SAEL: Leveraging Large Language Models with Adaptive Mixture-of-Experts for Smart Contract Vulnerability Detection

Lei Yu[†‡1], Shiqi Cheng[†1], Zhirong Huang[†‡], Jingyuan Zhang[†‡], Chenjie Shen[†‡],
Junyi Lu[†‡], Li Yang[†*], Fengjun Zhang[†§*], Jiajia Ma[†]
[†]Institute of Software, Chinese Academy of Sciences, Beijing, China
[‡]University of Chinese Academy of Sciences, Beijing, China
[§]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
{yulei2022, chengshiqi, huangzhirong2022, zhangjingyuan2023, lujunyi2022}@iscas.ac.cn,
shenchenjie22@mails.ucas.ac.cn, {yangli2017, fengjun, majiajia}@iscas.ac.cn

***Abstract*—With the increasing security issues in blockchain, smart contract vulnerability detection has become a research focus. Existing vulnerability detection methods have their limitations: 1) Static analysis methods struggle with complex scenarios. 2) Methods based on specialized pre-trained models perform well on specific datasets but have limited generalization capabilities. In contrast, general-purpose Large Language Models (LLMs) demonstrate impressive ability in adapting to new vulnerability patterns. However, they often underperform on specific vulnerability types compared to methods based on specialized pre-trained models. We also observe that explanations generated by general-purpose LLMs can provide fine-grained code understanding information, contributing to improved detection performance.**

Inspired by these observations, we propose SAEL, a LLM-based framework for smart contract vulnerability detection. First, we design prompts targeting specific smart contract vulnerabilities to guide general-purpose LLMs in detecting vulnerabilities and providing explanations. The detection results generated by LLMs serve as prediction features. Then, we employ prompt-tuning on CodeT5 and T5 respectively to process contract code and explanations, enhancing model performance on specific tasks. To leverage the strengths of each component, we introduce Adaptive Mixture-of-Experts, a dynamic architecture for smart contract vulnerability detection. This mechanism dynamically adjusts feature weights through a Gating Network, which selects the most relevant features by applying TopK filtering and Softmax normalization, and a Multi-Head Self-Attention mechanism, which enhances cross-feature relationships by processing multiple attention heads in parallel. This design ensures that prediction results for LLMs, explanation features, and contract code features are effectively integrated through gradient optimization. The loss function focuses on the independent prediction performance of each feature and the overall performance of weighted predictions. Experimental results show that SAEL outperforms existing methods in detecting various vulnerabilities.

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

I. Introduction

Blockchain technology has grown rapidly and become popular across various sectors due to its decentralized nature [1]. As a significant innovation, blockchain allows for the creation of secure, decentralized, and distributed digital ledgers that record transactions [2]. By using cryptographic methods, blockchain ensures that each transaction is secure and verified, making it a highly dependable technology [3], [4]. In particular, graph analysis techniques, including detecting criminal communities [5] and multi-scale anomaly detection [6], play a crucial role in identifying irregular patterns and enhancing the security of blockchain networks. Smart contracts are automated programs on the blockchain that let developers create rules for managing digital assets like cryptocurrency. They automatically execute when specific conditions are met. Once these programs are deployed on the blockchain, they are permanent [7]. However, the unchangeable and complex nature of smart contracts means that their security challenges are increasingly evident [7]. The infamous DAO attack [8]–[12] illustrates the severe consequences of such vulnerabilities. This attack led to the unauthorized transfer of Ethereum worth 60 million dollars, causing significant disruptions in the blockchain community [13], [14]. This highlights the critical need to enhance the security of smart contracts to avoid such crushing outcomes in the future.

Researchers have developed various techniques to identify vulnerabilities in smart contracts. One popular approach is symbolic execution, which is implemented in tools such as Oyente [15], Mythril [16], Osiris [17], and Manticore [18]. Another commonly used technique is static analysis, which is employed by tools like Slither [19] and SmartCheck [20]. However, these methods rely solely on fixed patterns and have poor generalizability. As shown in Fig. 1, Slither incorrectly identifies a reentrancy vulnerability by detecting external calls and state variable modifications, but fails to consider the onlyOwner modifier that prevents such attacks, resulting in a false positive. Clear [21] adopts a Contrastive Learning (CL) model to learn complex relationships between contracts. Zhuang et al. [22] and Luo et al. [23] introduce a graph neural network-based approach that converts smart contract code into graph representations. However, the complexity of the graph

[1]Lei Yu and Shiqi Cheng contributed equally to this work.
[*]Corresponding authors: Li Yang and Fengjun Zhang.

structures employed in these techniques makes them difficult to reproduce and effectively represent programs. Peculiar [24] and PSCVFinder [25] achieve precise detection of smart contract vulnerabilities by fine-tuning pre-trained models. As shown in Fig. 2, their detection capabilities for reentrancy and timestamp dependency vulnerabilities are superior to directly using general-purpose LLMs. However, they may struggle to handle scenarios not well-represented in the training data, exhibiting poor generalizability. As illustrated in Fig. 1, specialized pre-trained models (PSCVFinder and Peculiar) fail to understand the role of the onlyOwner modifier in Fig. 1, resulting in false positives. In contrast, general-purpose LLM correctly understands the function of the onlyOwner modifier, recognizing that the refund function is protected by the onlyOwner modifier. Even if investor.call.value(amount)() triggers a callback function of a malicious contract, that malicious contract cannot call the refund function again because it is not the contract owner. Consequently, general-purposed LLM correctly concludes that the contract does not contain a reentrancy vulnerability. This comparison highlights the complementary strengths of different approaches: general-purpose LLMs demonstrate superior capability in adapting to new vulnerability patterns, while specialized pre-trained models provide high performance for well-defined vulnerability types.

We observe that explanations generated by general-purpose LLMs can serve as fine-grained code understanding information, improving the performance of smart contract vulnerability detection. As shown in Fig. 3, for a specific contract, the general-purpose LLM (Qwen1.5-72B-Chat) identified a timestamp dependency vulnerability in the contract and provided explanations about the usage of block.timestamp (only used in a require statement), the structure and security checks of the function (constrained by two require statements), and the usage of state variables (read-only). These details allow for the analysis that the timestamp is only used for simple comparison, does not directly affect the state, and state variables are not modified. These inferences based on detailed explanations suggest that the contract may not actually contain a timestamp dependency vulnerability, which is consistent with the label. This insight motivates us to incorporate the explanation generated by general-purpose LLMs into smart contract vulnerability detection models.

