From Cryptic to Clear—Training on LLM Explanations to Detect Smart Contract Vulnerabilities

YIZHOU CHEN, Key Lab of HCST (PKU), MOE; School of Computer Science, Peking University, Beijing, China
ZEYU SUN, National Key Laboratory of Space Integrated Information System, Institute of Software, Chinese Academy of Sciences, Beijing, China
GUOQING WANG and QINGYUAN LIANG, Key Lab of HCST (PKU), MOE; School of Computer Science, Peking University, Beijing, China
XIAO YU, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China
DAN HAO, School of Electronic and Computer Engineering, Peking University, Shenzhen, China

Smart contracts have revolutionized the way transactions are executed, offering decentralized and immutable frameworks. The immutability of smart contracts poses significant risks when vulnerabilities exist in their code, leading to financial losses. Despite advancements in using deep learning for smart contract vulnerability detection (SCVD), existing methods struggle with the complex logic and intricate semantics embedded within smart contract code. Large Language Models (LLMs) have shown promise in providing deeper insights into smart contract logic. However, LLMs, such as GPT follow a decoder-only architecture and are trained in an unsupervised manner rather than learning specific labels. In the SCVD task, these LLMs have difficulty in capturing information related to vulnerabilities, leading to very low accuracy. Therefore, we propose CodeXplain, a novel SCVD approach that leverages the deep insights into code from LLM and the supervised learning capabilities of deep learning models to set the latest advance and performance. In particular, we deeply analyze 14 types of dangerous and common smart contract vulnerabilities. Based on the rationale of these vulnerabilities, nine perspective prompts are introduced to guide LLMs in generating code explanations that contribute to SCVD. Then, we propose a CodeT5-based semantic fusion module integrating smart contract code and code explanations. Finally, the performance of SCVD is improved by performing supervised learning on trusted labels. Experimental results on 3,544 real-world smart contracts demonstrate that CodeXplain outperforms 16 state-of-the-art SCVD methods, achieving an F1-score of 94.12% and an accuracy of 93.88%, surpassing all baselines.

CCS Concepts: • **Security and privacy** → *Software security engineering*; • **Computing methodologies** → *Knowledge representation and reasoning*;

This work was supported by the National Natural Science Foundation of China under Grant Nos. 62372005 and 62402482.
Authors' Contact Information: Yizhou Chen, Key Lab of HCST (PKU), MOE; School of Computer Science, Peking University, Beijing, China; e-mail: yizhouchen@stu.pku.edu.cn; Zeyu Sun, National Key Laboratory of Space Integrated Information System, Institute of Software, Chinese Academy of Sciences, Beijing, China; e-mail: zeyu.zys@gmail.com; Guoqing Wang, Key Lab of HCST (PKU), MOE; School of Computer Science, Peking University, Beijing, China; e-mail: guoqingwang@stu.pku.edu.cn; Qingyuan Liang, Key Lab of HCST (PKU), MOE; School of Computer Science, Peking University, Beijing, China; e-mail: liangqy@stu.pku.edu.cn; Xiao Yu, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China; e-mail: xiao.yu@zju.edu.cn; Dan Hao (corresponding author), School of Electronic and Computer Engineering, Peking University, Shenzhen, China; e-mail: haodan@pku.edu.cn.

ACM 1557-7392/2026/5-ART146
https://doi.org/10.1145/3765753

Additional Key Words and Phrases: Supervised Learning, Deep Learning, Smart Contract, Vulnerability Detection, LLM

ACM Reference format:
Yizhou Chen, Zeyu Sun, Guoqing Wang, Qingyuan Liang, Xiao Yu, and Dan Hao. 2026. From Cryptic to Clear—Training on LLM Explanations to Detect Smart Contract Vulnerabilities. *ACM Trans. Softw. Eng. Methodol.* 35, 6, Article 146 (May 2026), 24 pages.
https://doi.org/10.1145/3765753

1 Introduction

Blockchain is essentially an emerging software system that uses smart contracts and computer networks to maintain transaction data. With the emergence and popularity of blockchain and smart contracts, they bind hundreds of millions of virtual assets [7, 37, 38, 40]. As an essential component of blockchain technology, smart contracts offer a decentralized and immutable way to execute agreements, reducing reliance on intermediaries and enhancing efficiency [2]. Despite the transformative potential of smart contracts, their increasing deployment has also exposed significant vulnerabilities [52]. Smart contracts, by design, are immutable once deployed, which means any errors or vulnerabilities in the code are permanently enshrined on the blockchain. Unlike traditional software, where bugs can be patched or updated post-deployment, smart contracts offer no such flexibility [3, 4]. This immutability becomes a critical weakness when the code contains vulnerabilities that can be exploited.

Over the past few years, there have been numerous high-profile incidents where vulnerabilities were exploited, leading to significant financial losses. One of the most infamous cases is the DAO hack in 2016, where an attacker exploited a reentrancy vulnerability to siphon off $60 million worth of Ether [19, 30]. More recently, the Decentralized Finance sector has become a prime target for attackers, with vulnerabilities in smart contracts leading to losses in the billions [6].

Several researchers have attempted to mitigate the threat of **smart contract vulnerabilities (SCVs)** through various methods [24, 25, 34, 35, 47, 51, 57]. These approaches typically construct various types of semantic or structural graphs—such as control flow graphs, dataflow graphs, and abstract syntax trees—to model the structure and semantics of smart contracts. While these graph-based methods offer a structured representation of code, they inherently rely on the implicit encoding of program semantics. The model must learn complex patterns from these abstract structures, which makes it difficult to directly capture high-level code logic and contextual dependencies. This limitation adversely affects the performance of SCVD models.

Recently, **large language models (LLMs)** such as GPT-4 have emerged as promising tools for the understanding and auditing of smart contracts. These LLMs can generate human-readable text that can be leveraged to provide deeper insights into the code's logic and potential vulnerabilities [12, 20, 26, 42]. However, despite their advances, the application of LLMs to SCVD is still in its infancy, and current research indicates that they fall short of being reliable tools for this task. This is directly related to the architecture and training approach of GPT, which is a typical decoder model and is trained in an unsupervised manner, where the training goal is to predict the next word or token rather than learning a specific label. In the task of SCVD, the model needs to predict the presence of vulnerabilities based on specific patterns in the code, which requires supervised learning combined with deterministic labels. The GPT model, however, does not have explicit vulnerability-related labeled information input during training and is difficult to identify vulnerabilities. Overall, traditional deep learning methods struggle to understand smart contract code in depth, thus limiting their detection performance. LLMs have the ability to understand smart

contract code in depth, but they are not easily supervised fine-tuned for task-specific objectives. Besides, the recent studies have shown that the performance of existing LLMs in detecting SCVs is not
satisfactory [26, 41].

To address the limitations of existing methods, we propose CodeXplain, a code explanation-enhanced vulnerability detection approach based on LLMs. The core of CodeXplain lies in leveraging LLMs to generate detailed explanations that clarify the complex logic and semantics of smart contract code. Subsequently, combining the smart contract code and LLM's deep understanding of the code, a supervised learning approach is used to fuse this information and improve the performance of SCVD. To ensure that these explanations are both targeted and exhaustive, we design nine fine-grained prompts that guide the LLM to generate explanations from multiple perspectives of the smart contract code. These prompts are based on the 14 types of common and severe vulnerabilities. Each prompt is carefully tailored to address and explore distinct aspects of the code, ensuring that the generated explanations provide comprehensive coverage of potential vulnerabilities. This strategy allows for a deep and multi-faceted understanding of the smart contract's logic and structure, including: *basic functionality interpretation*, *step-by-step analysis*, *logic and flow interpretation*, *state management analysis*, *event and function interaction*, *error handling and exceptions*, *Gas Efficiency Examination*, *contract interaction analysis*, and *ownership and access control*. By covering these diverse aspects, CodeXplain ensures that vulnerabilities arising from various facets of the code are thoroughly considered. Finally, we design a semantic fusion model based on CodeT5, which effectively merges the insights from the LLM-generated explanations with the source code. CodeT5 is particularly well suited for this task, as it excels in both understanding the semantics of code and processing natural language inputs. Using supervised learning techniques, we train the semantic fusion model to enhance SCVD performance.

To evaluate the performance of the proposed CodeXplain, we conduct experiments on 3,544 real-world smart contract instances and compare it with 16 cutting-edge SCVD methods, including popular code pre-training models, LLMs, and the state-of-the-art SCVD methods. According to the experimental results, CodeXplain achieves an F1-score of 94.12% and an accuracy of 93.88% and outperforms all baselines. First, CodeXplain outperforms existing pre-trained models including CodeBERT [14], GraphCodeBERT [16], UnixCoder [15], and CodeT5 [45]. It improves by 8.44%, and 7.84% in the F1-score and accuracy compared to the best-performing UnixCoder. Second, CodeXplain is superior to LLM baselines, including GPT-3.5, GPT-4 [1], DeepSeek-Coder-V2 [56, 57], CodeLlama-13b [39], CodeLlama-34b [39], and iAudit [26]. In this comparison, CodeXplain improved by 4.03–37.84% and 3.80–80.36% in the F1-score and accuracy compared to these LLMs. Finally, CodeXplain unsurprisingly outperforms all SCVD methods, which achieves 8.15% and 8.15% absolute improvement in the F1-score and accuracy over the best SCVD method, Clear [8].

