(42.31%)	25/130(19.23%)	62/130 (47.69%)
Tx.origin	**56/61 (91.80%)**	47/61 (77.04%)	43/61 (70.49%)	42/61(68.85%)	44/61(72.13%)	39/61(63.93%)	30/61(49.18%)	39/61 (63.93%)	44/61 (72.13%)
Total	**314/365 (86.02%)**	235/365 (64.38%)	224/365 (61.40%)	199/365(54.52%)	159/365(43.56%)	196/365(53.70%)	148/365(40.55%)	79/275 (28.73%)	133/275 (48.36%)

is because if an inappropriate expert is chosen for processing, the MoE model cannot function effectively and may even produce side effects. Furthermore, when the number of active experts is equivalent to the total number of experts, the MoE model degenerates into a Dense model. However, since the model still applies the routing mechanism, its performance significantly degrades, even performing worse than a standard Dense model. Regarding the total number of experts, our configuration (which includes two experts for each type of vulnerability repair) means that the setting with a total of 10 experts offers a more flexible selection for the router compared to the setting with 5 experts. Moreover, it's worth noting that selecting the wrong expert doesn't imply the model completely lacks repair capability; rather, performance is somewhat affected. Therefore, we strive to maximize the balance between repair performance and resource utilization when making expert selections.

Overall, when the total number of experts is constant, increasing the number of activated experts does not lead to a significant performance improvement; in fact, excessive activation can even have negative consequences.

Answer to RQ2: Considering both performance and efficiency, we opt for a Top2 (K=2) setting. When the total number of experts remains constant, increasing the count of activated experts does not lead to a significant performance improvement. In fact, over-activation can even have a negative impact.

C. RQ3:Effectiveness of MoE Framework

To investigate the effectiveness of the MoE framework and our training methodology, we train three variants: Dense , Router$_{random}$ and w/o cpt. The Dense variant uses the same base model as the MoE experts, LLaMA3.2-3B, for training. Simply put, the dense model is obtained by directly fine-tuning on the entire training dataset. To validate the effectiveness of the router, we initialize the router randomly and do not train it. The remaining components of the model are trained according to our established methodology, resulting in the $Router_{random}$ variant. The "w/o cpt" represents the Model without Continual Pre-Training. As shown in Table II , the performance results for the three variants are presented.

TABLE II
PERFORMANCE METRICS FOR DIFFERENT VARIANTS

Type	Dense	Router$_{random}$	w/o cpt	MoEFix
Reentrancy	72.62%	34.52%	84.52%	89.28%
Delegatecall	59.62%	44.23%	76.92%	82.69%
Timestamp	76.31%	42.10%	78.95%	86.84%
Integer overflow	76.15%	33.84%	72.31%	82.30%
Tx.origin	88.52%	62.30%	90.16%	91.80%
Total	75.07%	41.09%	79.73%	**86.02%**

First, for the Dense model, the results reveal a clear and significant performance degradation compared to MoEFIx, with an overall performance drop of 10.95%. The most severe declines in vulnerability repair performance occur for Reentrancy and Delegatecall vulnerabilities, decreasing by 16.66% and 23.07%, respectively. We analyze a reason for this. These two types of vulnerabilities share fundamental similarities in their risk patterns: both involve external contract calls, and the core risk stems from incorrect function call sequencing, especially concerning the timing of state changes. Since capturing and differentiating the characteristics of these two types of vulnerabilities is inherently challenging, subsequent vulnerability repair also places high demands on the model. Therefore, for similar vulnerabilities, simply fine-tuning with a Dense model struggles to effectively learn and distinguish these subtle but critical patterns. For Router$_{random}$, the results reveal a significant decrease in performance across all five vulnerability types. Overall, Router$_{random}$ performs 44.93% worse than MoEFix. Within the MoE framework, each expert is trained to handle specific vulnerabilities, and the routing mechanism ensures that vulnerabilities are directed to the appropriate expert for processing. When the router assigns vulnerabilities randomly, they may be sent to a suboptimal expert, which prevents them from being correctly repaired.