Based on these findings, we propose SAEL, an LLM-based smart contract vulnerability detection framework. First, we design prompts tailored to specific smart contract vulnerabilities to guide LLMs in analyzing smart contract code, detecting vulnerabilities, and providing fine-grained explanations. This addresses the poor generalizability of static analysis and pre-trained methods. The predictions generated by LLMs serve as predictive features. We conducted a comprehensive empirical study comparing the performance of different LLMs on reentrancy and timestamp dependency vulnerability datasets, as shown in Fig. 2. We found that Qwen1.5-72B-Chat exhibits performance closest to GPT-4-Turbo in smart contract vulnerability detection. To balance performance and overhead, we employ it as the LLM model in this study. Yu et al. [25] demonstrated that using prompt tuning outperforms fine-tuning on smart contract vulnerability detection tasks, while CodeT5 [26] achieves better results compared to other models such as GraphCodeBERT [27]. Therefore, we adopt prompt-tuning instead of fine-tuning on CodeT5 [26] and T5 [28] to handle contract code and explanations, respectively. Finally, to fully leverage the advantages of each component and effectively incorporate explanations into the detection model, we introduce Adaptive Mixture-of-Experts, a dynamic framework that integrates raw code features, LLM-generated explanations, and LLM predictions. Unlike static ensemble methods, Adaptive Mixture-of-Experts dynamically adjusts feature weights through a Gating Network, which selects the most relevant features by applying a TopK mechanism to filter the most significant feature dimensions, followed by Softmax normalization to produce a gating vector. In addition, a Multi-Head Self-Attention mechanism captures complex cross-dimensional relationships among the features by computing attention scores across multiple heads and combining them to enhance contextual understanding. These components ensure robust and adaptive detection by optimizing a learnable loss function through gradient descent, allowing the model to adaptively balance the contributions of different feature types. The loss function focuses on the independent predictive performance of each feature and the overall performance of the weighted predictions. By minimizing the loss function, we can obtain the optimal combination of feature weights.

We evaluated our SAEL framework on over 200,000 real-world smart contracts in the SmartBugs Wild dataset [29] for reentrancy vulnerabilities and the ESC dataset [30]. For integer overflow/underflow and delegatecall vulnerabilities, we select two largest publicly available vulnerability dataset for smart contracts [31], [32] and mix them. The results demonstrate that SAEL outperforms state-of-the-art methods, with F1 scores 2.33%, 3.16%, 10.67%, and 13.32% higher for reentrancy, timestamp dependency, integer overflow/underflow, and delegatecall vulnerabilities, respectively. Moreover, SAEL exhibits strong zero-shot capabilities across the four vulnerability types.

The main contributions of this paper are as follows:

- We are among the first to incorporate LLM-generated explanations as features to enhance smart contract vulnerability detection, demonstrating their impact on improving detection performance.
- To fully harness the strengths of each feature, we introduce Adaptive Mixture-of-Experts for smart contract vulnerability detection. It dynamically adjusts the weights assigned to the prediction results based on different features.
- Our approach sets new state-of-the-art performance in detecting reentrancy, timestamp dependency, integer overflow/underflow, and delegatecall vulnerabilities in smart contracts.

All source code and data in this study are publicly available at [33].

II. Background and Motivation

A. Problem Statement

We propose an automated approach to detect vulnerabilities in individual smart contract functions. Our method assigns a label $\hat{y}$ to each function f, where $\hat{y} = 1$ indicates a vulnerability and $\hat{y} = 0$ denotes security. We focus on four key vulnerability types.

Reentrancy vulnerability occurs when a contract calls an external contract or sends Ether before completing all necessary internal state changes. An attacker can exploit this vulnerability by repeatedly calling the vulnerable function before the original call is completed, potentially leading to unexpected behavior such as multiple withdrawals of funds.

Timestamp dependence vulnerability occurs when smart contracts rely on block timestamps for critical operations. Miners can manipulate these timestamps, potentially compromising contract integrity and leading to financial losses. This vulnerability often affects contracts using timestamps for random number generation or key decision-making processes.

Integer Overflow/Underflow occurs when the result of an arithmetic operation exceeds the storage range of the variable. In an overflow, the value "wraps around" to the minimum value for that type, while in an underflow, it "wraps around" to the maximum value. This can lead to unexpected contract behavior such as incorrect balances or out-of-control loops.

Delegatecall is a low-level function call that allows a contract to dynamically load code from another contract. While this provides powerful upgradeability, it can lead to severe security vulnerabilities if used improperly. The main risk is that the called contract executes in the context of the calling contract and can thus modify the calling contract's storage.

We primarily focus on these four vulnerabilities for the following reasons: (i) Empirical evidence shows that approximately 70% of financial losses in Ethereum smart contract attacks are attributed to these vulnerabilities [34]. (ii) Exisiting works [34]–[36] demonstrates that these vulnerabilities occur with higher frequency in Ethereum smart contracts compared to others.

B. Motivating Examples

In this section, we use two real-world smart contract examples to illustrate the complementary strengths of different approaches in detecting smart contract vulnerabilities: specialized pre-trained models fine-tuned on specific vulnerability datasets, general-purpose Large Language Models (LLMs), and the explanations generated by LLMs.

Example 1: Comparative Analysis Reveals Distinct Strengths of Different Approaches. Static analysis methods (e.g., Slither) rely on predefined vulnerability patterns and may struggle with complex scenarios. As shown in Fig. 1, Slither incorrectly identifies a reentrancy vulnerability by detecting external calls and state variable modifications, without considering the onlyOwner modifier that prevents such attacks. Specialized pre-trained models (e.g., PSCVFinder and Peculiar), fine-tuned on specific vulnerability datasets, demonstrate high performance in detecting known vulnerability types, often outperforming general-purpose LLMs on these specific tasks, as illustrated in Fig. 2. However, they may struggle with scenarios not well-represented in their training data, showing poor generalization and failing to understand the onlyOwner modifier in Fig. 1. In contrast, general-purpose LLMs correctly comprehend the role of the onlyOwner modifier, leading to the accurate conclusion that the smart contract does not contain a reentrancy vulnerability. This comparison highlights the complementary strengths of different approaches: general-purpose LLMs offer superior capability in adapting to new vulnerability patterns, while specialized pre-trained models provide high performance for well-defined vulnerability types.

Example 2: Explanations Generated by LLMs Can Infer the Correct Answer. Our case studies reveal that explanations generated by general-purpose Large Language Models (LLMs) can provide valuable insights for improving smart contract vulnerability detection. For example, consider the contract in Fig. 3, the LLM incorrectly states that the contract may have a timestamp dependency vulnerability. Despite predicting a timestamp dependency vulnerability in the smart contract, the LLM provided relatively comprehensive code analysis as shown in Fig. 3: 1) It correctly identified the location where the timestamp is used. 2) It accurately described the structure and security checks of the function. 3) It correctly identified the state variables and how they are used in the function (only read, not modified). Based on the code analysis, we can infer that: 1) The timestamp is only used for simple comparison and does not directly affect the state. 2) The function includes multiple security checks. 3) State variables are not modified. These inferences based on detailed explanations suggest that the contract may not actually contain a timestamp dependency vulnerability, which is consistent with the ground truth. This example demonstrates how LLM-generated explanations can serve as a form of code analysis, potentially refining the initial prediction of the model.

Based on these examples, we observe that while specialized pre-trained models may outperform general-purpose LLMs in detecting specific, well-defined vulnerability types, general-purpose LLMs exhibit superior flexibility in adapting to new or complex vulnerability patterns. Moreover, the explanations generated by LLMs provide an additional layer of analysis that can enhance detection performance, even when the initial prediction is incorrect. Combining these approaches with LLM-generated explanations can complement each other and enhance detection performance.

III. Approach

The overall workflow of SAEL is shown in Fig. 4. The SAEL framework consists of three key modules: the design of prompt templates, T5-based Prompt-tuning, and Adaptive Mixture-of-Experts (MoE). Each module is carefully designed to enhance the performance of smart contract vulnerability detection.