In summary, the main contributions of this article are as follows:

- We introduce a new perspective for SCVD, which differs from traditional graph-based approaches. Instead of implicitly learning semantics from structural representations, we leverage LLM-generated explanations to explicitly convey the logical intent of smart contracts.
- We identify and analyze 14 critical SCVs and introduce a novel framework with nine tailored perspectives for auditing smart contracts using LLMs. This targeted approach enhances the relevance and informativeness of vulnerability explanations. To our knowledge, this represents the first systematic method specifically designed for SCVD-oriented explanation aspects.
- We are the first to propose a hybrid SCVD framework, CodeXplain, that combines the strengths of LLM-generated code explanations with the supervised learning ability to improve the

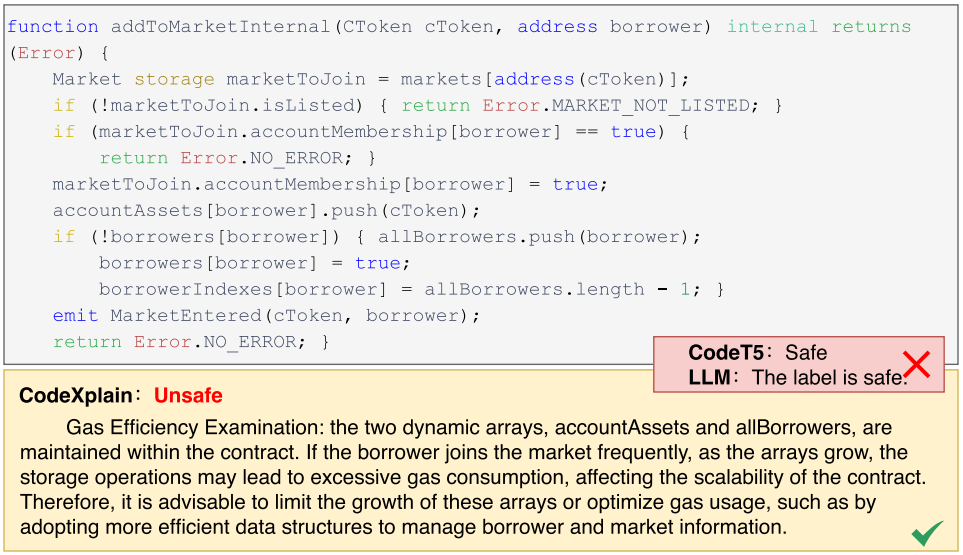

Fig. 1. An example of detection results of different methods.

performance of SCVD. Quantitative experimental results show that CodeXplain outperforms all baselines and sets the optimum performance.

— The source code of CodeXplain is publicly available for further research and experimentation. All source code is publicly available at https://github.com/chenpp1881/CodeXplain.

2 Preliminary

2.1 Motivation

Pre-trained models are deep learning models trained on large-scale datasets to capture general features [18, 36]. Pre-trained often use unsupervised or self-supervised learning methods. They can be fine-tuned with labeled data to perform specific tasks, thus enabling them to learn task-specific features [23]. LLMs represent an advanced category of pre-trained models specifically designed to understand and generate text at scale. LLMs are built upon transformer architectures and trained on extensive text corpora to capture a broad range of language patterns, nuances, and contextual relationships. These models are designed to perform a broad range of natural language processing tasks, including text generation, question-answering, and translation [1, 17, 39, 56].

While both pre-trained models and LLMs share foundational similarities, particularly in their initial training on large datasets, they differ in their applications and adaptability. A key distinction lies in supervised learning ability to utilize labels. Pre-trained models can be further fine-tuned with labeled data, allowing them to specialize in specific tasks [23]. In contrast, LLMs are typically not fine-tuned on labeled data but instead leverage their pre-existing knowledge to perform a wide variety of tasks. Figure 1 shows an example where neither traditional CodeT5 or LLM can correctly determine possible problems with the code, but our CodeXplain can reveal these problems through multi-perspective code explanations.

In the task of SCVD, the absence of labeled data for LLMs often results in suboptimal performance. In contrast, traditional pre-trained models can be fine-tuned with labeled data, but they often lack the deep understanding of code that LLMs possess. Given these limitations, we propose a combined approach that leverages the strengths of both pre-trained models and LLMs. By integrating the supervised learning capability of pre-trained models with the deep code comprehension offered by

LLMs, this method can enhance the effectiveness of SCVD. To the best of our knowledge, we are the first research work in the SCVD field to make such an attempt.

2.2 Problem Definition

Given a smart contract code snippet C, a large language model M_{LLM}, and a deep learning model M_{DL}, our objective is to accurately predict whether the code snippet C contains SCVs by leveraging the interpretative capabilities of M_{LLM} and the predictive power of M_{DL}. Let X_C denote the input space of code snippets, and X_I represents the space of fine-grained code explanations generated by M_{LLM}. The set of possible labels $Y = \{0, 1\}$ is defined, where 0 indicates a non-vulnerable code and 1 indicates a vulnerable code. The function $f : X_C \times X_I \rightarrow Y$ represents the underlying vulnerability detection function that maps a pair consisting of a code snippet and its explanation to a label in Y. The prediction function learned by the deep learning model M_{DL}, denoted as $h : X_C \times X_I \rightarrow Y$, aims to approximate f as accurately as possible. To achieve this, we employ a loss function ℓ and train the model M_{DL} by minimizing the expected loss over a distribution D of code snippets, their corresponding explanations, and labels. Mathematically, our goal is to minimize the expected loss defined as:

$$\min_{h} \mathbb{E}_{(C,I_C,y)\sim D}[\ell(y, h(C, I_C))],$$

where $I_C = M_{LLM}(C)$ is the explanation of the code snippet C generated by the large language model M_{LLM}.

2.3 Challenge

Although combining pre-trained models and LLMs for SCVD is an intuitive and promising method, several challenges arise in effectively training and coordinating these models to detect SCVs. The primary challenges encountered in this process include:

— *How to obtain code explanations that help in vulnerability detection?* A fundamental challenge in this work is generating code explanations that are specifically tailored to aid in vulnerability detection. The effectiveness of these explanations hinges not merely on their level of detail, but on their ability to highlight aspects of the code that are most relevant to identifying SCVs.

— *How to make effective vulnerability detection?* Even with accurate code explanations and source code, making effective vulnerability judgments remains a complex challenge. Code explanations are often presented in natural language. SCVD often requires reasoning based on both code and explanations, rather than dealing with them separately. Relying on either one alone may not result in an accurate judgment. And co-inference requires the model to have the ability to understand both code and natural language. Additionally, it is imperative for the model to align the information in the natural language description with the specific logic of the code in order to accurately identify vulnerabilities and avoid erroneous judgments.

3 Approach

To address these challenges, we propose a novel approach called CodeXplain. The workflow of this method is illustrated in Figure 2. In the first module, we design a series of prompts from multiple perspectives to generate fine-grained explanations of the smart contract code. These prompts guide the explanation process, ensuring that all aspects of the code, including functionality, logic, and security features, are thoroughly examined. The second module is the semantic fusion module, which integrates the smart contract code and these explanations. This module leverages CodeT5 as the pedestal model to embed both the source code and the explanations generated in the first module. By using attention mechanisms, the module fuses these two representations, enabling a comprehensive

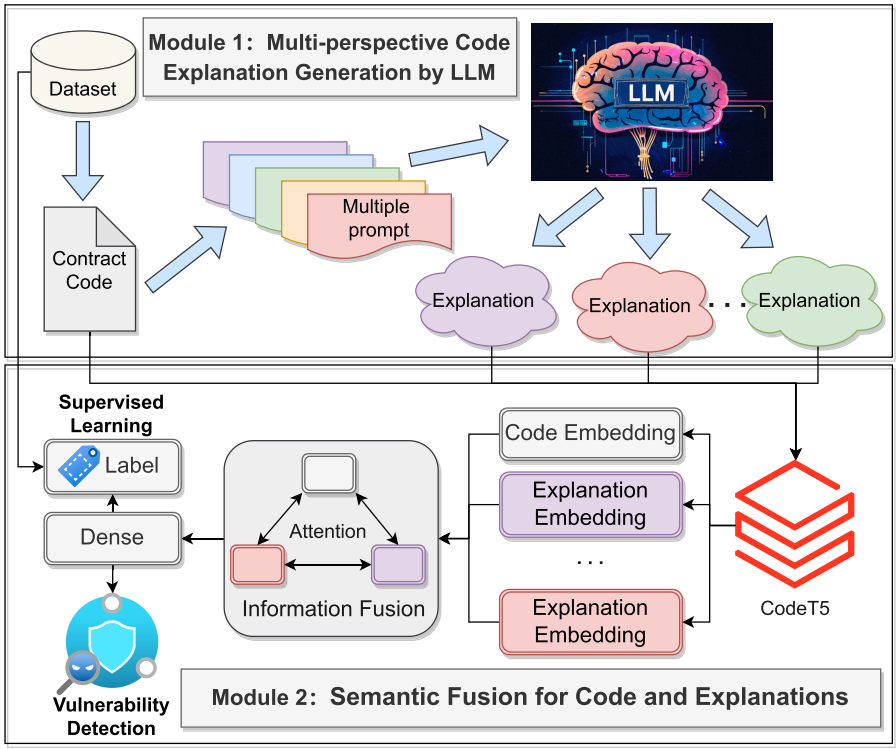

Fig. 2. The overview of CodeXplain.

understanding of both the structural and contextual aspects of the code. Finally, we utilize reliable labels of smart contracts for supervised learning, further enhancing the performance of SCVD.

3.1 Multi-Perspective Code Explanation Generation by LLM

We present the design of the first module of CodeXplain, which leverages LLMs to generate fine-grained, context-specific explanations of smart contract code.

