SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

June 2025, Vol. 68, Iss. 6, 160101:1–160101:16
https://doi.org/10.1007/s11432-024-4402-2

Special Topic: Enabling Techniques and Cutting-Edge Applications of Foundation Models

Agent4Vul: multimodal LLM agents for smart contract vulnerability detection

Wanqing JIE[1,2], Wangjie QIU[1,2*], Haofu YANG[1], Muyuan GUO[1], Xinpeng HUANG[1,2], Tianyu LEI[1], Qinnan ZHANG[1*], Hongwei ZHENG[3] & Zhiming ZHENG[1,2]

[1]*The Institute of Artificial Intelligence, Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing, Beihang University, Beijing 100191, China*
[2]*Zhongguancun Laboratory, Beijing 100080, China*
[3]*Beijing Academy of Blockchain and Edge Computing, Beijing 100190, China*

Received 1 December 2024/Revised 9 February 2025/Accepted 17 February 2025/Published online 19 May 2025

Abstract Smart contract vulnerabilities have emerged as a significant threat to blockchain system security under the Web 3.0 ecosystem. According to recent research, large language models (LLMs) have demonstrated immense potential in smart contract security audits but still lack the capability for effective vulnerability detection. Consequently, leveraging the capabilities of LLMs to effectively enhance the performance of smart contract vulnerability detection remains a critical challenge. In this paper, we propose Agent4Vul, a novel framework utilizing multimodal LLM agents to enhance smart contract vulnerability detection. Specifically, we design two LLM-based agents: Commentator and Vectorizer. The Commentator agent generates comments for the source code, while the Vectorizer agent converts contents into vector representations. Subsequently, we develop a multimodal learning architecture comprising the semantic branch and the graph branch, which collectively integrate features from the source code, generated comments, and the bytecode control flow graph (CFG). We empirically evaluate a large-scale real dataset of smart contracts and compare 19 state-of-the-art baseline approaches. The results show that Agent4Vul achieves (1) superior performance over all baseline approaches on the four types of common vulnerabilities in real attacks; (2) 3.61%–16.32% higher F1-scores than existing artificial intelligence (AI) approaches, outperforming even advanced LLMs like GPT-4o and o1. This work lays a solid foundation for LLM-driven smart contract security and introduces innovative applications of LLMs in software engineering.

Keywords smart contract, vulnerability detection, LLM, agent, multimodal learning

Citation Jie W Q, Qiu W J, Yang H F, et al. Agent4Vul: multimodal LLM agents for smart contract vulnerability detection. Sci China Inf Sci, 2025, 68(6): 160101, https://doi.org/10.1007/s11432-024-4402-2

1 Introduction

Smart contracts have been an essential component with the introduction of blockchain technology, especially with the flourishing of Ethereum [1]. Due to the inherent openness, transparency, and immutability, smart contracts have become increasingly popular [2–4]. However, exploitable vulnerabilities in smart contracts typically attract hackers, resulting in huge financial losses, more than $7.85 billion by May 2024, according to DefiLlama hacks [5]. Hence, it is imperative to expedite the development of a novel and effective approach for detecting smart contract vulnerabilities in order to rapidly discover and fix security issues in blockchain platforms [6–8].

Existing methods for smart contract vulnerability detection fall into two main categories: rule-based and data-driven approaches. Rule-based methods [9–16] rely on predefined rules or pattern matching to identify vulnerabilities. While these methods are effective for known issues, they struggle to generalize to the diverse and evolving nature of smart contract vulnerabilities. Moreover, the construction and maintenance of rule sets are labor-intensive and require extensive domain expertise, which limits their scalability. In contrast, data-driven approaches [17–22] leverage neural networks to learn patterns of vulnerabilities from data, offering greater flexibility and adaptability. The advent of large language models (LLMs) has spurred growing interest in their application to smart contract security auditing [23–27].