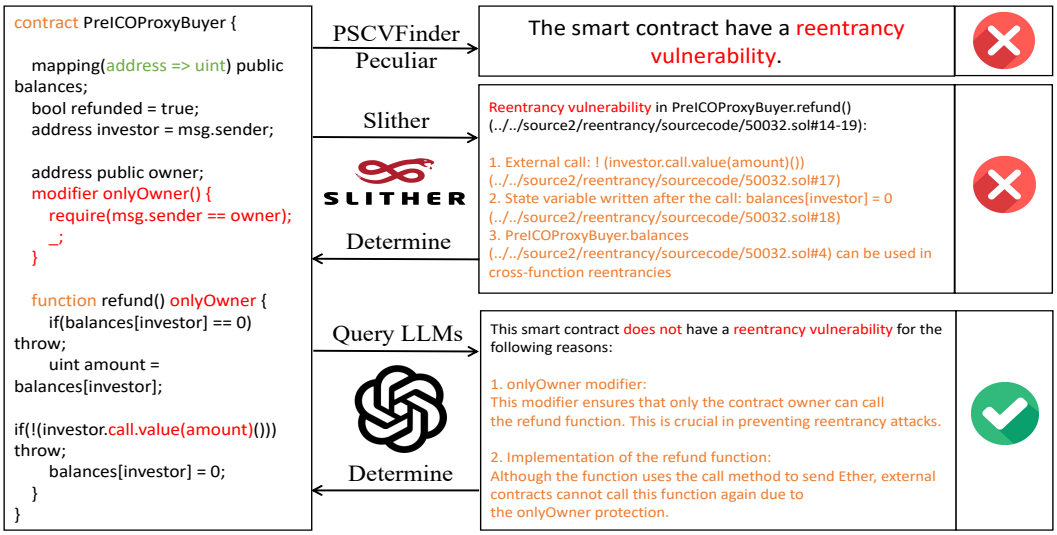

Fig. 1. Motivation examples of different methods on smart contract vulnerability detection task.

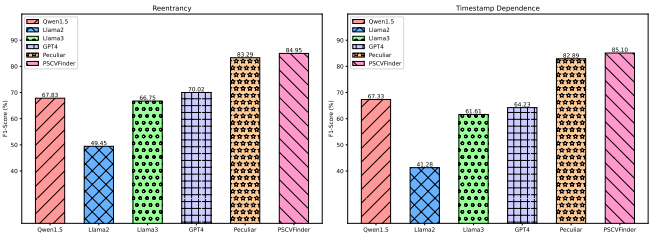

Fig. 2. Performance comparison of specialized pre-trained models and general-purpose LLMs.

```
1  contract Safe {
2      address public owner;
3      uint256 public lock;
4      function withdrawal( address to, uint value)
5      returns (bool){
6      require(msg.sender == owner);
7      uint256 time = block.timestamp;
8      require(time >= lock);
9      require(to != address(0));
10     return true;
11     }
12 }
```

This contract may have a timestamp dependency vulnerability:
1. Timestamp Usage:
Line 8 uses block.timestamp: require(time >= lock);
2. Function Structure:
Includes ownership check, timestamp comparison, and address validation, relying on external inputs and block information.
3. State Variable Impact:
The contract defines two public state variables: 'owner' (address) and 'lock' (uint256). In the withdrawal function, these variables are only read for access control and timestamp comparison.

Fig. 3. The case related to timestamp where LLMs made incorrect predictions.

A. The Design of Prompt Templates

Smart Contract Vulnerability Detection methods [37]–[39] based on LLMs typically employ models like GPT-4-Turbo, but the cost of calling their APIs is high. To reduce costs while maintaining detection performance, we choose the open-source Qwen1.5-72B-Chat as the LLM model for our work. We evaluated the performance of Qwen1.5-72B-Chat and GPT-4-Turbo on smart contract vulnerability detection tasks and found that Qwen1.5-72B-Chat achieves comparable detection effectiveness with significantly lower computational cost, making it an ideal choice for our framework.

For four types of vulnerabilities (e.g., reentrancy, timestamp dependence), we carefully designed prompt templates. Unlike existing works [37]–[39], which simply provide general vulnerability descriptions or assign the identity of a "smart contract security expert" to LLMs, our prompt templates are tailored for each specific vulnerability type. For instance, as shown in Fig. 5, the prompt template for detecting reentrancy vulnerabilities includes: 1. A detailed definition of reentrancy vulnerabilities and necessary background knowledge. 2. A description of typical characteristics associated with reentrancy vulnerabilities (e.g., external calls in loops). 3. Instructions for analyzing the given code, identifying vulnerabilities, explaining their causes, and locating problematic code sections.

To ensure structured and logical reasoning, the prompt adopts a Chain-of-Thought reasoning process, guiding the LLM step-by-step to: (1) understand the vulnerability's definition and characteristics, (2) analyze the code structure, (3) identify potential vulnerabilities, and (4) explain the causes or provide evidence of security. This process helps the LLM focus on key aspects of the code and ensures reliable detection results.

Mimic-in-the-Background Approach: To further enhance consistency, we employ the "mimic-in-the-background" approach [40]. Specifically, the LLM generates five responses for the same prompt in the background. The system then selects the most frequently appearing answer, ensuring the final response is both representative and reliable.

B. T5-based Prompt-tuning

To extract semantic information from smart contract code, we utilize two pre-trained models: the code language model CodeT5 [26] and the text language model T5 [28]. CodeT5 processes raw smart contract code, while T5 processes natural language explanations generated by the LLM. Instead of fine-

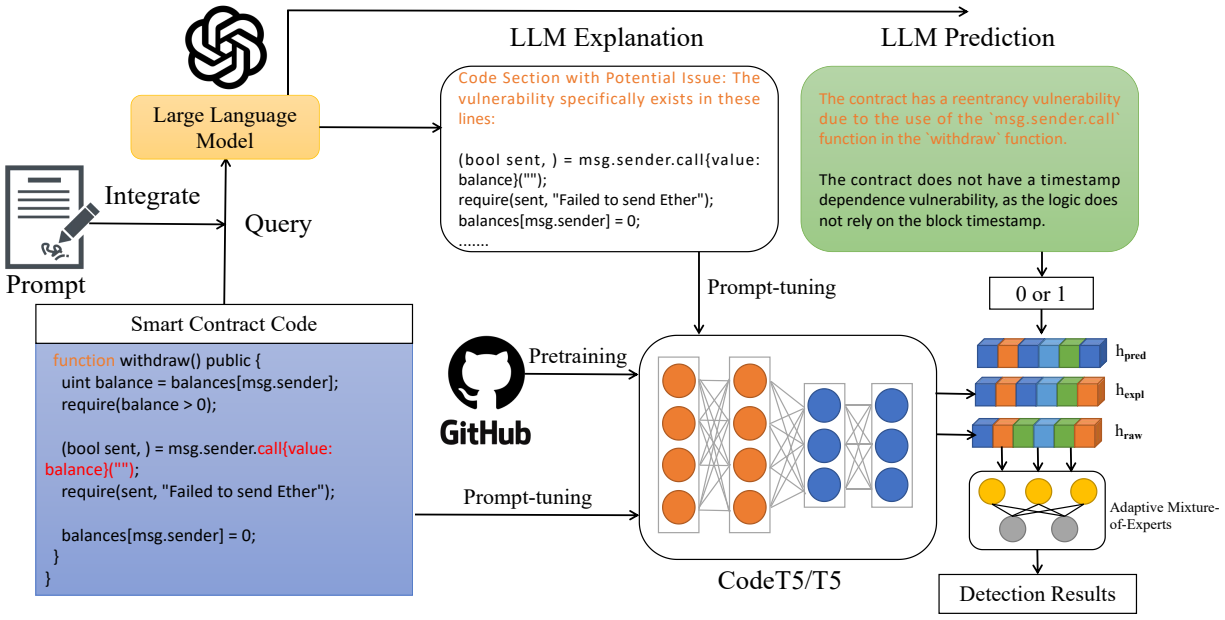

Fig. 4. The overall architecture of the proposed model SAEL.