Based on previous work [11, 31, 33], we conduct an in-depth analysis of 14 types of common and dangerous vulnerabilities found in smart contracts, identifying the specific perspectives of the code that are most likely to harbor vulnerabilities. The relationship between these key perspectives and vulnerabilities is shown in Table 1. To effectively capture these security-critical aspects, we design a structured code explanation approach that guides LLMs to focus on nine vulnerability-prone aspects rather than generating arbitrary explanations. This targeted design ensures that the explanations align with real-world security concerns, making them more informative and actionable for vulnerability detection. For example, reentrancy vulnerabilities often stem from improper state management and faulty logic flow, particularly when state variables are updated incorrectly before or after external calls. As shown in Figure 3, Using prompts for "Logic and Flow Interpretation" and "State Management Analysis" allows for a detailed examination of how the contract manages state changes during execution, making it easier to detect potential reentrancy vulnerabilities. Additionally, explaining the basic functions of the contract through the "Basic Functionality Interpretation" prompt can highlight externally invoked functions, which could be entry points for reentrancy attacks. By recognizing these critical functions early, the likelihood of

Table 1. The Relationship between the Key Perspectives and Vulnerabilities

No.	Vulnerability	Perspectives
1	Reentrancy Problem	State Management Analysis, Logic and Flow Interpretation, Basic Functionality Interpretation
2	Transaction Ordering Dependency	Step-by-Step Analysis, Logic and Flow Interpretation
3	Block Timestamp Dependency	Logic and Flow Interpretation, Basic Functionality Interpretation
4	Exception Handling Problem	Error Handling and Exceptions
5	Integer Overflow/Underflow	Step-by-Step Analysis
6	Unchecked and Failed Send	Error Handling and Exceptions
7	Destroyable	Ownership and Access Control, Basic Functionality Interpretation, Event and Function Interaction
8	Suicidal Contract	Ownership and Access Control, Event and Function Interaction, Basic Functionality Interpretation
9	No Restricted Write	Ownership and Access Control, State Management Analysis, Basic Functionality Interpretation
10	No Restricted Transfer	Ownership and Access Control, Basic Functionality Interpretation
11	Non-Validated Arguments	Step-by-Step Analysis, Contract Interaction Analysis
12	Greedy Contracts	Logic and Flow Interpretation, Basic Functionality Interpretation, Event and Function Interaction, Contract Interaction Analysis
13	Prodigal Contracts	Ownership and Access Control, State Management Analysis, Basic Functionality Interpretation
14	Gas Costly Patterns	Gas Efficiency Examination

Logic and Flow Interpretation

Ether Deposit (via credit function):
A user sends Ether to the contract via the credit function.
The contract updates the deposit mapping by adding the Ether amount (msg.value) to the sender's current balance (deposit[msg.sender]).

Check Balance (via getCreditedAmount function):
The user can call getCreditedAmount to view their current deposit balance.
This function returns the Ether balance associated with the sender's address (msg.sender).

Ether Withdrawal (via withdraw function):
The user can request a withdrawal by calling the withdraw function with a specific amount.
If the user has enough balance in the deposit mapping, the contract sends the requested Ether amount to the user using msg.sender.call.value(amount)(). However, using .call without checking for success is dangerous and can expose the contract to reentrancy attacks.
After the Ether is sent, the contract updates the deposit mapping by subtracting the withdrawn amount from the user's balance.

```
contract DAO {   mapping(address => uint256) public deposit;

    function credit(address to) payable {
        deposit[msg.sender] += msg.value;
    }
    function getCreditedAmount(address)
        returns (uint) {
        return deposit[msg.sender];
    }
    function withdraw(uint amount) {
        if (deposit[msg.sender] >= amount) {
        msg.sender.call.value(amount)();
        deposit[msg.sender]-= amount; }
    }
```

Smart Contract

State Management Analysis

State Variable:
...
Function credit(address to) payable :
...
Function getCreditedAmount(address) returns (uint) :
...
Function withdraw(uint amount):
Accessing State:
Before any action is taken, the contract first checks the current state of the sender's balance by reading the deposit mapping (deposit[msg.sender] >= amount).
This ensures that the sender has enough funds in their deposit to withdraw the requested amount of Ether.
Modifying State:
If the check is successful, meaning the user has enough funds, the contract then modifies the state by reducing the user's balance by the withdrawn amount.
This subtracts the amount from the sender's balance stored in the deposit mapping.
Storage Changes:
After the state has been modified, the new balance (after the withdrawal) is stored in the blockchain, reflecting the updated balance for the sender.
State Management Analysis:
Inconsistent State in withdraw: the state variable deposit[msg.sender] is updated after sending Ether to msg.sender. This could result in inconsistent state if an external contract maliciously reenters the withdraw function before the balance is reduced.

Fig. 3. An example of code explanation and vulnerable code.

identifying reentrancy vulnerabilities is enhanced. Below, we outline specific prompts, each with a specific focus and rationale:

— *Basic Functionality Interpretation*:
Prompt: "*Explain the purpose and functionality of the following smart contract code. Highlight the key components and their roles.*"
Rationale: This prompt is designed to generate an overview of the smart contract's main functionalities, helping to establish a foundational understanding of the code's purpose.
— *Step-by-Step Analysis*:
Prompt: "*Break down the following smart contract code into smaller components. Provide a step-by-step explanation of how each part contributes to the overall operation.*"
Rationale: This prompt aims to dissect the code into multiple components, offering a granular view of how each component functions and contributes to the contract's overall operation.
— *Logic and Flow Interpretation*:
Prompt: "*Interpret the logical flow of the following smart contract code. Provide a detailed explanation of how the code executes from start to finish.*"
Rationale: This prompt guides the model to map out the execution sequence, providing insights into how the contract's logic unfolds and where potential risks may lie.
— *State Management Analysis*:
Prompt: "*Describe how state variables are managed in the following smart contract code. Explain how data is stored, modified, and accessed throughout the contract.*"
Rationale: This prompt focuses on how state variables are handled within the code, ensuring that the interpretation highlights any potential issues related to data storage, modification, and access.
— *Event and Function Interaction*:
Prompt: "*Explain the interaction between events and functions in the following smart contract code. Describe how events are emitted and how functions trigger these events.*"
Rationale: This prompt is designed to explore the interaction between events and functions. Understanding these interactions is essential for identifying potential security vulnerabilities in how events are managed and functions are executed.
— *Error Handling and Exceptions*:
Prompt: "*Analyze how error handling is implemented in the following smart contract code. Explain the mechanisms used to catch and manage exceptions or errors.*"
Rationale: This prompt guides the model to examine how errors and exceptions are managed within the contract, identifying potential weaknesses in the code's ability to handle unexpected situations.
— *Contract Interaction Analysis*:
Prompt: "*Explain how this smart contract interacts with other contracts or external entities. Detail how calls are made, and responses are handled within the code.*"
Rationale: Smart contracts often interact with other contracts or external entities, which can introduce security risks. This prompt focuses on analyzing these interactions, ensuring that the interpretation captures any vulnerabilities that might arise from external calls and responses.
— *Ownership and Access Control*:
Prompt: "*Interpret the ownership and access control mechanisms in the following smart contract code. Describe how permissions are enforced and how control is managed.*"
Rationale: Ownership and access control are fundamental to the security of a smart contract. This prompt directs the model to examine how permissions are enforced and control is managed, providing insights into potential vulnerabilities related to access and ownership.

— *Gas Efficiency Examination*:
Prompt: "*Evaluate the gas efficiency of the following smart contract code. Explain how each operation impacts gas usage and suggest any optimizations.*"
Rationale: Understanding gas usage can reveal inefficient code that could be exploited or lead to denial-of-service attacks.

By utilizing these prompts, the LLM is able to generate multiple perspectives on the code, offering explanations that emphasize the functionality of specific components. This approach ensures that the generated interpretations align with common vulnerability patterns observed in smart contracts. Therefore, these explanations can assist in obtaining a more comprehensive understanding of potential vulnerabilities, ultimately aiding in the identification of SCVs. Formally, let C represent the smart contract code and let $P = \{P_1, P_2, \ldots, P_n\}$ denote the set of prompts designed to guide the LLM in generating specific code interpretations, where P_i represents the ith prompt. The LLM function generates an interpretation I_i for each prompt P_i applied to the code C, expressed as:

$$I_i = LLM(C, P_i), \quad i = 1, 2, \ldots, n. \tag{1}$$

The complete set of generated explanations is given by $I = \{I_1, I_2, \ldots, I_n\}$. Finally, the input $S = \{C\} \cup I$ to the next module consists of both the original smart contract code and the generated explanations. This set S serves as the input to the subsequent stage, where it can be utilized for further vulnerability detection.

3.2 Semantic Fusion for Code and Explanations

We propose a semantic fusion module that integrates smart contract source code with its corresponding natural language explanations, leveraging CodeT5 as the foundational model. CodeT5 inherits the T5 text-to-text architecture, which is inherently designed for natural language understanding, making it well suited for processing textual explanations generated by LLMs. Furthermore, CodeT5 is pre-trained on a diverse set of tasks involving both code and natural language, such as code summarization, code generation, and code comment generation. These tasks require mapping between natural language and code representations, enabling CodeT5 to effectively encode smart contract code and its explanations, thereby ensuring high-quality feature representations. Subsequently, we use the attention mechanism to fuse code explanations and smart contract source code. Since code and natural language are inherently different modalities, a simple concatenation of these representations would treat them as independent features, failing to establish meaningful interactions. To address this, we leverage a cross-attention-based fusion mechanism, which enables the model to dynamically align natural language explanations with the corresponding code semantics. For example, a function explanation in natural language may be related to multiple lines in the code. The attention mechanism can efficiently capture these relationships and assign appropriate weights to each explanation or code fragment. The workflow of the module is shown in Figure 4.