* Corresponding author (email: wangjieqiu@buaa.edu.cn, zhangqn@buaa.edu.cn)

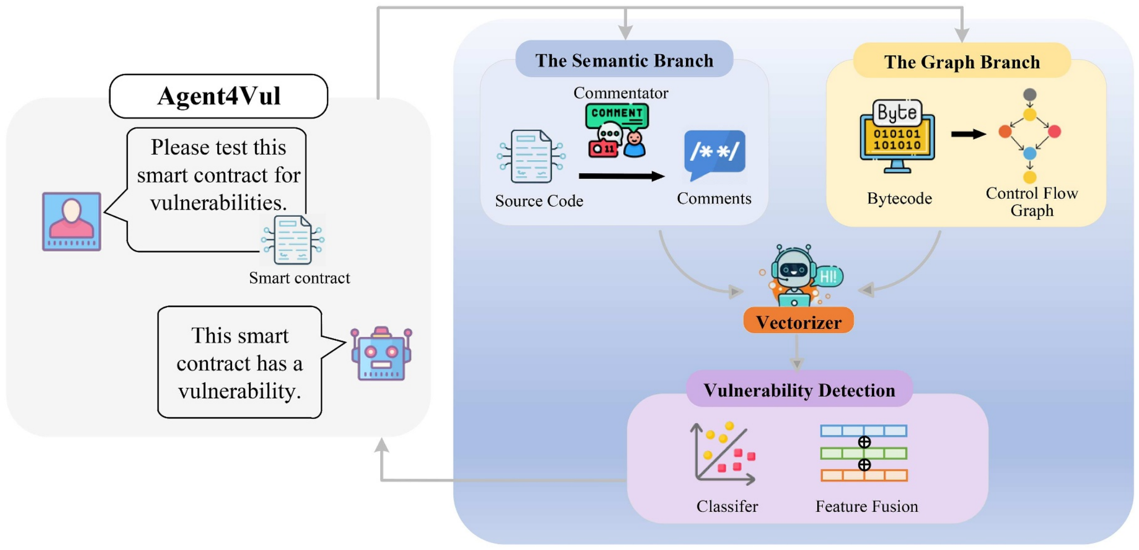


Figure 1 (Color online) Agent4Vul vulnerability detection flowchart.

These models demonstrate remarkable capabilities in code comprehension, making them promising tools for vulnerability detection. However, two critical challenges hinder the effectiveness of LLMs in this domain.

C1: limited detection performance. Systematic evaluations reveal that even advanced LLMs like GPT-4 achieve an F1 score of only 41.1% when simulating professional audit procedures using chain-of-thought (CoT) prompts [24]. This underscores the limitations of existing LLM-based methods in accurately identifying vulnerabilities.

C2: diverse vulnerability characteristics. Different types of vulnerabilities exhibit varying underlying mechanisms and detection complexities, resulting in inconsistent LLM performance across categories [23]. Addressing this heterogeneity requires a framework capable of adapting to the diverse characteristics of vulnerabilities.

While LLMs possess strong code comprehension capabilities, a significant gap remains in effectively translating these capabilities into robust vulnerability detection frameworks. This challenge underscores the need for a sophisticated approach that leverages the multimodal processing capabilities of LLMs to address the unique characteristics of smart contract vulnerabilities and improve detection performance.