Prompt Template

System: You are an experienced smart contract vulnerability detector. Your task is to analyze the provided smart contract code and identify potential vulnerabilities. You can mimic answering them in the background five times and provide me with the most frequently appearing answer. Please strictly adhere to the output format specified in the question.

User: Given the following smart contract code:

{contract_code}

Please analyze this smart contract code to determine if it contains the following vulnerability, and provide your reasoning:

Reentrancy Vulnerability: This occurs when a function can be interrupted and called again before its execution is completed, potentially leading to inconsistent data states or unintended fund withdrawals.

For each vulnerability detected:
- Provide a brief description explaining why this portion of code may lead to the vulnerability.
- Identify specific lines or sections of code where potential issues lie.

If no vulnerabilities are found, please explain why the smart contract code is secure in these aspects.

Please provide your response in the following JSON format:

{
"vulnerability_detected": <0 or 1>.
"analysis": "Your analysis goes here."
}

Fig. 5. The Prompt Design for Reentrancy Vulnerability.

tuning, we employ prompt-tuning, which has been shown to outperform fine-tuning on smart contract vulnerability detection tasks [25].

Prompt-tuning learns a continuous task-specific prompt prepended to the input, guiding the model to focus on relevant information. A cloze-style template $f_{\text{prompt}}(x)$ with an input slot $[X]$ and an answer slot $[Z]$ is designed as follows:

$$f_{\text{prompt}}(x) = \text{"[X] The code is [Z]"} \tag{1}$$

A verbalizer V maps label words to the predicted class:

$$V = \begin{cases} \text{Vulnerable:} & \text{[defective, bad]} \\ \text{Secure:} & \text{[clean, perfect]} \end{cases} \tag{2}$$

The outputs of CodeT5 and T5 are denoted as h_{raw} and h_{expl}, respectively:

$$h_{\text{raw}} = \text{CodeT5}_{\text{raw}}(x^{\text{raw}}) \in \mathbb{R}^{N \times d} \tag{3}$$

$$h_{\text{expl}} = \text{T5}_{\text{expl}}(x^{\text{expl}}) \in \mathbb{R}^{N \times d} \tag{4}$$

Additionally, the LLM prediction outputs are encoded into one-hot vectors and transformed into uniform feature embeddings h_{pred}. These three feature representations (h_{raw}, h_{expl}, h_{pred}) serve as inputs to the next module.

Feature Integration: The outputs of CodeT5 (h_{raw}) and T5 (h_{expl}) are concatenated with the LLM prediction embeddings (h_{pred}) to form the combined feature vector $x = [h_{\text{raw}}, h_{\text{expl}}, h_{\text{pred}}]$. This vector serves as the input to the Adaptive Mixture-of-Experts module for dynamic expert selection.

C. Adaptive Mixture-of-Experts

To maximize the strengths of different feature types and improve the adaptability of the detection process, we introduce an Adaptive Mixture-of-Experts (MoE) architecture. This

architecture dynamically selects and combines expert outputs using a gating network and multi-head self-attention mechanisms, ensuring that the most relevant features are utilized for smart contract vulnerability detection.

Gating Network Design. The Gating Network plays a critical role in dynamically assigning weights to each expert based on the combined input features. Specifically, the input vector x is constructed by concatenating raw code embeddings (h_{raw}), explanation embeddings (h_{expl}), and LLM prediction embeddings (h_{pred}). The Gating Network processes the input vector with a multi-layer transformation pipeline, including:

1) **Feature Extraction**: A linear layer reduces the dimensionality of the input vector x, ensuring computational efficiency.
2) **Key Feature Selection**: A TopK function with $k = 3$ is applied to retain only the top 3 most significant features in the vector, improving focus on the most relevant information.
3) **Normalization**: The Softmax function normalizes the retained features, generating the gating vector $G(x)$, which dynamically determines the weight of each expert.

The final formula for the gating vector $G(x)$ is:

$$G(x) = \text{Softmax}(\text{TopK}(H(x), k = 3)) \tag{5}$$

where $H(x)$ represents the feature vector transformed by the Gating Network. The TopK function with $k = 3$ ensures that only the most important features contribute to the final gating vector. This specific value of k was selected to align with our feature space, which comprises three primary types (raw code, explanations, and predictions), setting $k = 3$ ensures each feature type has the opportunity to be considered in the vulnerability detection process.

Multi-Head Self-Attention Mechanism. To further enhance feature representation, we leverage a Multi-Head Self-Attention mechanism. This mechanism enables parallel processing of multiple attention heads, each focusing on a different subspace of the input vector. The steps are as follows:

- Transform the input vector into query, key, and value matrices.
- Compute attention weights by applying the scaled dot product between queries and keys, followed by a Softmax normalization.
- Use the attention weights to compute a weighted sum of the value vectors.
- Concatenate outputs from all attention heads and pass them through a final linear transformation to restore the original dimension.

This mechanism ensures comprehensive exploration of input features while preserving cross-dimensional contextual relationships.

Expert Models and Weighted Accuracy Matrix. The MoE model includes three specialized expert models:

1) **Raw Code Expert**: Focuses on syntactic and structural patterns in h_{raw}, such as loop structures and external function calls.
2) **Explanation Expert**: Processes h_{expl} to extract contextual insights from LLM-generated explanations.
3) **Prediction Expert**: Processes h_{pred} to capture high-level semantic patterns based on LLM predictions.

Each expert outputs a vector O_i representing its prediction confidence. The outputs from all experts are combined into an accuracy matrix M, where each row corresponds to an expert, and each column represents a vulnerability type:

$$M = [O_1, O_2, O_3] \tag{6}$$

The gating vector $G(x)$ is then used to weight the accuracy matrix, producing a weighted accuracy matrix:

$$M_{\text{weighted}} = G(x) \cdot M \tag{7}$$

Final Prediction For each vulnerability type, the expert tool with the highest weighted accuracy is selected to make the final prediction:

$$O_{\text{final}} = \sum_{i=1}^{3} G_i(x) \cdot O_i \tag{8}$$

Loss Function Optimization. Our loss function comprises three components for optimizing the model, where O_i means the output predicted with each individual feature and O_{final} means the output of weighted features.