The performance for all types of vulnerabilities decreased for the w/o cpt variant. Among these, the performance for Reentrancy, Delegatecall, and Tx.origin vulnerabilities saw a smaller decline. Compared to the significant drop in the random variant, the performance for the w/o cpt variant decreased

by 4.76%, 5.77%, and 1.64%, respectively. This indicates that repairing these vulnerabilities relies more on the fine-grained learning that occurs during the expert stage, which enables the model to better recognize their core characteristics and repair patterns. In contrast, the removal of the Continual Pre-training process leads to a relatively more significant performance drop for Integer overflow and Timestamp dependency vulnerabilities. We attribute this to the fact that Integer overflow detection requires a deep understanding of Solidity's type system and operational rules. Similarly, timestamp dependency requires Continual Pre-Training to capture the semantics of the block timestamp. Thus, this prior knowledge is crucial for the model's performance.

Answer to RQ3: The experiments demonstrate the effectiveness of the MoE framework and Continual Pre-Training within our method. MoEFix consistently outperforms the Dense model. Moreover, performance degrades to varying degrees with random router and without Continual Pre-training.

TABLE III
COMPUTATIONAL AND TIME FOR DIFFERENT VARIANTS

Model	TFLOPs	Train Time	Infer Time
MoEFix-10top1	∼0.67M	∼1.84h	∼21s
MoEFix-10top2	∼0.73M	∼2.05h	∼34s
MoEFix-10top4	∼0.81M	∼2.34h	∼52s
MoEFix-5top1	∼0.31M	∼0.89h	∼19s
MoEFix-5top2	∼0.35M	∼1.02h	∼31s

D. RQ4: Cost of MoEFix

To thoroughly assess the overhead and time consumption of MoEFix, we constructed several variants. We used the notation MtopN(where M and N are integers) to denote a total of M experts with N activated experts. As shown in the Table III, we recorded the computational overhead and time cost for these variants and the standard configuration.

The experiments show that when the total number of experts remains constant, the number of activated experts does not significantly impact training time or computational overhead. This can be attributed to our training process, where we perform designated training for the experts. The majority of the training time and computational cost are concentrated in this expert training phase. Differences in training time and cost primarily arise from other components, such as the router. Although the router is influenced by the number of activated experts, its contribution to the overall training process is minor, especially since the expert layers are frozen at this stage. Consequently, when the total number of experts is the same, an identical number of experts must be trained, regardless of how many are activated. In contrast, when the total number of experts changes (e.g., from 10 to 5), the training time and overhead change significantly. However, during model inference, the difference in inference time is notable. This is because a different number of activated experts means a different number of activated parameters (e.g., activating one expert corresponds to 3B parameters, while activating two corresponds to 6B). Inference time has a strong correlation with the number of parameters, which explains why different numbers of activated experts lead to significant differences in inference time.

Answer to RQ4: With the same total number of experts, the number of activated experts does not significantly affect training time or computational overhead. However, there is a marked difference in inference time.

TABLE IV
ROUTER ACCURACY FOR EACH VULNERABILITY TYPE

Type	Router Acc	Succ/Corr	Succ/Wrong
Reentrancy	94.04% (79/84)	74/79	1/5
Delegatecall	78.85% (41/52)	37/41	6//11
Timestamp	86.84% (33/38)	31/33	2/5
Integer overflow	83.85% (109/130)	99/109	8/21
Tx.origin	95.08%(58/61)	55/58	1/3
Total	87.67%(320/365)	296/320	18/45

E. RQ5: Accuracy of Router Selection

To investigate the impact of routing accuracy, we conducted a deeper analysis of the experts. A routing decision is considered correct if its TopK selection includes at least one correct expert. The **Router Acc** represents the percentage of correct routing decisions.