Scope and contributions. In this paper, we propose Agent4Vul, an advanced framework that leverages multimodal LLM-based agents for detecting vulnerabilities in smart contracts. To address the limitations of existing approaches, Agent4Vul combines the code comprehension capabilities of LLMs with multimodal learning techniques to effectively detect vulnerabilities with diverse characteristics, as shown in Figure 1. The framework introduces two intelligent agents: Commentator and Vectorizer, which are designed to handle the multimodal nature of smart contract features. The Commentator agent generates detailed comments for smart contract source code, enhancing the semantic understanding of the code, while the Vectorizer agent transforms the content, including the comments and source code, into high-dimensional vector representations to extract critical features. Building on these agents, the framework incorporates two feature processing branches: the semantic branch and the graph branch, which collectively enable comprehensive multimodal vulnerability detection. (1) In the semantic branch (Subsection 3.2), the Commentator agent employs five tailored prompt-based strategies to generate comments, translating LLMs' deep code comprehension capabilities into actionable insights for vulnerability detection. The Vectorizer agent subsequently encodes both the comments and the source code into vector representations, facilitating the extraction of key semantic features from the smart contract. (2) In the graph branch (Subsection 3.3), the Vectorizer agent processes the control flow graph (CFG) of the smart contract's bytecode, embedding node information into high-dimensional representations. These embeddings are then passed through a graph attention network (GAT), leveraging a message-passing mechanism to effectively capture the structural information of the smart contract. (3) Finally (Subsection 3.4), features from the semantic and graph branches are integrated using a multimodal feature fusion strategy and fed into a classification model, achieving comprehensive and robust vulnerability detection. This approach fully harnesses the semantic insights from the source code and the structural characteristics of the bytecode, delivering a robust multimodal collaborative framework for smart contract vulnerability detection powered by LLM-based agents. (4) We conduct extensive experiments to evaluate the performance of Agent4Vul (Subsection 4), benchmarking it against 19 state-of-the-art (SOTA) vulnerability detection methods. The experiments utilize a large-scale dataset comprising over 40000 real-world smart contracts [21]. The results demonstrate that Agent4Vul significantly outperforms existing approaches, achieving notable improvements in F1 scores ranging from 3.61%–16.32%.

To summarize, the primary contributions of this paper are as follows.

- **A robust multimodal LLM agent-based detection framework.** We present Agent4Vul, a novel framework built upon multimodal LLM-based agents to address both the performance limitations of existing LLM-based methods and the challenge of adapting to diverse types of vulnerabilities. By integrating semantic and graph structure information extracted through LLM-driven code comprehension and multimodal learning, Agent4Vul ensures comprehensive and accurate vulnerability detection across various categories of smart contract vulnerabilities.
- **Dual agents guided vulnerability representation.** By introducing two LLM-based agents, Agent4Vul transforms the code comprehension capabilities of LLMs into actionable vulnerability representations. The Commentator agent leverages the flexible prompt strategy library (FPSL), comprising five distinct types of prompts, which guide the reasoning process during comment generation to identify potential vulnerabilities. These generated comments, together with the source code, are subsequently processed by the Vectorizer agent in the semantic branch, enabling the extraction of enriched and representative features for enhanced vulnerability detection.
- **Framework implementation and comprehensive evaluation.** We implement Agent4Vul and validate its effectiveness through extensive experiments on a large-scale dataset comprising over 40000 real-world smart contracts. The results demonstrate that Agent4Vul outperforms 19 SOTA methods, achieving substantial improvements in F1 scores. Compared to rule-based, deep learning-based, and LLM-based approaches, Agent4Vul consistently delivers superior performance, highlighting its advantages in both accuracy and efficiency for smart contract vulnerability detection.

2 Related work

There are various research approaches in the field of smart contract vulnerability detection, which can be categorized into rule-based methods, deep learning-based methods, and LLM-based methods.

2.1 Rule-based methods

Rule-based methods use preset rules and patterns to find possible vulnerabilities in smart contracts, mainly through symbolic execution techniques and static analysis. Symbolic execution and control flow analysis were used in early efforts (Oyente [12] and Mythril [13]) to simulate smart contract execution paths in order to uncover vulnerabilities. More advanced static analysis techniques were later offered by tools like Securify [14] and Slither [15], which used control flow reasoning and semantic information from data flow to increase detection accuracy. Smartcheck [10] focused on detecting integer overflows and underflows using symbolic execution, while Osiris [11] concentrated on identifying common security flaws using static analysis using predefined rule sets. Additionally, branch distance-driven fuzz testing was used by sFuzz [9] to create test cases for potential vulnerabilities in a dynamic manner. These techniques are excellent at identifying known vulnerabilities through static and dynamic analysis methods; however, they are less successful against complicated undiscovered vulnerabilities since they frequently rely on established rules and expert knowledge.