Feature Weight Adjustment Loss L_{feature} focuses on the independent performance of each feature prediction, assessing their contribution through weighted cross-entropy loss:

$$L_{\text{feature}} = \sum_{i \in \{\text{raw,expl,pred}\}} w_i' \cdot L_{\text{cross entropy}}(O_i, Y) \tag{9}$$

Overall Cross-Entropy Loss L_{pred} measures the overall prediction performance for the weighted average prediction of all features:

$$O_{\text{final}} = \sum_{i \in \{\text{raw,expl,pred}\}} w_i' \cdot O_i \tag{10}$$

$$L_{\text{pred}} = L_{\text{cross entropy}}(O_{\text{final}}, Y) \tag{11}$$

Weight Regularization Loss L_{reg} is designed to balance the loss functions to prevent rapid weight updates:

$$L_{\text{reg}} = \gamma(\|w'_{\text{raw}} - w_{\text{raw}}\|^2 + \|w'_{\text{expl}} - w_{\text{expl}}\|^2 + \|w'_{\text{pred}} - w_{\text{pred}}\|^2) \tag{12}$$

The final loss function is a weighted combination of these components, where α denotes the balance coefficient that minimizes the average final loss:

$$L_{\text{total}} = \alpha \cdot (L_{\text{feature}} + L_{\text{reg}}) + (1 - \alpha) \cdot L_{\text{pred}} \tag{13}$$

Weight Update: We use gradient descent based on the gradient of the total loss function to adaptively adjust weights during training:

$$w_i' := w_i' - \eta \cdot \frac{\partial L_{\text{total}}}{\partial w_i'} \tag{14}$$

This optimization process ensures that our Adaptive Mixture-of-Experts dynamically adjusts the importance of

each feature type according to different smart contracts, fully leveraging the strengths of each component to achieve optimal vulnerability detection performance.

IV. Experiments

A. Research Questions

To evaluate our proposed SAEL approach, we conduct experiments to answer the following research questions:

- **RQ1**: How effective does our proposed model SAEL perform compared to state-of-the-art methods?
- **RQ2**: What is the contribution of various key components and different features in the proposed SAEL framework to its overall performance?
- **RQ3**: How do the parameters of SAEL affect the performance of the model?
- **RQ4**: Can SAEL identify smart contract vulnerabilities in a zero-shot manner?

B. Dataset

For the evaluation of our approach in detecting reentrancy vulnerabilities, we employ the recently introduced SmartBugs Wild Dataset [29] as our benchmark. This comprehensive dataset comprises 47,398 distinct Solidity language files, encompassing a total of approximately 203,716 contracts with identified vulnerabilities.

To assess the effectiveness of our method in identifying timestamp dependency vulnerabilities, we make use of the ESC (Ethereum Smart Contracts) Dataset [30]. This dataset is composed of 40,932 Ethereum smart contracts and concentrates on two specific types of vulnerabilities: reentrancy and time dependence. The dataset includes a total of 307,396 functions, out of which nearly 4,833 functions contain the block.timestamp match pattern, which serves as a potential indicator of time dependence vulnerabilities. In our experiments, we specifically focus on the functions that exhibit the block.timestamp match pattern as our dataset.

For our study on integer overflow/underflow and delegatecall vulnerabilities, we have integrated two of the most comprehensive publicly available vulnerability datasets for smart contracts [31], [32].

C. Baselines

In our evaluation, we first select a set of baselines specifically designed for Smart Contract Vulnerability Detection. They can be broadly classified into three categories: rule-based techniques, pre-trained models-based techniques and LLM-based techniques.

Baseline methods, categorized as rule-based techniques, employ predefined heuristics to detect vulnerabilities in smart contracts. This category includes tools such as Manticore [18], Mythril [16], Osiris [17], Oyente [15], Slither [19], Securify [41], and Smartcheck [20].

Pre-trained models-based techniques, rely on pre-trained models like CodeT5 [26], CodeBERT [42], GraphCodeBERT [27] and fine-tuning techniques to identify smart contract vulnerabilities, including Peculiar [24], PSCVFinder [25] and ReVulDL [43].

LLM-based techniques, which rely on LLMs to identify vulnerabilties in smart contract, including GPTScan [40] and iAudit [44].

TABLE I
Training Hyperparameters

	Hyperparameter	Value
Large Language Models Inference	Max input length	2048
	Max output length	512
	Top-p	1
	Temperature	0
	Repetition penalty	1.2
CodeT5/T5 Training	Learning rate	5e-5
	Max input length	512
	Max output length	32
	Beam size	10
	Batch size	32

D. Metrics

To evaluate the performance of our proposed model and other baseline approaches in identifying smart, we employed widely accepted evaluation criteria, namely Precision, Recall, and F1-score. Precision measures the proportion of correctly identified vulnerabilities among all the predicted positive cases. Recall, on the other hand, represents the fraction of correctly detected vulnerabilities out of all the actual vulnerabilities present in the dataset. Lastly, the F1-score provides a balanced measure by calculating the harmonic mean between Precision and Recall, giving equal weight to both metrics.

Why we choose these metrics. In real-world scenarios, secure smart contracts significantly outnumber those with vulnerabilities, resulting in imbalanced datasets. Under such circumstances, accuracy measures may yield misleading results. In contrast, the three metrics can more accurately reflect model performance on imbalanced data.

E. Implementation Details

We leveraged CodeT5 [26] and T5 [28] which followed the initialization of its pre-training work. As shown in Table I, for the training of CodeT5 and T5, we set the max input length to 512, the max output length to 32, the batch size to 32, and the learning rate to 5e-5. We used a beam size of 10 during the training process. We performed a 3:1:1 split for training, validation, and test to evaluate our model. We implemented all training with 1 NVIDIA GeForce RTX H800 GPU with 80GB memory and CUDA 12.2 on PyTorch. It took about 4 hours for smart contract vulnerabilities detection training. For the inference of large language models, we set the max input length to 2048 and the max output length to 512. To ensure the deterministic output, we set the temperature to 0 and top-p to 1. A repetition penalty of 1.2 was applied to avoid repeated generation. The inference was performed on a server equipped with 2 NVIDIA GeForce RTX H800 GPUs, each with 80GB memory.

TABLE II
THE PERFORMANCE OF OUR METHOD COMPARED WITH 12 BASELINES IN TERMS OF PRECISION, RECALL AND F1-SCORE.

Methods	Reentrancy				Timestamp Dependency				Overflow/Underflow				Delegatecall			
	P(%)	R(%)	F1(%)	Rank	P(%)	R(%)	F1(%)	Rank	P(%)	R(%)	F1(%)	Rank	P(%)	R(%)	F1(%)	Rank
Manticore	50.00	50.36	50.18	13	–	–	–	–	–	–	–	–	–	–	–	–
Mythril	50.35	51.80	51.06	12	50.00	41.79	45.53	7	25.30	46.67	32.81	10	42.99	74.19	54.44	5
Osiris	59.06	53.96	56.39	10	52.41	36.71	43.18	8	45.33	75.56	56.57	5	–	–	–	–
Oyente	65.79	53.96	59.29	8	45.17	38.41	41.51	9	60.87	46.67	52.83	6	40.43	30.65	34.86	9
Slither	52.00	65.47	57.96	9	67.26	72.46	69.77	4	32.28	45.56	37.79	8	39.04	91.94	54.81	4
Securify	52.78	54.68	53.71	11	–	–	–	–	–	–	–	–	–	–	–	–
Smartcheck	77.87	68.35	72.80	6	39.24	37.44	38.32	10	31.25	38.89	34.65	9	32.93	43.55	37.50	8
Peculiar	89.13	88.49	88.81	4	–	–	–	–	74.73	75.56	75.14	2	66.67	61.29	63.87	3
ReVulDL	91.49	92.81	92.14	2	88.09	85.75	86.90	3	–	–	–	–	–	–	–	–
PSCVFinder	92.65	90.65	91.64	3	90.64	88.89	89.76	2	65.00	72.22	68.42	3	69.23	72.58	70.87	2
GPTScan	62.22	80.58	70.22	7	57.41	74.88	64.99	6	39.11	77.78	52.04	7	31.43	88.71	46.41	6
iAudit	65.79	89.93	75.99	5	57.07	84.78	68.22	5	42.86	83.33	56.60	4	28.80	85.48	43.09	7
SAEL	**93.62**	**94.96**	**94.29**	**1**	**90.16**	**95.17**	**92.60**	**1**	**79.00**	**87.78**	**83.16**	**1**	**78.46**	**82.26**	**80.31**	**1**

F. Experimental Results

In this section, we present experimental results to answer the research question.