Formally, we represent CodeT5 as f_{CodeT5}, which is used to encode both the smart contract source code C and each code explanation I_i, generated from prompts. Here, the code and each explanations are encoded independently, ensuring that each input is processed separately while sharing the same CodeT5 parameters ensuring consistent latent space projection. The encoding functions for the code and explanations are defined as follows:

$$E_C = f_{\text{CodeT5}}(C), \tag{2}$$

$$E_{I_i} = f_{\text{CodeT5}}(I_i) \quad \text{for } i = 1, 2, \ldots, 9. \tag{3}$$

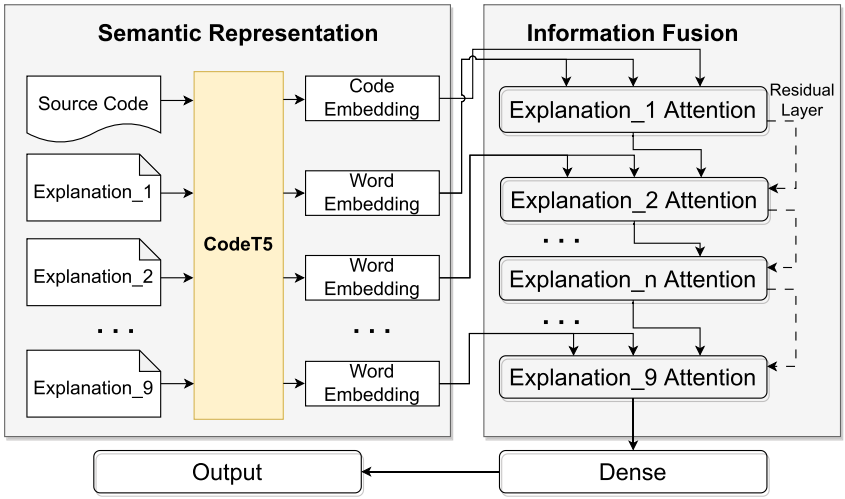

Fig. 4. The workflow of the semantic fusion module.

Let $A(Q, K, V)$ represent the attention mechanism, where Q denotes the query, K the key, and V the value. This process is represented as follows:

$$A(Q, K, V) = \text{Softmax}\left(\frac{Q_s K_s^T}{\sqrt{d_k}}\right) V_s, \tag{4}$$

where $head_s$ is the output of the sth attention head.

Subsequently, a fully connected neural network is built which contains two transformation layers and an activation function (ReLU) according to the following equation:

$$M = \text{Concat}(head_1, head_2, \ldots, head_S)W, \tag{5}$$

$$M' = \max(0, M_C \bullet W_1 + \boldsymbol{b}_1)W_2 + \boldsymbol{b}_2, \tag{6}$$

where W, W_1, and W_2 are weight matrices, $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are bias vectors, and M' is the output of multi-head attention. The multi-head attention is repeated N times and outputs a feature matrix.

In our approach, the code explanations E_{I_i} are used as both the keys and values, while the source code E_C serves as the queries. To mitigate the impact of the varying order of the nine explanations on training, we randomly shuffle the order of the explanations before every training step. Then, the attention mechanism is applied iteratively for each code explanation aspect:

$$E_C^{i+1} = A(E_C^i, E_{I_i}, E_{I_i}) + E_C^i \quad \text{for } i = 1, 2, \ldots, 9. \tag{7}$$

After applying the attention mechanism across multiple iterations, the final output is an eigen-matrix M'. Then, we apply an average pooling operation, denoted as $\text{AvgPool}(M')$, to generate a lower-dimensional representation:

$$M_{\text{pooled}} = \text{AvgPooling}(M'). \tag{8}$$

Next, the M_{pooled} is fed into a fully connected layer with a softmax activation function to output the predicted probability $\hat{y}$ according to:

$$\hat{y} = \text{Softmax}(M_{\text{pooled}} \bullet W + b), \tag{9}$$

where W is the weight matrix and b is the bias vector.

For training, the cross-entropy loss, $\mathcal{L}$, is used:

$$\mathcal{L} = -y \log(\hat{y}), \tag{10}$$

where y represents the ground truth labels.

4 Experimental Settings

In this article, we focus on multiple perspectives that may trigger SCVs and present a novel method for SCVD called CodeXplain. It combines the code understanding capabilities of LLMs with the supervised fine-tuning strengths of CodeT5. This combination leverages the code understanding power of LLMs and the task-specific accuracy of pre-trained models, creating a more effective framework for SCVD. Subsequently, CodeXplain is employed for the detection of SCVs. Therefore, in the subsequent sections of this article, we conduct a comprehensive evaluation of CodeXplain from two key perspectives: (1) its overall performance in SCVD tasks; (2) the specific contribution of LLMs' code understanding capabilities to the SCVD; (3) whether CodeT5 is the most suitable pedestal model in CodeXplain.

To guide our evaluation, we formulate the following **research questions (RQs)**:

RQ1: How does CodeXplain perform in comparison to existing state-of-the-art methods?
RQ2: What is the contribution of the code explanations of LLMs to the SCVD?
RQ3: How do different pedestal pre-training models affect the performance of CodeXplain?

4.1 Datasets

Our study utilizes a publicly available SCVD dataset, which was collected and published by Ma et al. [26]. This dataset is well balanced, comprising 1,734 positive samples representing vulnerable contracts and 1,810 negative samples representing non-vulnerable contracts. The positive samples are meticulously curated from 263 reputable audit reports, ensuring a high degree of reliability for vulnerability detection tasks. In total, the dataset contains over 170k lines of Solidity code, covering contracts written in multiple Solidity versions.

Following the original experimental setup, we adopt the same dataset partitioning strategy, with 80% used for training and the remaining 20% for testing.

Our study utilizes a publicly available SCVD dataset, which was collected and published by Ma et al. [26]. This dataset is well balanced, comprising 1,734 positive samples representing vulnerability contracts and 1,810 negative samples representing non-vulnerability contracts. The positive samples are meticulously curated from 263 reputable audit reports, ensuring a high degree of reliability for vulnerability detection tasks. In alignment with the original setup by previous work, we maintain the same dataset partitioning strategy. We divide 80% of the dataset into the training set and the remaining 20% as the test set.

4.2 Baselines

In this study, we use 16 vulnerability detection methods as the baselines and evaluate the performance of our CodeXplain. These baselines represent a diverse range of approaches that reflect the current state-of-the-art in the SCVD, including LLM-based methods, pre-trained model-based methods, and traditional SCVD approaches (deep learning-based methods and static analysis tools).

The first category of baselines consists of pre-trained models such as CodeBERT [14], GraphCodeBERT [16], UnixCoder [15], and CodeT5 [45]. These models are specifically designed for tasks involving source code and benefit from training on both code and natural language datasets. For these baselines, we fine-tune these pre-trained models on the smart contract dataset and evaluate their vulnerability detection performance.

The second category of baselines is based on LLMs, including GPT-3.5, GPT-4 [1], CodeLlama-13b [39], CodeLlama-34b [39], DeepSeek-Coder-V2 [56], and iAudit [26]. LLMs are capable of capturing both the syntax and semantics of code due to their training in extensive text corpora. In this study, we employ simple prompts to generate predictions about contract vulnerabilities using a zero-shot approach.

Lastly, we include six state-of-the-art SCVD methods as baselines. Among them, Slither [13] and Mythril [29] employ static analysis techniques to analyze smart contract code. These tools have been recognized as the most effective static analysis methods for SCVD [8, 35]. In addition, we incorporate several deep learning-based SCVD methods, including DMT [35], EGFL [10], VDCEP [9], and Clear [8]. These approaches leverage techniques such as graph neural networks and contrastive learning to capture patterns associated with vulnerabilities. Unlike LLM-based or pre-trained models, these methods are trained from scratch on smart contract datasets, without relying on large-scale pre-training.

4.3 Evaluation Metrics

Vulnerability detection is a binary classification task. Therefore, there are four possible outputs:

— *True Positive (TP)*: The number of the vulnerable contracts that are correctly identified.
— *False Positive (FP)*: The number of the non-vulnerable contracts that are wrongly predicted to be vulnerable.
— *False Negative (FN)*: The number of vulnerable contracts that are wrongly predicted as the non-vulnerable contracts.
— *True Negative (TN)*: The number of the non-vulnerable contracts that are correctly identified.

Following existing software-engineering research [49, 53–55], we measure the performance of a vulnerability detection method via precision, recall, F1-score, and accuracy metrics. Therefore, we adopt these four metrics in evaluation:

$$\text{Precision} = \frac{TP}{TP + FP},$$

$$\text{Recall} = \frac{TP}{TP + FN},$$

$$\text{F1-score} = 2 \times \frac{Precision \times Recall}{Precision + Recall},$$

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}.$$

4.4 Implementation Details

We adopt the same dataset partitioning strategy used in prior research [26] to ensure consistency and comparability of results. In the vulnerability detection step, we perform training for 50 epochs to fine-tune the model. This model is then applied to the test set to evaluate CodeXplain performance, and the corresponding results are reported. In terms of hardware configuration, the experiments are conducted on a server equipped with an Intel i7-10700F CPU (8 cores), 32 GiB memory, and eight Nvidia RTX 3090 GPUs with a total graphics memory of 192 GB.

For CodeT5, we only use Encoder. This is because its task is to generate a vector representation of the code and the explanations, not to generate new code. Encoder is sufficient for subsequent semantic fusion and vulnerability detection. For hyper-parameters, the maximum length limit of CodeT5 is 1,024, and the rest of the hyper-parameters are set to the default values.[1] During

[1] https://huggingface.co/Salesforce/codet5-base/tree/main.

Table 2. The Performance Evaluation of CodeXplain Is Compared with Four Pre-Trained Models

Methods	Precision (%)	Recall (%)	F1-Score (%)	Accuracy (%)
CodeBERT	91.79	71.04	80.07	81.75
GraphCodeBERT	93.01	81.48	86.48	87.25
CodeT5	92.76	75.90	83.47	84.49
UnixCoder	91.59	82.51	86.79	87.05
CodeXplain	**93.34**	**94.92**	**94.12**	**93.88**

Bold indicates the best performance.

training, the learning rate is initialized to 1×10^{-5} and is optimized using the AdamW optimizer [57]. The batch size is fixed at 16. For LLMs, we chose DeepSeek-Coder-V2 [56] as the base LLM for generating code explanations due to its exceptional performance in code-specific tasks and strong coding reasoning capabilities. DeepSeek-Coder-V2 is an open source Mixture-of-Experts code language model that achieves results comparable to GPT4-Turbo. It also expands its support for programming languages from 86 to 338 (including Solidity). In standard benchmark evaluations, DeepSeek-Coder-V2 outperforms closed source models such as GPT4-Turbo, Claude 3 Opus, and Gemini 1.5 Pro in coding benchmarks. Given its superior reasoning abilities, broad programming language support, and enhanced context processing, DeepSeek-Coder-V2 is an ideal choice for generating smart contract code explanations.