2.2 Deep learning-based methods

Deep learning-based methods identify vulnerabilities by learning features from large-scale datasets, thereby overcoming the drawbacks of rule-based systems. Using sequential models, Vanilla-RNN [16] analyzes smart contract bytecode sequences, taking advantage of the model's capacity to recognize features in bytecode to identify vulnerabilities. In a similar vein, ReChecker [17] uses execution sequence analysis to automatically find vulnerabilities in reentrancy attacks. Control flow graphs of smart contracts are used by graph convolutional networks (GCN) [18] to extract structural information for vulnerability identification. To improve detection performance even more, TMP [19] developed a hybrid model that combines expert patterns and graph structure information. In order to increase detection accuracy, AME [20] combined expert knowledge and deep learning by employing interpretable graph characteristics. Furthermore, cross-modal mutual learning strategies were used by methods like SMS and DMT [21] to improve detection precision. While DMT uses a dual-modal teacher network with both source code and bytecode inputs, SMS uses a single-modal student network. This allows knowledge transfer through mutual learning to improve detection performance. By fusing generative models' potent reasoning powers with cross-modal learning procedures, these methods greatly increase detection accuracy and comprehensiveness while lowering reliance on expert knowledge.

2.3 LLM-based methods

Since LLMs are widely used in many different fields, a number of LLM-based techniques have been developed that take advantage of generative models' natural language comprehension and reasoning abilities (e.g., GPT-4) for vulnerability discovery. Chen et al. [23] examined the differences between ChatGPT and current vulnerability detection technologies in order to identify vulnerabilities in smart contracts using GPT-3.5 and GPT-4. Simple quick engineering or chain-of-thought simulation, as shown by Du et al. [24], improves the LLM's comprehension of contract code; nonetheless, its direct application to detection tasks is still not ideal, with an F1 score of only 41.1%. Converting the model's capacity for code comprehension into practical vulnerability identification is the difficult part. In order to improve detection performance and decrease false positives, GPTLENS [25] uses a more sophisticated role differentiation approach. The detection task is divided into two phases: creation and discrimination. During each phase, the LLM carries out auditing and review tasks independently. This approach is very reliant on LLM, and its success is extremely contingent upon role design and particular job circumstances, necessitating a great deal of data validation. Adding to this, TrustLLM [26] shows significant promise by segmenting the role duties of the LLM and performing multi-stage model fine-tuning according to role kinds. However, the size of the contract dataset and the types of vulnerabilities still have a significant impact on the efficiency of fine-tuning and detection performance, requiring additional validation in multi-type vulnerability detection assignments. Furthermore, GPTScan [27] serves mostly as a code auxiliary analysis tool, gathering scenario details and important attributes for use by static detection technologies that come after. In order to identify logical flaws in smart contracts, it combines GPT with static analysis techniques. Rule-based preprocessing and post-confirmation are used to reduce false positives.

To the best of our knowledge, our Agent4Vul system is the first multimodal LLM agents with flexible prompt strategy library reasoning for smart contracts, converting LLM's code comprehension capabilities into vulnerability detection powers. We have tested this framework's efficacy using a large amount of smart contract data and a range of common contract vulnerability types. By addressing the limitations of current approaches, Agent4Vul displays greater flexibility and accuracy in handling multimodal data integration and complex contract scenarios.

3 Proposed Agent4Vul framework

To address the challenges outlined earlier, we propose a novel approach called Agent4Vul, which leverages multimodal LLM agents to identify vulnerabilities in smart contracts. From the perspective of LLMs, smart contract vulnerabilities are categorized into two primary branches: the semantic branch and the graph branch. These branches are then further developed into detection pathways, integrating agents and multimodal learning to enhance vulnerability detection.