1) RQ1: To evaluate the effectiveness of our proposed SAEL method, we compared it with state-of-the-art baseline methods. The experimental results are presented in Table II.

For reentrancy vulnerability detection, SAEL achieved the best performance in terms of Precision, Recall, and F1-score, reaching 93.62%, 94.96%, and 94.29%, respectively, significantly outperforming all other baseline methods. ReVulDL, a pre-trained model-based method, was the second-best performer with an F1-score of 92.14%. For timestamp dependency vulnerability detection, SAEL also demonstrated superior performance, ranking first in all three evaluation metrics with a Precision of 90.16%, Recall of 95.17%, and F1-score of 92.60%, surpassing all baseline methods. Another pre-trained model-based method PSCVFinder achieved the F1-score of 89.76%, ranking second in this task. This demonstrates the effectiveness of explanations for smart contract vulnerability detection tasks. In the detection of integer overflow/underflow vulnerabilities, SAEL continued to show excellent performance, achieving the highest F1-score of 83.16%. This surpassed the second-best method, Peculiar, which had an F1-score of 75.14%. For delegatecall vulnerability detection, SAEL maintained its leading position with an F1-score of 80.31%, outperforming the next best method, PSCVFinder, which achieved an F1-score of 70.87%.

[RQ1]: SAEL consistently outperformed 12 state-of-the-art baseline methods across all four types of vulnerabilities (reentrancy, timestamp dependency, integer overflow/underflow, and delegatecall).

2) : We explore the influence of various components and features on the performance of smart contract vulnerability detection in SAEL. The framework utilizes three key features: raw smart contract code features (R), explanations generated by LLMs (E), and predictions provided by LLMs (P).

Fig. 6 demonstrates the effectiveness of the REP feature, which is obtained by integrating R, E, and P features through Adaptive Mixture-of-Experts. The REP feature achieves the highest F1-scores for all three vulnerability types: 94.29% for reentrancy, 92.60% for timestamp dependency, and 83.16% for integer overflow/underflow. When R, E, and P features are used individually, they result in lower F1-scores across all vulnerability types. The raw code features (R) consistently contribute the most, followed by explanations (E), with predictions (P) having the least impact. This finding suggests that the incorporation of explanations (E) and prediction results (P) generated by LLMs significantly boosts performance compared to using raw code features (R) alone.

The explanations and prediction results generated by LLMs plays a crucial role in the smart contract vulnerability detection task. Detecting vulnerabilities in smart contracts requires a deep understanding of the semantics and context of the code. LLMs, pre-trained on vast amounts of code and natural language data, possess the ability to comprehend code semantics and context. The explanations generated by LLMs highlight potential vulnerabilities and provide the reasons behind the predictions. These explanations offer a high-level understanding of the behavior of the code and potential security risks, compensating for the limitation of lacking semantic understanding when relying solely on raw code features.

The impact of different modules on the performance of SAEL is analyzed in Fig. 7. The complete SAEL model (Base) achieves the best results across all three vulnerability types, with F1-scores of 94.29% for reentrancy, 92.60% for timestamp dependency, and 83.16% for integer overflow/underflow. When the language model module (w/o LLM) is removed, the F1-scores decrease across all vulnerability types. Here, w/o LLM refers to performing prompt-tuning only on the raw code without utilizing LLM-generated explanations and predictions. Further removing the Adaptive Mixture-of-Experts module (w/o MOE) results in a more significant performance drop.

The w/o MOE condition indicates direct averaging of the prediction results from the three feature types without dynamic weight adjustment.

These findings highlight the positive contributions of both modules to the overall performance across all vulnerability types. The Adaptive Mixture-of-Experts module appears to have a more pronounced impact, especially for reentrancy and timestamp dependency vulnerabilities, where its removal leads to a larger performance drop compared to removing the LLM module.

[RQ2]: The explanations and predictions generated by LLMs significantly enhance vulnerability detection performance. The Adaptive Mixture-of-Experts module further optimizes detection by dynamically adjusting feature weights.

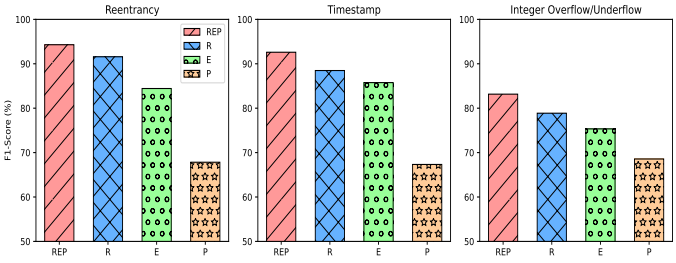

Fig. 6. Analysis of Different Features

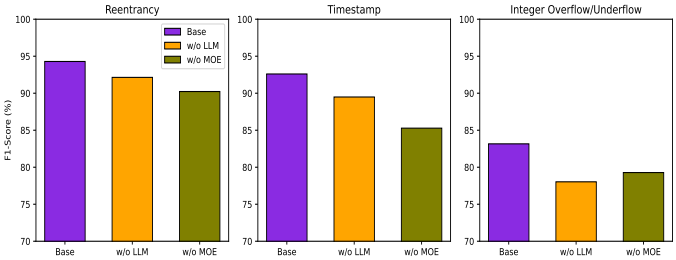

Fig. 7. Analysis of Different Modules

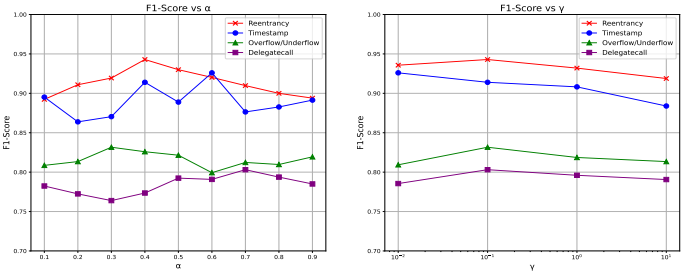

Fig. 8. Performance of SAEL with different parameters.

3) RQ3: Based on the experimental results shown in Fig. 8, our SAEL exhibits parameter sensitivity across four types of vulnerabilities: reentrancy, timestamp dependency, integer overflow/underflow, and delegatecall. As the two key parameters α and γ in Adaptive Mixture-of-Experts vary, the F1-scores for these vulnerability types show different trends.

The parameter α balances the feature weight adjustment loss $L_{feature}$ and the overall cross-entropy loss L_{pred} in the loss function design. For reentrancy and timestamp dependency vulnerabilities, the F1-scores peak when α is around 0.4-0.6, indicating that a moderate balance between the two losses improves detection performance. Integer overflow/underflow vulnerabilities show a relatively stable performance across different α values, while delegatecall vulnerabilities exhibit more fluctuations.