To ensure the robustness and reliability of our experimental results, we employ fivefold cross-validation, where the dataset is partitioned into five subsets. Each model is trained and evaluated five times, with a different subset serving as the test set in each fold. Additionally, we repeat each experiment five times within each fold and report the average performance across all runs as the final result.

For the training of baselines, CodeBERT, GraphCodeBERT, CodeT5, and UnixCoder undergo full fine-tuning to adapt to the SCVD task. CodeLlama uses LoRA for lightweight adaptation, utilizing the final token representation for classification. DeepSeek-Coder-V2, GPT-3.5, GPT-4, and iAudit perform classification by generating label names as task outputs. And to ensure that our experimental results are reproducible, the temperature for all LLMs-based methods is set to 0.

5 Experimental Results

5.1 RQ1: How Does CodeXplain Perform in Comparison to Existing State-of-the-Art Methods?

To assess the effectiveness of CodeXplain, we compare it against 16 representative baseline methods. The experimental results are categorized into three groups: pre-trained models, LLMs, and SCVD methods. We present all quantitative results and provide a comprehensive analysis below.

5.1.1 Performance Comparison with Pre-Trained Models. We first compare the proposed CodeXplain with four pre-trained models, i.e., CodeBERT [14], GraphCodeBERT [16], UnixCoder [15], and CodeT5 [45], based on four metrics: precision, recall, F1-score, and accuracy. All results are reported in Table 2. We observe that CodeXplain exhibits a substantial performance improvement over existing pre-trained models. Specifically, CodeXplain achieves the highest performance across all metrics. Compared to the best baseline, UnixCoder, it improves by 1.91% in precision, 15.04%

in recall, 8.44% in F1-score, and 7.84% in accuracy. These improvements can be attributed to several key factors. First, pre-trained models such as UnixCoder, CodeBERT, and GraphCodeBERT primarily focus on learning general-purpose representations of code but lack an explicit mechanism for understanding vulnerabilities. In contrast, CodeXplain integrates vulnerability-focused explanations generated by LLMs, enabling the model to capture security-specific patterns that pre-trained models may overlook. Second, while pre-trained models rely solely on the source code for vulnerability detection, CodeXplain incorporates multi-perspective explanations that enrich the model's understanding of complex contract semantics. This contextual information enables Coexplain to help the model gain a deeper understanding of the code's logic. Finally, the significant improvement in recall (12.41–23.88%) highlights CodeXplain's ability to reduce FNs, which is crucial in SCVD.

5.1.2 Performance Comparison with LLMs. Subsequently, we compare the CodeXplain with five LLM, i.e., GPT-3.5, GPT-4, CodeLlama-13b, CodeLlama-34b, DeepSeek-Coder-V2, and iAudit. For CodeLlama-13b, CodeLlama-34b, and iAudit, these models are fine-tuned on the dataset to adapt to the SCVD task. For GPT-3.5, GPT-4, and DeepSeek-Coder-V2, we evaluate their performance not only in a zero-shot setting but also in a few-shot setting, where the models receive representative in-context examples to enhance their predictions. To further ensure robustness, we do not rely on a single prompt but instead apply five distinct prompts to assess each smart contract (all prompts can be viewed in the CodeXplain reproducible repository[2]). The final prediction is determined using a majority voting mechanism, where the LLM's responses across different prompts are aggregated to ensure more reliable predictions. This prompt design and voting mechanism are validated in prior research [26] as an effective approach for improving prediction accuracy and consistency.

All results are reported in Table 3. The experimental results show that our CodeXplain is still superior to all LLMs. Although GPT-3.5, GPT-4, and DeepSeek-Coder-V2 show excellent recall, achieving a perfect score of 100%. However, both models have significantly low precision, indicating a high number of FPs, making them less reliable overall in providing accurate vulnerability detection. In addition, LLMs analyze code based on patterns and context rather than executing or deeply understanding the runtime behavior. This limitation can lead to misclassifying non-vulnerable patterns as vulnerable if they superficially resemble known issues [8]. CodeLlama-13b and CodeLlama-34b share similarities with GPT-3.5 and GPT-4 in terms of their high recall but struggle with low precision. Their F1-score reflects the imbalance between recall and precision, indicating that despite detecting many vulnerabilities, they also produce a high number of FPs. Therefore, these models still fall short of providing reliable prediction results for SCVD tasks. In contrast, all metrics of CodeXplain have consistently remained high, in contrast to the above methods. This demonstrates the comprehensive vulnerability detection capabilities of CodeXplain, as well as its reliable prediction of SCVs.

In addition, iAudit shows balanced performance with a precision of 93.12% and a recall of 87.98%. However, its F1-score and accuracy are lower than CodeXplain, indicating it misses some vulnerabilities. In contrast, CodeXplain achieves the highest precision of 93.34% while maintaining an excellent recall of 94.92%. This results in a superior F1-score of 94.12% and the highest accuracy of 93.88%, positioning CodeXplain as the best-performing method. Compared to iAudit, CodeXplain improved by 4.03% and 3.80% in the F1-score and accuracy.

To assess the robustness and consistency of different methods, we visualize the F1-score distributions across five repeated runs for each model using a boxplot, as shown in Figure 5. The boxplot displays the median, **interquartile range (IQR)**, and potential outliers, allowing for a more comprehensive comparison beyond average performance.

[2]https://github.com/chenpp1881/CodeXplain/tree/main/Prompts/GPT_prompt_class.

Table 3. The Performance Evaluation of CodeXplain Is Compared with Five LLMs

Methods	Precision (%)	Recall (%)	F1-Score (%)	Accuracy (%)
GPT-3.5$_{zero\text{-}shot}$	51.84	**100.00**	68.28	52.05
GPT-3.5$_{few\text{-}shot}$	52.51	**100.00**	68.86	53.31
GPT-4$_{zero\text{-}shot}$	51.84	**100.00**	68.28	52.05
GPT-4$_{few\text{-}shot}$	52.51	**100.00**	68.86	53.31
DeepSeek-Coder-V2$_{zero\text{-}shot}$	53.48	**100.00**	69.69	54.24
DeepSeek-Coder-V2$_{few\text{-}shot}$	54.92	**100.00**	70.90	55.36
CodeLlama-13b	51.25	97.32	67.14	50.83
CodeLlama-34b	52.16	94.21	67.15	52.41
iAudit	93.12	87.98	90.47	90.44
CodeXplain	**93.34**	**94.92**	**94.12**	**93.88**

Bold indicates the best performance.

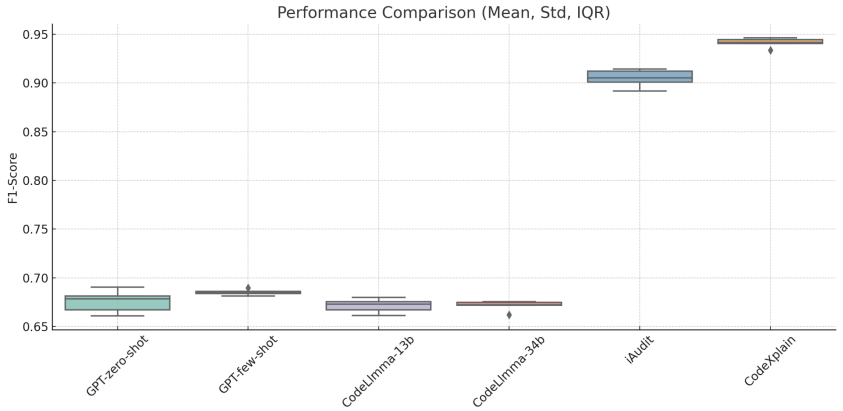

Fig. 5. F1-score distribution of different methods across five runs, including mean, standard deviation, and IQR. IQR, interquartile range.

From the figure, we observe that CodeXplain achieves not only the highest median F1-score but also the narrowest IQR and lowest variance among all methods, indicating both superior performance and strong stability. iAudit performs relatively well but exhibits larger variation across runs, suggesting it may be more sensitive to certain input conditions. In contrast, GPT-based models (both zero-shot and few-shot) and CodeLlama models show significantly lower performance with wider score distributions, highlighting their limited effectiveness and reduced reliability in the SCVD task. Notably, few-shot prompting marginally improves over zero-shot, but still underperforms compared to fine-tuned baselines.

5.1.3 Performance Comparison with SCVD Methods. We then compare CodeXplain with six state-of-the-art SCVD methods, including Slither [13], Mythril [29], DMT [35], EGFL [10], VDCEP [9], and Clear [8].

The initial comparison evaluates CodeXplain against two static analysis tools, Mythril and Slither. The performance of these methods is presented in lines 1 and 2 of Table 4. We observe that CodeXplain exhibits substantial performance improvements over existing static analysis tools. Specifically, it achieves an absolute 27.75% and 25.90% increase in F1-score and accuracy compared to Slither, and a 33.39% and 30.55% improvement over Mythril, respectively.