3.1 Overview of Agent4Vul

We present Agent4Vul, a cutting-edge multimodal framework for smart contract vulnerability detection, as shown in Figure 2 and Algorithm 1. In the context of the paper, multimodality in smart contracts refers to the combination of the following modalities.

Source code. The original code is written in high-level programming languages (e.g., Solidity) that define the logic and structure of the smart contract.

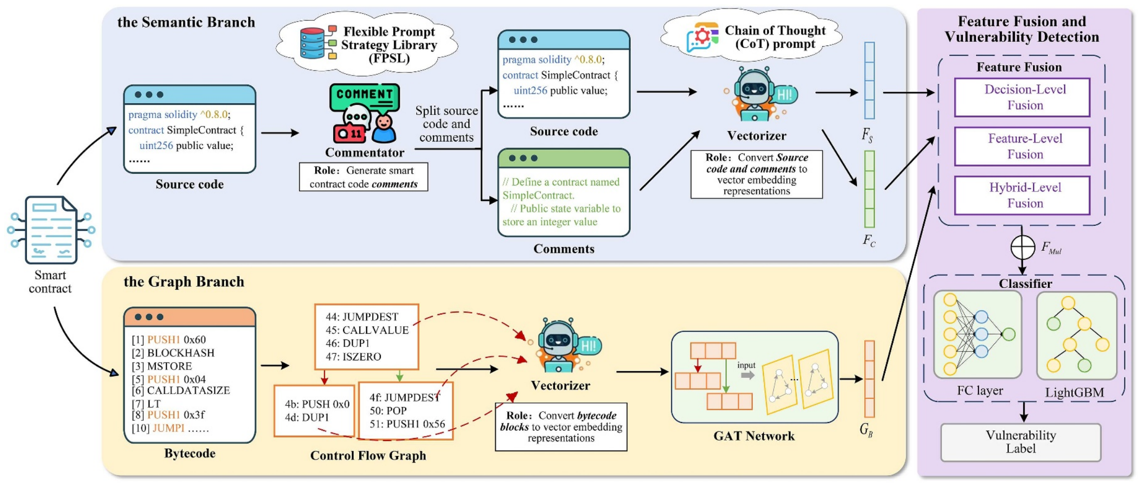

the Semantic Branch
Flexible Prompt Strategy Library (FPSL)
Source code
Commentator
Role: Generate smart contract code comments
Split source code and comments
Source code
Comments
Chain of Thought (CoT) prompt
Vectorizer
Role: Convert Source code and comments to vector embedding representations
F_S
F_C
Smart contract
the Graph Branch
Bytecode
Control Flow Graph
Vectorizer
Role: Convert bytecode blocks to vector embedding representations
GAT Network
G_B
Feature Fusion and Vulnerability Detection
Feature Fusion
Decision-Level Fusion
Feature-Level Fusion
Hybrid-Level Fusion
F_{Mul}
Classifier
FC layer
LightGBM
Vulnerability Label

Figure 2 (Color online) Overall framework of Agent4Vul.

Algorithm 1 Pipeline of the proposed Agent4Vul.

```
Input: The source code of the smart contracts;
Output: The vulnerability labels of the smart contracts; # The semantic branch;
1: (source code, comment) ← Commentator(source code);
2: ($F_S$, $F_C$) ← Vectorizer(source code, comment); # The graph branch;
3: bytecode ← Compiler(source code);
4: $(V, E, X)_{\rm cfg}$ ← BytecodeExtractor(bytecode); # $(V, E, X)_{\rm cfg}$ represent the node IDs, edges, and node attributes, respectively;
5: $h$ ← Vectorizer($X$);
6: $G_B$ ← GAT($V, E, h$); # Feature fusion and vulnerability detection;
7: $F_{\rm Mul}$ = Fusion($F_S, F_C, G_B$);
8: label = Classifier($F_{\rm Mul}$);
```

Comments. Descriptive comments generated by the Commentator agent, which not only explain the code's functionality but also reason about potential vulnerabilities, setting them apart from conventional code comments.