Furthermore, the parameter γ controls the proportion of weight regularization loss Lreg in the total loss to prevent rapid changes in feature weights. When γ is between 10^{-2} and 10^{0}, the detection performance for all vulnerability types is relatively stable, indicating that moderate weight regularization aids convergence and generalization. However, as γ increases beyond 10^{0}, the model performance generally declines across all vulnerability types, with reentrancy and timestamp dependency showing the most significant drops. This suggests that these two vulnerability types require more flexible feature weight adjustments, which are hindered by excessive regularization. Integer overflow/underflow and delegatecall vulnerabilities show less sensitivity to high γ values, maintaining relatively stable performance even at $\gamma = 10^{1}$. This could indicate that the detection of these vulnerabilities relies more on consistent, general patterns that are less affected by strict weight regularization.

[RQ3]: Regarding to the four vulnerabilities, the loss function balance parameter α and weight regularization parameter γ in Adaptive Mixture-of-Experts have a significant impact on model performance and need to be properly adjusted.

4) RQ4: To evaluate the zero-shot performance of SAEL on smart contract vulnerability detection, we randomly selected 400 labeled samples from various datasets: 100 contracts for reentrancy vulnerability detection from the Ethereum Smart Contract Dataset [22], and 300 contracts from the Ethernet Open Dataset [31] (100 each for timestamp dependency, integer overflow/underflow, and delegatecall vulnerability detection). All selected contracts were excluded from the train and test datasets. The ratio of vulnerable to non-vulnerable contracts for each vulnerability type is as follows: reentrancy (32:68), timestamp dependency (43:57), and both integer overflow/underflow and delegatecall (22:78).

We show a simplified case in Fig. 9. The reentrancy vulnerability in this contract differs from traditional ones in the following aspects: the triggering conditions are more hidden, requiring password checks and prize pool condition judgments; exploiting the vulnerability requires cross-contract interactions; complex reward calculation logic may interfere

```
1  function claim_reward(uint uid, bytes32 passcode)
2  public payable{
3  ......
4      require(msg.value >= parameters["price"]);
5      require(is_passcode_correct(uid, passcode));
6
7      uint final_reward = get_reward(uid) + msg.value;
8      if (final_reward > parameters["price_pool"])
9          final_reward = parameters["price_pool"];
10
11     require(msg.sender.call.value(final_reward)());
12 ......
13     parameters["price_pool"] -= final_reward;
14     if (uid + 1 < users.length)
15         users[uid] = users[users.length - 1];
16     users.length -= 1;
17 }
```

The claim_reward function in the smart contract contains a reentrancy vulnerability. The issues lie in the following points:
1.The function starts by requiring a minimum amount of ether to be sent with the transaction (line 4).
2.It then checks the correctness of the passcode (line 5).
3.Next, it calculates the final reward and checks if it exceeds the prize pool (lines 7-9).
4.The vulnerability lies in the external call on line 11 using msg.sender.call.value(final_reward)(). This call can trigger the fallback function of a malicious contract, which can then recursively call claim_reward before the balance is updated, allowing the attacker to drain the contract's funds.

Line of code to focus on:
Line 11: require(msg.sender.call.value(final_reward)());

To mitigate this vulnerability, the contract should use the transfer function instead of call.value(). The transfer function prevents reentrancy by only transferring a fixed amount of gas. Alternatively, the contract can use a state variable to track whether the reward has been claimed and update the state before making any external calls.

Fig. 9. A simplified real-world smart contract with a complex reentrancy vulnerability and its explanation generated by SAEL.

TABLE III
THE RESULTS OF SAEL IN A ZERO-SHOT MANNER.

Vulnerabilities	Precision(%)	Recall(%)	F1(%)
Reentrancy	93.50	90.60	92.00
Timestamp	88.60	90.70	89.60
Overflow/Underflow	85.70	81.80	83.70
Delegatecall	85.00	77.30	81.00

with the ability of detection tools to identify key vulnerability points; the location of contract state updates is uncommon, increasing detection difficulty. Usually reentrancy vulnerabilities are caused by updating the balance status after the transfer, but in this case the bonus pool balance is deducted before deleting the user information. Traditional tools based on rule matching and program analysis including Oyente [15], Securify [41], Smartcheck [20], Slither [19] and Mythril [16] fail to accurately detect such complex vulnerabilities, mainly due to the difficulty of rule matching in covering all variants, the difficulty of program analysis in understanding code semantics, and complex logic interfering with vulnerability localization. As shown in Table III, SAEL demonstrated strong vulnerability detection capabilities in a zero-shot manner.

SAEL addresses the above challenges by leveraging LLMs for in-depth code understanding and combining them with the Adaptive Mixture-of-Experts to dynamically adjust different features. Specifically, LLMs can deeply understand code semantics, accurately grasping vulnerability triggering conditions. Adaptive Mixture-of-Experts enables flexible capture of key features which can adapt to complex vulnerability scenarios. Furthermore, the analysis generated by SAEL provide detailed explanations of vulnerability principles.

[RQ4]: Our findings demonstrate the strong zero-shot capability of SAEL. Through a case study, we showcase the superior ability of SAEL to detect complex vulnerabilities compared to rule-based methods, while also providing the comprehensive explanation.

V. RELATED WORK

In this section, we review four key areas of related work for smart contract vulnerability detection and related applications: rule-based methods, pre-trained model approaches, LLM-based methods, and applications of LLMs in broader software engineering tasks.

A. Rule-based Methods

Various studies have employed traditional program analysis methods to identify particular vulnerabilities in smart contracts. Oyente [15] leverages symbolic execution to uncover four types of vulnerabilities by exploring different execution paths within smart contracts. Mythril [16] and SmartCheck [20] rely on pattern matching techniques to detect vulnerabilities based on a set of predefined rules. Securify [41] utilizes formal verification using logical languages to ensure the security of smart contracts. Osiris [17] and Manticore [18] combine symbolic execution and taint analysis, while Maian [45] uses symbolic analysis and concrete validation. Slither [19] is a static analysis framework that employs data flow and taint analysis.

B. Pre-trained models-based Methods

Several studies have leveraged pre-trained models in various ways to enhance smart contract security. Peculiar [24] focused on improving generalization through pre-training, and ReVulDL [43] utilized a graph-based pre-training model to capture propagation chain relationships. PSCVFinder [25] utilizes prompt-tuning to bridge the gap between pre-training task and smart contract vulnerability detection task.

C. LLM-based Methods

Recent studies [38], [39] have evaluated the performance of LLMs on real-world datasets, revealing that LLMs encounter performance-related challenges due to a high prevalence of false positives. Hu et al. [37] investigated the application prospects of LLMs in smart contract vulnerability detection from new perspectives. Sun et al. [40] introduced GPTScan, the first tool that combines GPT with program analysis for detecting logic vulnerabilities in smart contracts. GPTScan breaks down each logic vulnerability type into scenarios and properties, utilizes GPT to match candidate vulnerabilities, and then confirms them through static analysis. Ma et al. [44] introduced iAudit, a two-stage framework leveraging large language models for detecting vulnerabilities and providing explanations. These works demonstrate the great potential of LLMs in the field of smart contract security.