As shown in the Table IV, we have compiled the routing selection distribution across different vulnerability types. MoEFix achieved a routing total accuracy of 87.67%, and notably, it reached 95.08% for Tx.origin. However, the selection accuracy for Delegatecall vulnerabilities was relatively low, at 78.84%. Upon examining the incorrect expert selections, we found that the majority were routed to the Reentrancy expert. This can be attributed to the inherent similarity in their risk patterns: both involve external contract calls and their core risk stems from incorrect function call order, particularly regarding the timing of state changes. To further analyze the impact of routing accuracy on model performance, we compared the model's performance when selecting the correct expert versus an incorrect one. As shown in the Table IV, the **Succ/Corr** is the ratio of successfully repaired vulnerabilities when the router makes a correct selection. It is clear that model repairing accuracy drops significantly when an incorrect expert is chosen. The success rate for Reentrancy, for instance, is only 20%. This decrease is expected, as each expert was fine-tuned on a specific type of vulnerability and therefore lacks

the ability to recognize features and handle other vulnerability types. While some incorrectly selected experts can still achieve a successful repair, this fundamentally relies on the inherent capabilities of the base model itself.

Answer to RQ5: In general, the router plays a critical role in MoEFix, significantly impacting vulnerability repairing performance. The model repairing accuracy will drop significantly when an incorrect expert is chosen.

VI. Threats to Validity

Internal Validity: To mitigate the randomness inherent in large language models, we used the "mimic-in-the-background" prompting strategy from Sun et al. [37]. Specifically, we set the temperature to 0.1 to increase determinism and had the model generate five independent responses for each input. The most frequent response was then chosen as the final output, effectively reducing the impact of randomness on our experimental results. To prevent the potential data leakage existed between the training and test sets, we used similarity filtering, proportional splitting, and manual review. For potential data leakage in large language models, we applied Decoding Matching to each test sample to ensure the test set was not included in the underlying LLM's internal knowledge.

External Validity: MoEFix relies on datasets of various vulnerability types, particularly for the crucial step of subspace division. However, the construction of these datasets involves multiple processing steps. Although we perform vulnerability detection and filtering using different tools (static analysis tools and LLMs), and concurrently leverage LLMs for manual repair and verification, we cannot fully guarantee the absolute accuracy of every step. Moreover, due to limitations in both dataset availability and computational resources, the scope of our method is currently restricted to specific types of vulnerabilities. Future research will focus on developing more efficient data construction methods to address a broader range of smart contract vulnerabilities.

VII. Related Work

A. Vulnerability Repair for Smart Contracts

In the realm of smart contract vulnerability repair, current research shows significant progress. SCRepair [10] stands as the first search-based Automated Program Repair (APR) tool for smart contracts. In contrast to SCRepair, sGuard [12] employs a source-code-based, semantic-driven approach to ensure that contract vulnerabilities are correctly addressed. Furthermore, sGuard is unique in its use of a self-contained detection module for vulnerability localization. However, sGuard's reliance on symbolic execution for detection makes it inaccurate and slow. This impacts the correctness of its repairs and patches inserted by sGuard may introduce new vulnerabilities or impact the original business logic. Therefore, Gao et al. [13] proposed sGuard+, a machine learning-based approach for smart contract vulnerability repair. This method addresses the issue of weak patches that cannot fully defend against vulnerabilities and improves repair correctness through an accurate machine learning detection method and corresponding localization algorithm. Subsequently, Guo et al. [11] introduced RLRep, a reinforcement learning-based method for smart contract vulnerability repair which provides repair action suggestions based on the vulnerable smart contract without any supervision.

B. Mixture-of-Experts (MoE) in Software Engineering

Mixture of Experts (MoE) [17] is a hybrid model consisting of multiple sub-models, known as experts, which are integrated together. The key concept of MoE is the use of a router to determine the token set that each expert handles, thereby reducing interference between different types of samples [24]. In the field of software engineering, research on Mixture of Experts models remains relatively novel. Omer et al. developed ME-SFP [55], a software defect prediction method that uses the MoE. This method trains its experts with decision trees and multilayer perceptrons and uses a Gaussian Mixture Model to select the appropriate expert. Aditya and Santosh [56] explored two variations of the MoE approach for defect prediction tasks: Implicit Mixture of Experts (IMoE) and Explicit Mixture of Experts (EMoE). The IMoE method randomly partitions the input data into several subspaces based on the error function employed, with local experts specializing in each subspace. In contrast, the EMoE method explicitly divides the input data into several subspaces using clustering techniques prior to the training process. Wu et al. [57] applied the MoE model to code smell detection by leveraging the experts to select static analysis tools. Yang et al. [16] propose MoEVD for vulnerability detection, which partitions the input space according to CWE types.