Table 4. The Performance Evaluation of CodeXplain Is Compared with Four SCVD Methods

Methods	Precision (%)	Recall (%)	F1-Score (%)	Accuracy (%)
Slither	72.49	61.20	66.37	67.98
Mythril	67.91	54.92	60.73	63.33
DMT	79.74	82.35	80.99	80.06
EGFL	83.46	84.10	83.69	83.10
VDCEP	89.41	80.98	84.98	85.22
Clear	87.32	84.81	85.97	85.73
CodeXplain	**93.34**	**94.92**	**94.12**	**93.88**

Bold indicates the best performance.

Beyond outperforming static analysis tools, the experimental results further demonstrate that CodeXplain surpasses all existing deep learning-based methods, achieving the highest performance across all evaluation metrics. To be specific, CodeXplain achieves 8.15% and 8.15% absolute improvement in the F1-score and accuracy over the best baseline method, Clear, respectively. The optimal results achieved by CodeXplain stem from its utilization of multi-perspective code explanations, which facilitate a profound comprehension of the intricate logic and semantics inherent in smart contracts. This sets it apart from alternative approaches that rely solely on traditional supervised learning. By generating human-readable explanations, CodeXplain captures nuanced behaviors that can lead to subtle vulnerabilities, resulting in more thorough detection compared to methods like DMT or EGFL, which may miss such issues.

Answer to RQ1: The CodeXplain utilizes nine perspective prompts to ensure a comprehensive audit of various aspects of the smart contract. It then employs supervised learning techniques to enhance the performance of SCVD and consistently maintain superior results compared to 16 state-of-the-art baselines.

5.2 RQ2: What Is the Contribution of the Code Explanations of LLMs to the SCVD?

To verify the effectiveness of multi-perspective code explanation for SCVD. Here, CodeXplain is evaluated based on different levels of code explanation, with the number of code explanations ranging from 0 (no code explanation) to 9 (full explanation using all prompts). For the order of explanation of the ablation, we perform the ablation using the order shown for each explanation in Section 3.1. Here, when nine types of code explanations are ablated, it is equivalent to conducting SCVD using CodeT5.

The metrics evaluated are precision, recall, F1-score, and accuracy. All results are summarized in Figure 6, where the horizontal coordinate indicates the number of code explanations and the vertical coordinate indicates the value of the evaluation metrics. The order of explanation is the same as in Section 3.1. Next, we perform a trend analysis for each metric.

We observe that CodeXplain integrates all code explanations and achieves the highest precision at 94.34%. Overall, the precision of all models is consistently maintained within the range of 91.89–94.34%. Recall, on the other hand, shows a clear upward trend as more explanations are introduced. Without code explanation, the recall begins at 77.05%. As the number of code explanations increases, the recall steadily rises, reaching its peak value of 94.92% with all nine

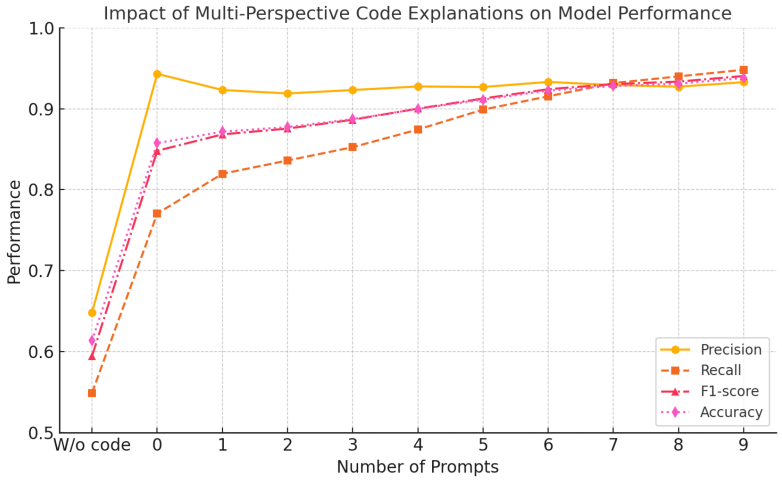

Fig. 6. Impact of multi-perspective code explanations on model performance.

code explanations. This significant improvement highlights the effectiveness of multi-perspective code explanations in enabling the model to detect a greater number of vulnerabilities, emphasizing the value of integrating such explanations for enhanced detection.

The F1-score, which balances both precision and recall, follows a similar improvement. Starting at 84.81% without any explanations, the F1-score progressively increases, reaching 94.12% when all nine code explanations are used. Like recall and F1-score, accuracy also improves consistently as the number of explanations increases. The trend suggests that as CodeXplain integrates more comprehensive explanations from the LLM, it becomes increasingly effective at predicting vulnerabilities in smart contracts. This overall progression underscores the strength of multi-perspective code explanation in enhancing the model's ability to identify SCVs accurately.

In Figure 6, we evaluate the impact of removing the source code by conducting an experiment where the model only utilizes LLM-generated explanations for vulnerability detection. This setting is denoted as "w/o Code." The results show a significant drop in performance compared to the CodeXplain, with the F1-score decreasing to 59.47% and accuracy dropping to 61.35%. This decline suggests that explanations alone are insufficient for accurately identifying vulnerabilities. The primary reason is that in our approach, code explanations serve as auxiliary information rather than the sole basis for detection. CodeXplain is designed to leverage explanations to help the model better understand the semantics of the code, rather than replacing the role of the source code itself. Without access to the original code, the model loses crucial syntactic and structural information necessary for vulnerability detection, leading to degraded performance. This result confirms our initial hypothesis that while LLM-generated explanations enhance the model's understanding, they cannot fully replace source code in the detection process, reinforcing the necessity of jointly considering both code and explanations.

Answer to RQ2: The trend analysis reveals that increasing the number of code explanations in CodeXplain leads to improved performance across all key metrics, especially in recall and F1-score. By using all nine code explanations, CodeXplain achieves its best performance, demonstrating the effectiveness of multi-perspective code explanations in enhancing the SCVD.

Table 5. Impact of Different Pedestal Models on CodeXplain Performance

Methods	Precision (%)	Recall (%)	F1-Score (%)	Accuracy (%)
CodeXplain$_{CodeBERT}$	90.76	91.26	91.01	90.69
CodeXplain$_{GraphCodeBERT}$	91.88	89.62	90.73	90.55
CodeXplain$_{UnixCoder}$	92.70	93.72	93.21	92.95
CodeXplain	**93.34**	**94.92**	**94.12**	**93.88**

Bold indicates the best performance.

5.3 RQ3: How Do Different Pedestal Pre-Training Models Affect the Performance of CodeXplain?

To test whether CodeT5 is the most suitable pedestal model for CodeXplain. We perform variant experiments, using different pre-trained models as pedestal model of CodeXplain, including CodeBERT, GraphCodeBERT, UnixCoder, which are denoted as "CodeXplain$_{model}$." For example, we use CodeXplain$_{CodeBERT}$ to represent the CodeBERT integrated with CodeXplain. They are chosen because, all these pre-trained models are pre-trained on natural language and code data. All experimental settings are the same as described in Section 4. The results are shown in Table 5.

As can be seen from the table, CodeT5 is best suited when used as a pedestal model. CodeXplain outperforms all variants in precision, recall, F1-score, and accuracy, achieving 93.34%, 94.92%, 94.12%, and 93.88% in these metrics, respectively. This is due to CodeT5's dual ability to encode both code data and natural language. This capability not only enables a strong understanding of the syntax, structure but also provides a deeper insight into the context through code explanations.

Answer to RQ3: The dual capability of CodeT5 to encode both code data and natural language explanations enables better auditing of smart contracts. By integrating source code and explanation, CodeT5 enhances its ability to detect SCVs more effectively than other pedestal pre-training models, making it the optimal choice for CodeXplain.

6 Discussion

6.1 Cost-Effectiveness and Practical Deployment

The design of CodeXplain emphasizes practical deployment with minimal operational costs. The model requires only a single LLM interface and is implemented using a locally deployed semantic fusion module with a parameter size of merely 136 M. Consequently, CodeXplain offers state-of-the-art performance with exceptionally low cost, making it a practical choice.

Since CodeXplain relies on DeepSeek-Coder-V2's API to generate explanations, inference costs are tied to API pricing and token consumption. Instead of generating unnecessary details, we use structured, security-focused prompts that maximize the quality of extracted insights while minimizing token usage. On average, analyzing a single contract requires approximately 500–800 tokens per prompt, and each contract is processed with nine prompts, leading to an estimated 4,500–7,200 tokens per contract. These tradeoffs demonstrate that CodeXplain optimally balances explanation quality and API efficiency, ensuring scalability for real-world SCVD applications.

6.2 Limitation

While our approach significantly enhances the interpretability of **smart contract vulnerability detection (SCVD)** through code explanation and logical analysis, it has inherent limitations that

restrict its ability to detect certain classes of vulnerabilities. In particular, our method struggles to identify execution-time vulnerabilities and security pattern violations that require dynamic analysis or explicit security rule enforcement. Two notable examples of vulnerabilities that cannot be effectively detected by our approach are *Call Stack Depth Limitation* and *Use of ORIGIN for Authentication*.

The *Call Stack Depth Limitation* vulnerability arises due to constraints in the EVM, which imposes a maximum call stack depth of 1,024. When a contract makes deeply nested external calls, it risks exceeding this limit. If the call stack depth is exhausted, contract execution fails, potentially leading to unintended denial-of-service conditions. Our method does not track execution-time behavior or simulate transaction execution under various conditions. Because the stack depth limit is only reached during real contract execution, it is impossible to determine from code alone whether a given contract will exceed the limit.

Another vulnerability that our method cannot effectively detect is the *Use of ORIGIN for Authentication*. The Solidity global variable tx.origin returns the original sender of a transaction, even if the transaction has been forwarded through multiple contracts. Developers unfamiliar with this mechanism may mistakenly use tx.origin for access control, assuming it reliably identifies the caller. However, attackers can exploit this by tricking an authorized user into interacting with a malicious contract, which then invokes the protected function on their behalf, effectively bypassing authentication. Our method relies on code explanation. While our approach can describe the behavior of tx.origin, it does not inherently recognize it as a security flaw.