Bytecode CFG. A graph representation of the contract's bytecode [28], which highlights the control flow structure of the contract at the bytecode level.

The Agent4Vul framework integrates three modalities into a comprehensive smart contract vulnerability detection approach. It employs two LLM-based agents, each focusing on different contract aspects.

• **Commentator agent: comment generation.** The Commentator agent generates detailed, actionable comments that not only explain the code's logic but also reason about potential vulnerabilities. It uses a diverse prompt strategy library to guide the LLM in producing insightful comments tailored to vulnerability detection.

• **Vectorizer agent: vectorization.** The Vectorizer agent converts multimodal information (source code, comments, and bytecode) into high-dimensional vectors, capturing critical semantic features. Using the CoT reasoning approach, it processes the input step by step, ensuring meaningful feature representations for effective vulnerability detection.

The Agent4Vul framework is structured into two main branches, each focusing on distinct modalities of smart contract data.

(1) Semantic branch. The primary task of the semantic branch is to process both the source code and the comments to extract semantic features that are critical for vulnerability detection. This branch combines the output of the Commentator agent with the source code, using the Vectorizer agent to generate feature vectors that capture the underlying semantic patterns indicative of vulnerabilities.

(2) Graph branch. The graph branch processes the bytecode CFG of the smart contract. By embedding the control flow structure into high-dimensional vectors, this branch aims to capture the graph-based features that reflect the execution paths and potential vulnerabilities at the bytecode level. The Vectorizer agent plays a pivotal role in this process, transforming the CFG data into meaningful representations that are subsequently analyzed to detect vulnerabilities in the contract's structure.

(3) Feature fusion and vulnerability detection. Together, these two branches work to provide a robust framework for smart contract vulnerability detection, combining the insights from high-level code and comments with low-level graph-based analysis of bytecode.

3.2 Semantic branch

To extract the semantic features of smart contracts, we develop the semantic branch, as illustrated within the blue-bordered area in Figure 2. The commentator agent serves as the cornerstone of this branch, generating detailed comments for the contract code and structuring the smart contract's source code and comment information for further analysis. To achieve this, we design an FPSL for the Commentator agent, as shown in Figure 3. This library includes five carefully crafted prompt strategies to guide the LLM in reasoning through the code and generating high-quality comments. These strategies are outlined below.

Prompt strategy 1: simple description. This strategy provides a straightforward approach to comment generation, instructing the LLM to annotate each line of code with simple and concise descriptions of its functionality. This serves as a baseline for generating human-readable explanations.

Prompt strategy 2: detailed description. This strategy emphasizes a more granular analysis, requiring the LLM to highlight variable usage, function definitions, and specific actions performed within the code. The comments are detailed and aligned precisely with the corresponding lines, maintaining the integrity of the source code structure.

Prompt strategy 3: role-playing. This strategy assigns the LLM a role with specific goals and constraints, encouraging it to behave like a smart contract expert. By simulating a professional commentator's reasoning workflow, the LLM generates comments that are both technically accurate and contextually relevant.

Prompt strategy 4: CoT. This strategy employs the CoT paradigm, instructing the LLM to demonstrate its reasoning process while generating comments. By systematically analyzing, reasoning, and verifying, the agent produces comments that not only describe the functionality but also provide insights into the code's logic and purpose.