VIII. Conclusion

In this paper, we present MoEFix, a new smart contract vulnerability repair method based on a Mixture-of-Experts framework. MoEFix provides specialized repair solutions for different types of vulnerabilities by using dedicated expert models and a routing mechanism. We also redesigned the vulnerability repair process to better leverage the power of large language models and created a specialized dataset for smart contract vulnerability repair. Experimental results show that MoEFix outperforms existing methods in repairing various vulnerabilities, with an overall accuracy improvement of 21.64%. We believe that MoEFix offers a new research direction for smart contract vulnerability repair and has the potential to enhance the security and reliability of smart contracts in practical applications.

IX. Acknowledgment

This work was supported by the National Key Research and Development Program of China (No. 2023YFB3307202) and the Alliance of International Science Organizations Collaborative Research Program (No. ANSO-CR-KP-2022-03).

REFERENCES

[1] M. Swan, *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015.

[2] T. Hewa, M. Ylianttila, and M. Liyanage, "Survey on blockchain based smart contracts: Applications, opportunities and challenges," *Journal of Network and Computer Applications*, vol. 177, p. 102857, 2021.

[3] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2084–2106, 2019.

[4] V. Dhillon, D. Metcalf, M. Hooper, V. Dhillon, D. Metcalf, and M. Hooper, "The dao hacked," *blockchain enabled applications: Understand the blockchain Ecosystem and How to Make it work for you*, pp. 67–78, 2017.

[5] M. I. Mehar, C. L. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. M. Kim, and M. Laskowski, "Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack," *Journal of Cases on Information Technology (JCIT)*, vol. 21, no. 1, pp. 19–32, 2019.

[6] M. Alharby and A. Van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *arXiv preprint arXiv:1710.06372*, 2017.

[7] M. Di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *2019 IEEE international conference on decentralized applications and infrastructures (DAPPCON)*. IEEE, 2019, pp. 69–78.

[8] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "Smartshield: Automatic smart contract protection made easy," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 23–34.

[9] M. Rodler, W. Li, G. O. Karame, and L. Davi, "{EVMPatch}: Timely and automated patching of ethereum smart contracts," in *30th usenix security symposium (USENIX Security 21)*, 2021, pp. 1289–1306.

[10] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury, "Smart contract repair," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–32, 2020.

[11] H. Guo, Y. Chen, X. Chen, Y. Huang, and Z. Zheng, "Smart contract code repair recommendation based on reinforcement learning and multi-metric optimization," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–31, 2024.

[12] T. D. Nguyen, L. H. Pham, and J. Sun, "Sguard: towards fixing vulnerable smart contracts automatically," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1215–1229.

[13] C. Gao, W. Yang, J. Ye, Y. Xue, and J. Sun, "sguard+: Machine learning guided rule-based automated vulnerability repair on smart contracts," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–55, 2024.

[14] S. So and H. Oh, "Smartfix: Fixing vulnerable smart contracts by accelerating generate-and-verify repair using statistical models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 185–197.

[15] S. Wan, J. Saxe, C. Gomes, S. Chennabasappa, A. Rath, K. Sun, and X. Wang, "Bridging the gap: A study of ai-based vulnerability management between industry and academia," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. IEEE, 2024, pp. 80–87.

[16] X. Yang, S. Wang, J. Zhou, and W. Zhu, "Moevd: Enhancing vulnerability detection by mixture-of-experts (moe)," *arXiv preprint arXiv:2501.16454*, 2025.

[17] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural computation*, vol. 3, no. 1, pp. 79–87, 1991.

[18] S. E. Yuksel, J. N. Wilson, and P. D. Gader, "Twenty years of mixture of experts," *IEEE transactions on neural networks and learning systems*, vol. 23, no. 8, pp. 1177–1193, 2012.

[19] S. M. Imtiaz, A. Singh, F. Batole, and H. Rajan, "Irepair: An intent-aware approach to repair data-driven errors in large language models," *arXiv preprint arXiv:2502.07072*, 2025.

[20] J. Xu, Y. Fu, S. H. Tan, and P. He, "Aligning the objective of llm-based program repair," *arXiv preprint arXiv:2404.08877*, 2024.