D. Applications of LLMs in Software Engineering

Large Language Models (LLMs) have been increasingly applied to various software engineering tasks. For code review automation, Lu et al. [46] introduced DeepCRCEval, integrating LLMs to improve evaluation quality and efficiency. Lu et al. [47] proposed LLama-Reviewer, which uses parameter-efficient fine-tuning for effective and resource-efficient code review. In code understanding, Shen et al. [48] designed a dependency-aware framework for method naming and consistency checking, showing LLMs can enhance these tasks through advanced sampling strategies. For issue resolution,

Zan et al. [49] developed SWE-bench-java, a benchmark for evaluating LLMs on GitHub issue resolution, demonstrating LLMs' potential in automated software maintenance. These works reflect the versatility and effectiveness of LLMs in modern software engineering.

VI. Threats to Validity

Internal Validity: Recent research indicates that the precise influence of hyperparameters on the performance of LLMs and Deep Learning models remains unclear [28], [50], [51]. In our work, we have applied the Tree-structured Parzen Estimator (TPE) [52] to enhance our model performance. However, we recognize that alternative configurations could potentially deliver comparable or better results. Consequently, we plan to explore further settings in our subsequent research.

External Validity: The SAEL model requires substantial labeled training data to extract sufficient features, which may restrict its ability to detect new categories of vulnerabilities where traditional approaches may prove more effective. To overcome this limitation, we can reduce the reliance on labeled data by continuing the pretraining task on unsupervised datasets of smart contracts.

VII. Conclusion

In this paper, we propose SAEL, a smart contract vulnerability detection approach based on LLMs. Unlike the prior work, SAEL novelly utilizes explanations generated by general-purposed LLMs as a feature to enhance the performance of smart contract vulnerability detection. Furthermore, we introduce Adaptive Mixture-of-Experts to dynamically adjust the weights of prediction results for LLMs, explanation features, and contract code features. Our approach outperforms state-of-the-art methods. For future work, we aim to decrease our dependency on labeled data by furthering the pretraining process using unsupervised datasets of smart contracts.

VIII. Acknowledgement

This work was supported by 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] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[4] L. Yu, F. Zhang, J. Ma, L. Yang, Y. Yang, and W. Jia, "Who are the money launderers? money laundering detection on blockchain via mutual learning-based graph neural network," in *2023 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2023, pp. 1–8.

[5] Y. Yang, L. Yang, L. Li, X. Ma, L. Yu, and C. Zuo, "Dccgraph: Detecting criminal communities with augmented criminal network construction and graph neural network," in *2023 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2023, pp. 1–8.

[6] J. Zhang, L. Yu, Z. Huang, L. Yang, and F. Zhang, "Topology augmented multi-band and multi-scale filtering for graph anomaly detection," *ACM Transactions on Knowledge Discovery from Data*, 2025.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] L. Yu, Z. Huang, H. Yuan, S. Cheng, L. Yang, F. Zhang, C. Shen, J. Ma, J. Zhang, J. Lu *et al.*, "Smart-llama-dpo: Reinforced large language model for explainable smart contract vulnerability detection," *Proceedings of the ACM on Software Engineering*, vol. 2, no. ISSTA, pp. 182–205, 2025.

[12] H. Yuan, L. Yu, Z. Huang, J. Zhang, J. Lu, S. Cheng, L. Yang, F. Zhang, J. Ma, and C. Zuo, "Mos: Towards effective smart contract vulnerability detection through mixture-of-experts tuning of large language models," *arXiv preprint arXiv:2504.12234*, 2025.

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

[14] P. Hegedűs, "Towards analyzing the complexity landscape of solidity based ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 35–39.

[15] 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.

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

[17] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[18] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[19] 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.

[20] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, 2018, pp. 9–16.

[21] Y. Chen, Z. Sun, Z. Gong, and D. Hao, "Improving smart contract security with contrastive learning-based vulnerability detection," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 940–940.

[22] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.

[23] F. Luo, R. Luo, T. Chen, A. Qiao, Z. He, S. Song, Y. Jiang, and S. Li, "Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 954–954.

[24] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 378–389.

[25] L. Yu, J. Lu, X. Liu, L. Yang, F. Zhang, and J. Ma, "Pscvfinder: A prompt-tuning based framework for smart contract vulnerability detection," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 556–567.

[26] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[27] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*, 2020.

[28] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[29] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1349–1352.

[30] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion," *arXiv preprint arXiv:2106.09282*, 2021.

[31] Z. Liu, P. Qian, J. Yang, L. Liu, X. Xu, Q. He, and X. Zhang, "Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 1237–1251, 2023.

[32] P. Qian, Z. Liu, Y. Yin, and Q. He, "Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 2220–2229.

[33] L. Yu, S. Cheng, Z. Huang, J. Zhang, C. Shen, J. Lu, L. Yang, F. Zhang, and J. Ma, "Sael: Leveraging large language models with adaptive mixture-of-experts for smart contract vulnerability detection," 2025. [Online]. Available: https://zenodo.org/records/16421321

[34] 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.

[35] 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.

[36] 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.

[37] S. Hu, T. Huang, F. İlhan, S. F. Tekin, and L. Liu, "Large language model-powered smart contract vulnerability detection: New perspectives," *arXiv preprint arXiv:2310.01152*, 2023.

[38] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, Y. Wang, X. Lin, T. Chen, and Z. Zheng, "When chatgpt meets smart contract vulnerability detection: How far are we?" *arXiv preprint arXiv:2309.05520*, 2023.

[39] I. David, L. Zhou, K. Qin, D. Song, L. Cavallaro, and A. Gervais, "Do you still need a manual smart contract audit?" *arXiv preprint arXiv:2306.12338*, 2023.

[40] 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.

[41] 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.

[42] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[43] Z. Zhang, Y. Lei, M. Yan, Y. Yu, J. Chen, S. Wang, and X. Mao, "Reentrancy vulnerability detection and localization: A deep learning based two-phase approach," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[44] W. Ma, D. Wu, Y. Sun, T. Wang, S. Liu, J. Zhang, Y. Xue, and Y. Liu, "Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications," *arXiv preprint arXiv:2403.16073*, 2024.

[45] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, "Towards safer smart contracts: A sequence learning approach to detecting security threats," *arXiv preprint arXiv:1811.06632*, 2018.

[46] J. Lu, X. Li, Z. Hua, L. Yu, S. Cheng, L. Yang, F. Zhang, and C. Zuo, "Deepcrceval: Revisiting the evaluation of code review comment generation," in *International Conference on Fundamental Approaches to Software Engineering*. Springer Nature Switzerland Cham, 2025, pp. 43–64.

[47] J. Lu, L. Yu, X. Li, L. Yang, and C. Zuo, "Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 647–658.

[48] C. Shen, J. Zhu, L. Yu, L. Yang, and C. Zuo, "Dependency-aware method naming framework with generative adversarial sampling," in *2024 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2024, pp. 1–8.

[49] D. Zan, Z. Huang, A. Yu, S. Lin, Y. Shi, W. Liu, D. Chen, Z. Qi, H. Yu, L. Yu *et al.*, "Swe-bench-java: A github issue resolving benchmark for java," *arXiv preprint arXiv:2408.14354*, 2024.

[50] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 2623–2631.

[51] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng, "Exploring the potential of chatgpt in automated code refinement: An empirical study," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[52] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," *Advances in neural information processing systems*, vol. 24, 2011.