Our current evaluation focuses on 14 common vulnerability types that are well-defined and studied in prior SCVD literature [11, 31, 33]. These types are also supported by mainstream static analysis tools, which makes large-scale labeling and benchmarking feasible. However, we acknowledge that many other practically relevant vulnerabilities are not covered in our current setting. The primary reason is the lack of publicly available datasets with reliable and fine-grained annotations for these types. Identifying such functional bugs often requires domain-specific analysis and runtime context, which makes scalable labeling challenging. We consider extending CodeXplain to detect these broader classes of vulnerabilities as a promising direction for future work.

7 Threats of Validity

The threats to external validity come from the datasets. To reduce the former threat, our dataset is constructed based on reliable smart contract audit reports. Each data entry is labeled as either a vulnerable smart contract or a non-vulnerable one.

The threats to internal validity come from the implementation of CodeXplain and the compared vulnerability detection methods. To mitigate these threats, we implement CodeXplain based on PyTorch and off-the-shelf third-party libraries and adopt the reproducible package of the compared methods. Moreover, the reliability of the explanations generated by LLMs also threatens the effectiveness of CodeXplain. To ensure the reliability of the experiment, we use the LLM, DeepSeek-Coder-V2, which is pre-trained specifically for code tasks, to explain the code. Second, we conduct a manual inspection of 100 randomly selected contract explanations and compare these explanations with the contents of corresponding audit reports to verify their accuracy. The results indicate that the majority of the generated explanations align well with the audited contract logic, confirming their quality.

The threats to construct validity come from the metrics used to measure the performance of studied vulnerability detection methods. To reduce these threats, we use precision, recall, F1-score, and accuracy as previous work did [49, 53–55] since vulnerability detection can be regarded as a dichotomous task.

8 Related Work

8.1 SCVD

SCVD is regarded as a vital task for blockchain security. Many researchers proposed deep learning methods to improve the detection accuracy of SCVs. Specifically, Qian et al. [34] proposed snippet representation of smart contracts to extract important semantic information and utilized Bi-LSTM with attention to detect reentrancy vulnerabilities. Zhuang et al. [57] represented the semantic structure of the functions of the smart contract via contract graphs and used the GNNs to detect SCVs. Liu et al. [24] proposed a vulnerability detection method that combines deep learning with expert rules. This method converted the code into a semantic graph. Then, the graph features and expert rules were fused to verify SCVs. Liu et al. [25] encoded expert knowledge as numerical features and then converted the source code into semantic graphs to capture deep graph features. The expert knowledge and graph features are combined to conduct SCVD by GNN. Wu et al. [47] proposed a novel approach, Peculiar, which used a pre-trained technique for detecting the reentrancy vulnerability based on dataflow graph. The experiments were conducted on a large dataset, and the results showed that the Peculiar achieved promising performance. Zhang et al. [51] proposed a method named ReVulDL for reentrancy vulnerabilities. It used a graph-based pre-training model to detect reentrancy vulnerabilities and utilized interpretable machine learning to locate the suspicious statements in smart contract. Yu et al. [50] proposed a deep learning-based framework for SCVD, called DeeSCVHunter, targeting reentrancy and time dependence vulnerabilities. The key innovation is the introduction of vulnerability candidate slice, which helps models capture critical vulnerability-related code segments. Mi et al. [28] proposed an SCVD method to address the challenge of missing source code by extracting feature vectors from bytecode. It utilizes a metric learning-based deep neural network for classification, improving detection accuracy. Ye et al. [48] proposed a signature-based vulnerability detection framework for smart contracts, which extracts structural program features from both vulnerable and benign contracts to generate expressive vulnerability signatures. Cai et al. [5] proposed a GNN-based framework for SCVD, addressing the limitations of rule-based methods by leveraging graph representations of smart contract functions. The approach constructs a hybrid graph combining AST, CFG, and PDG to capture both syntactic and semantic features. Qian et al. [35] proposed a cross-modality mutual learning framework and a single-modality student network to enhance the detection of vulnerabilities in the bytecode of smart contracts. Chen et al. [8] employed a contrastive learning model to capture fine-grained correlation information between smart contracts, integrating it with semantic contract information for SCV detection. Cheng et al. [9] deconstructed code structure graphs into execution paths and extracted feature representations from these paths to identify vulnerabilities. Cheng et al. [10] represented bytecode as a control flow graph and utilized a graph neural network to extract node features for vulnerability assessment.

However, these approaches inherently struggle to fully understand the complex semantics and logic embedded in smart contract code. As a result, their performance often falls short of optimal. Instead, our CodeXplain generate multi-perspective explanations of the code, providing a deeper understanding of code structure and behavior by LLMs, offering a more effective solution for securing smart contracts. Our CodeXplain not only provides multi-perspective code explanations but also combines the excellent predictive power of supervised learning methods to improve the performance of SCVD.

8.2 LLM-Based Vulnerability Detection

In recent years, with the popularity of LLM, many researchers use LLM to conduct vulnerability detection tasks. For example, GPTScan [42] combined GPT with static analysis to detect smart

contract logic vulnerabilities. It breaks down logic vulnerabilities into scenarios and properties, leveraging GPT for code understanding and using static confirmation to reduce FPs. LLM4Vuln [41] decoupled LLMs' vulnerability reasoning capabilities from other skills, such as retrieving knowledge and following instructions. It evaluates how vulnerability reasoning can be enhanced through additional capabilities and identified zero-day vulnerabilities in smart contracts. Ullah et al. [44] evaluated whether LLMs can reliably identify security-related bugs by analyzing eight leading models across 228 code scenarios. Thapa et al. [43] leveraged large transformer-based language models to detect software vulnerabilities in C/C++ code. Mathews et al. [27] explored the use of LLMs for Android vulnerability detection, addressing the limitations of static and dynamic analysis tools. Purba et al. [32] evaluated the performance of four well-known LLMs in detecting software vulnerabilities using two public datasets, finding a significant performance gap compared to static analysis tools due to high false-positive rates. Li et al. [22] introduced LLift, an automated agent that integrates LLMs with static analysis to improve bug detection, specifically targeting use-before-initialization bugs. David et al. [12] explored the use of large LLMs for smart contract security audits, focusing on optimizing prompt engineering for improved analysis. This highlights the potential of LLMs for enhancing audit efficiency while still requiring manual auditor involvement for accuracy. Jiang et al. [21] propose using LLMs to automate the detection of gas-wasting code smells in Solidity smart contracts, addressing the inefficiency of manual analysis. By identifying 26 code smells, the method achieves significant reductions in deployment and message call costs, demonstrating its practical impact on optimizing gas usage. Wang et al. [46] propose SliSE, a two-stage detection tool that combines program slicing and symbolic execution to effectively identify reentrancy vulnerabilities in complex smart contracts. By analyzing the Inter-contract Program Dependency Graph to locate suspicious paths and verifying their reachability, SliSE enhances both the accuracy and efficiency of vulnerability detection. iAudit [26] combined fine-tuning and LLM-based agents to enhance both vulnerability detection and justification generation. It uses a two-stage fine-tuning process with Detector and Reasoner models to improve cause identification.

However, these studies are not targeted at analyzing specific aspects related to vulnerabilities. Moreover, they fail to integrate the powerful capabilities of supervised learning methods, instead relying solely on knowledge from pre-trained datasets or some vulnerable code fragments. This limited approach prevents them from achieving optimal results in vulnerability detection.

9 Conclusion and Future Work

The SCVs can severely compromise the security of transactions within the blockchain ecosystem. However, traditional deep learning methods struggle to comprehensively understand smart contract code. The absence of supervised learning with labeled data in LLMs results in subpar vulnerability detection performance. To address this challenge, we propose a novel neural network model, CodeXplain, which combines the strengths of LLM-generated code explanations with the supervised leaning method, providing a more thorough understanding of smart contract code and improving the accuracy of vulnerability detection. This article details the CodeXplain, including the design of the fine-grained prompts, the process of generating code explanations, and the fine-tuning of supervised learning method, as well as an evaluation of its performance in the SCVD. The experimental results show that the proposed CodeXplain outperforms the state-of-the-art detection methods. In future work, we aim to expand our experiments to a wider range of smart contract datasets. Additionally, further refinement of the CodeXplain framework, such as incorporating more advanced language models, may enhance its performance in detecting complex vulnerabilities. Finally, future research could explore hybrid methods that combine our current approach with complementary techniques, such as static and dynamic analysis, to create more comprehensive solutions for SCVD.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 Technical Report. arXiv:2303.08774. Retrieved from https://arxiv.org/abs/2303.08774

[2] Tareq Ahram, Arman Sargolzaei, Saman Sargolzaei, Jeff Daniels, and Ben Amaba. 2017. Blockchain technology innovations. In *IEEE Technology & Engineering Management Conference*, 137–141.

[3] Shaima A. L. Amri, Leonardo Aniello, and Vladimiro Sassone. 2023. A review of upgradeable smart contract patterns based on OpenZeppelin technique. *The Journal of the British Blockchain Association* 6, 1 (2023).

[4] William E. Bodell III, Sajad Meisami, and Yue Duan. 2023. Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32nd USENIX Security Symposium*, 1829–1846.

[5] Jie Cai, Bin Li, Jiale Zhang, Xiaobing Sun, and Bing Chen. 2023. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. *Journal of Systems and Software* 195 (2023), 111550.

[6] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 2024. Smart contract and DeFi security tools: Do they meet the needs of practitioners? In *46th IEEE/ACM International Conference on Software Engineering*, 1–13.

[7] Weili Chen, Zibin Zheng, Edith C.-H. Ngai, Peilin Zheng, and Yuren Zhou. 2019. Exploiting blockchain data to detect smart ponzi schemes on ethereum. *IEEE Access* 7 (2019), 37575–37586.