[21] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[22] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "Easyflow: Keep ethereum away from overflow," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 23–26.

[23] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: a survey," *arXiv preprint arXiv:1908.08605*, 2019.

[24] B. Lin, Z. Tang, Y. Ye, J. Cui, B. Zhu, P. Jin, J. Huang, J. Zhang, Y. Pang, M. Ning *et al.*, "Moe-llava: Mixture of experts for large vision-language models," *arXiv preprint arXiv:2401.15947*, 2024.

[25] T. Zhu, X. Qu, D. Dong, J. Ruan, J. Tong, C. He, and Y. Cheng, "Llama-moe: Building mixture-of-experts from llama with continual pre-training," *arXiv preprint arXiv:2406.16554*, 2024.

[26] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[27] C. S. Xia and L. Zhang, "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*, 2022, pp. 959–971.

[28] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481.

[29] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1646–1656.

[30] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1506–1518.

[31] D. McIntyre, "Bridging the gap between research and practice," *Cambridge journal of education*, vol. 35, no. 3, pp. 357–382, 2005.

[32] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[33] B. Mueller, "Mythril-reversing and bug hunting framework for the ethereum blockchain," 2017.

[34] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[35] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[36] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[37] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," *Proc. IEEE/ACM ICSE*, 2024.

[38] S. Hu, T. Huang, F. Ilhan, S. F. Tekin, and L. Liu, "Large language model-powered smart contract vulnerability detection: New perspectives," in *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2023, pp. 297–306.

[39] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[40] Anthropic, "Claude Large Language Model," [Year of Access], e.g., 2025, large language model developed by Anthropic. Accessed on [Month Day, Year], e.g., May 26, 2025. [Online]. Available: https://www.anthropic.com/claude

[41] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[42] K. Alrashedy and A. Binjahlan, "Language models are better bug detector through code-pair classification," *arXiv preprint arXiv:2311.07957*, 2023.

[43] D. Cifarelli, L. Boiardi, A. Puppo, and L. Jovanovic, "Safurai-csharp: Harnessing synthetic data to improve language-specific code llm," *arXiv preprint arXiv:2311.03243*, 2023.
[44] K. I. Roumeliotis, N. D. Tselikas, and D. K. Nasiopoulos, "Llama 2: Early adopters' utilization of meta's new open-source pretrained model," 2023.
[45] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 415–427.
[46] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 259–269.
[47] A. Storhaug, J. Li, and T. Hu, "Efficient avoidance of vulnerabilities in auto-completed smart contract code using vulnerability-constrained decoding," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 683–693.
[48] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *arXiv preprint arXiv:1701.06538*, 2017.
[49] L. Yu, S. Chen, H. Yuan, P. Wang, Z. Huang, J. Zhang, C. Shen, F. Zhang, L. Yang, and J. Ma, "Smart-llama: Two-stage post-training of large language models for smart contract vulnerability detection and explanation," *arXiv preprint arXiv:2411.06221*, 2024.
[50] W. Zhong, C. Li, K. Liu, T. Xu, J. Ge, T. F. Bissyandé, B. Luo, and V. Ng, "Practical program repair via preference-based ensemble strategy," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.
[51] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.
[52] Y. Zheng, R. Zhang, J. Zhang, Y. Ye, and Z. Luo, "Llamafactory: Unified efficient fine-tuning of 100+ language models," *arXiv preprint arXiv:2403.13372*, 2024.
[53] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 3505–3506.
[54] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.
[55] A. Omer, S. S. Rathore, and S. Kumar, "Me-sfp: A mixture-of-experts-based approach for software fault prediction," *IEEE Transactions on Reliability*, vol. 73, no. 1, pp. 710–725, 2023.
[56] A. Shankar Mishra and S. Singh Rathore, "Implicit and explicit mixture of experts models for software defect prediction," *Software Quality Journal*, vol. 31, no. 4, pp. 1331–1368, 2023.
[57] D. Wu, F. Mu, L. Shi, Z. Guo, K. Liu, W. Zhuang, Y. Zhong, and L. Zhang, "ismell: Assembling llms with expert toolsets for code smell detection and refactoring," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1345–1357.