[8] Yizhou Chen, Zeyu Sun, Zhihao Gong, and Dan Hao. 2024. Improving smart contract security with contrastive learning-based vulnerability detection. In *IEEE/ACM 46th International Conference on Software Engineering*, 1–11.

[9] Jianxin Cheng, Yizhou Chen, Yongzhi Cao, and Hanpin Wang. 2024. A vulnerability detection framework by focusing on critical execution paths. *Information and Software Technology* 174 (2024), 107517.

[10] Jianxin Cheng, Yizhou Chen, Yongzhi Cao, and Hanpin Wang. 2024. A vulnerability detection framework with enhanced graph feature learning. *Journal of Systems and Software* 216 (2024), 112118.

[11] Hanting Chu, Pengcheng Zhang, Hai Dong, Yan Xiao, Shunhui Ji, and Wenrui Li. 2023. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Information and Software Technology* 159 (2023), 107221.

[12] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. 2023. Do you still need a manual smart contract audit? arXiv:2306.12338. Retrieved from https://arxiv.org/abs/2306.12338

[13] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A static analysis framework for smart contracts. In *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. IEEE, 8–15.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. arXiv:2002.08155. Retrieved from https://arxiv.org/abs/2002.08155

[15] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. arXiv:2203.03850. Retrieved from https://arxiv.org/abs/2203.03850

[16] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. arXiv:2009.08366. Retrieved from https://arxiv.org/abs/2009.08366

[17] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, Y. K. Li, et al. 2024. DeepSeek-Coder: When the large language model meets programming–The rise of code intelligence. arXiv:2401.14196. Retrieved from https://arxiv.org/abs/2401.14196

[18] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, et al. 2021. Pre-trained models: Past, present and future. *AI Open* 2 (2021), 225–250.

[19] Daojing He, Rui Wu, Xinji Li, Sammy Chan, and Mohsen Guizani. 2023. Detection of vulnerabilities of blockchain smart contracts. *IEEE Internet of Things Journal* 10, 14 (2023), 12178–12185.

[20] Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. 2023. Large language model-powered smart contract vulnerability detection: New perspectives. In *5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications*. IEEE, 297–306.

[21] Jinan Jiang, Zihao Li, Haoran Qin, Muhui Jiang, Xiapu Luo, Xiaoming Wu, Haoyu Wang, Yutian Tang, Chenxiong Qian, and Ting Chen. 2025. Unearthing gas-wasting code smells in smart contracts with large language models. *IEEE Transactions on Software Engineering* 51, 4 (2025), 879-903.

[22] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. The Hitchhiker's guide to program analysis: A journey with large language models. arXiv:2308.00245. Retrieved from https://arxiv.org/abs/2308.00245

[23] Bingchang Liu, Chaoyu Chen, Zi Gong, Cong Liao, Huan Wang, Zhichao Lei, Ming Liang, Dajun Chen, Min Shen, Hailian Zhou, et al. 2024. Mftcoder: Boosting code LLMs with multitask fine-tuning. In *30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 5430–5441.

[24] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. arXiv:2106.09282. Retrieved from https://arxiv.org/abs/2106.09282

[25] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2023. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering* 35, 2 (2023), 1296-1310.

[26] Wei Ma, Daoyuan Wu, Yuqiang Sun, Tianwen Wang, Shangqing Liu, Jian Zhang, Yue Xue, and Yang Liu. 2024. Combining fine-tuning and LLM-based agents for intuitive smart contract auditing with justifications. arXiv:2403.16073. Retrieved from https://arxiv.org/abs/2403.16073

[27] Noble Saji Mathews, Yelizaveta Brus, Yousra Aafer, Mei Nagappan, and Shane McIntosh. 2024. Llbezpeky: Leveraging large language models for vulnerability detection. arXiv:2401.01269. Retrieved from https://arxiv.org/abs/2401.01269

[28] Feng Mi, Zhuoyi Wang, Chen Zhao, Jinghui Guo, Fawaz Ahmed, and Latifur Khan. 2021. VSCL: Automating vulnerability detection in smart contracts with deep learning. In *IEEE International Conference on Blockchain and Cryptocurrency*. IEEE, 1–9.

[29] Bernhard Mueller. 2017. A Framework for Bug Hunting on the Ethereum Blockchain. Retrieved from https://github.com/ConsenSys/mythril

[30] K. Lakshmi Narayana and K. Sathiyamurthy. 2023. Automation and smart materials in detecting smart contracts vulnerabilities in blockchain using deep learning. *Materials Today: Proceedings* 81 (2023), 653–659.

[31] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph Liu, and Robin Doss. 2019. Security analysis methods on ethereum smart contract vulnerabilities: A survey. arXiv:1908.08605. Retrieved from https://arxiv.org/abs/1908.08605

[32] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, and Bill Chu. 2023. Software vulnerability detection using large language models. In *IEEE 34th International Symposium on Software Reliability Engineering Workshops*. IEEE, 112–119.

[33] Peng Qian, Zhenguang Liu, Qinming He, Butian Huang, Duanzheng Tian, and Xun Wang. 2022. Smart contract vulnerability detection technique: A survey. arXiv:2209.05872. Retrieved from https://arxiv.org/abs/2209.05872

[34] Peng Qian, Zhenguang Liu, Qinming He, Roger Zimmermann, and Xun Wang. 2020. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access* 8 (2020), 19685–19695.

[35] Peng Qian, Zhenguang Liu Yifang Yin, and Qinming He. 2023. Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode. In *ACM Web Conference 2023*, 2220–2229.

[36] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences* 63, 10 (2020), 1872–1897.

[37] Nicole Radziwill. 2018. Blockchain revolution: How the technology behind bitcoin is changing money, business, and the world. *The Quality Management Journal* 25, 1 (2018), 64–65.

[38] Martin Röscheisen, Michelle Baldonado, Kevin Chang, Luis Gravano, Steven Ketchpel, and Andreas Paepcke. 1998. The Stanford InfoBus and its service layers: Augmenting the internet with higher-level information management protocols. *Digital Libraries in Computer Science: The MeDoc Approach* (1998), 213–230.

[39] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. arXiv:2308.12950. Retrieved from https://arxiv.org/abs/2308.12950

[40] Alexander Savelyev. 2017. Contract law 2.0: 'Smart' contracts as the beginning of the end of classic contract law. *Information & Communications Technology Law* 26, 2 (2017), 116–134.

[41] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. 2024. LLM4Vuln: A unified evaluation framework for decoupling and enhancing LLMs' vulnerability reasoning. arXiv:2401.16185. Retrieved from https://arxiv.org/abs/2401.16185

[42] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. Gptscan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In *IEEE/ACM 46th International Conference on Software Engineering*, 1–13.

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

[44] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2023. Can large language models identify and reason about security vulnerabilities? Not yet. arXiv:2312.12575. Retrieved from https://arxiv.org/abs/2312.12575

[45] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859. Retrieved from https://arxiv.org/abs/2109.00859

[46] Zexu Wang, Jiachi Chen, Yanlin Wang, Yu Zhang, Weizhe Zhang, and Zibin Zheng. 2024. Efficiently detecting reentrancy vulnerabilities in complex smart contracts. In *ACM on Software Engineering*, Vol. 1, 161–181.

[47] Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *IEEE 32nd International Symposium on Software Reliability Engineering*, 378–389.

[48] Jiaming Ye, Mingliang Ma, Yun Lin, Lei Ma, Yinxing Xue, and Jianjun Zhao. 2022. Vulpedia: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures. *Journal of Systems and Software* 192 (2022), 111410.
[49] Jiaojiao Yu, Kunsong Zhao, Jin Liu, Xiao Liu, Zhou Xu, and Xin Wang. 2022. Exploiting gated graph neural network for detecting and explaining self-admitted technical debts. *Journal of Systems and Software* 187 (2022), 111219.
[50] Xingxin Yu, Haoyue Zhao, Botao Hou, Zonghao Ying, and Bin Wu. 2021. Deescvhunter: A deep learning-based framework for smart contract vulnerability detection. In *International Joint Conference on Neural Networks*. IEEE, 1–8.
[51] Zhuo Zhang, Yan Lei, Meng Yan, Yue Yu, Jiachi Chen, Shangwen Wang, and Xiaoguang Mao. 2022. Reentrancy vulnerability detection and localization: A deep learning based two-phase approach. In *37th IEEE/ACM International Conference on Automated Software Engineering*, 1–13.
[52] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying exploitable bugs in smart contracts. In *IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 615–627.
[53] Kunsong Zhao, Jin Liu, Zhou Xu, Li Li, Meng Yan, Jiaojiao Yu, and Yuxuan Zhou. 2021. Predicting crash fault residence via simplified deep forest based on a reduced feature set. In *IEEE/ACM 29th International Conference on Program Comprehension*, 242–252.
[54] Kunsong Zhao, Jin Liu, Zhou Xu, Xiao Liu, Lei Xue, Zhiwen Xie, Yuxuan Zhou, and Xin Wang. 2022. Graph4Web: A relation-aware graph attention network for web service classification. *Journal of Systems and Software* 190 (2022), 111324.
[55] Kunsong Zhao, Zhou Xu, Meng Yan, Tao Zhang, Dan Yang, and Wei Li. 2021. A comprehensive investigation of the impact of feature selection techniques on crashing fault residence prediction models. *Information and Software Technology* 139 (2021), 106652.
[56] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence. arXiv:2406.11931. Retrieved from https://arxiv.org/abs/2406.11931
[57] YuanZhuang, ZhenguangLiu, PengQian, QiLiu, XiangWang, and QinmingHe. 2020. Smart contract vulnerability detection using graph neural network. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence*, 3283–3290.

Received 9 December 2024; revised 15 June 2025; accepted 29 July 2025