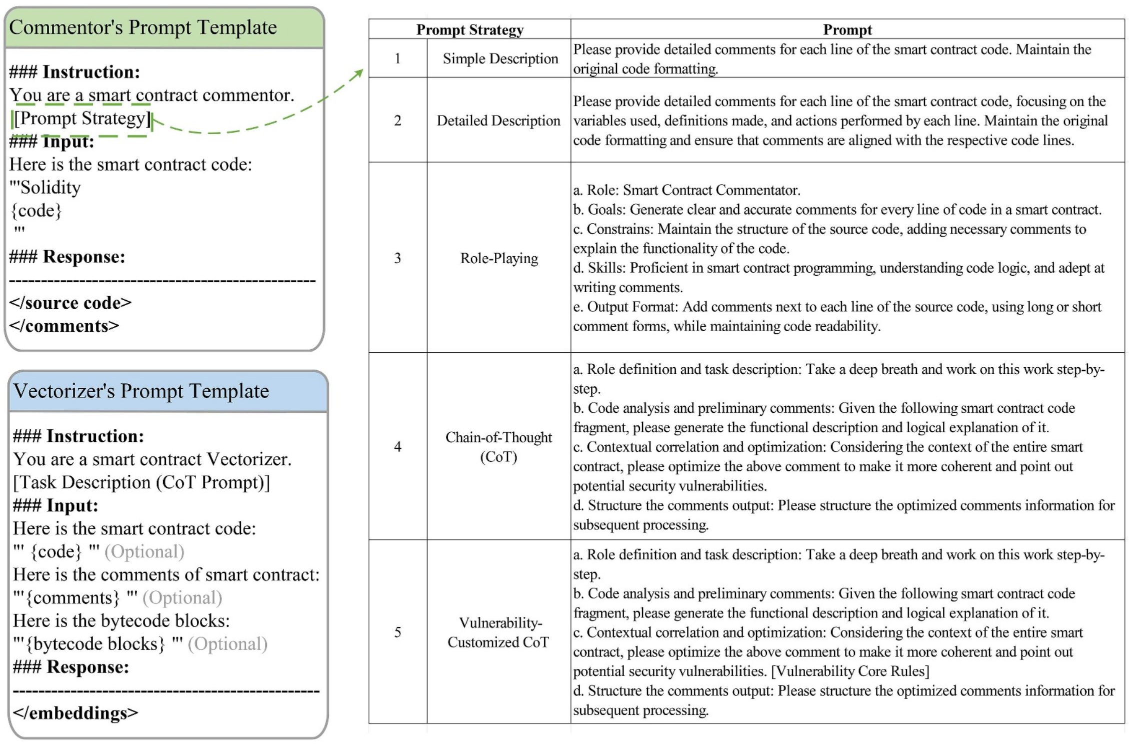
Prompt strategy 5: vulnerability-customized CoT. Inspired by [29], this strategy extends the CoT paradigm by incorporating a focus on specific vulnerabilities. For instance, when detecting reentrancy attacks, the prompt highlights critical elements such as balance variables and call.value usage. Users can further customize the CoT prompt based on specific rules or vulnerabilities of interest, tailoring the agent's output to meet detection needs.

The five prompt-based strategies in FPSL follow a systematic approach, aiming to strike a balance between interpretability, granularity, context relevance, and vulnerability-specific reasoning. Strategies 1 and 2 ensure comment structure through a layered refinement description mechanism. Strategy 3 introduces role constraints to enhance contextual modeling capabilities. Strategies 4 and 5 develop reasoning chains and focus on vulnerability features. The FPSL is designed to progressively enhance the capability of semantic information extraction, from fundamental code understanding to vulnerability-aware reasoning, while maintaining methodological adaptability. Through the FPSL, the Commentator agent effectively reasons about the smart contract, transforming the LLM's code comprehension capabilities into vulnerability-focused insights. This enables the agent to semantically capture features indicative of vulnerabilities, laying the foundation for enhanced detection performance within the semantic branch.

To effectively transform source code and comment text into high-quality vector representations for subsequent vulnerability detection, the Vectorizer agent adopts a CoT prompting paradigm, enabling a step-by-step reasoning process. As shown in Figure 4, the CoT prompts are specifically designed to guide the LLM in generating semantically rich vector representations from source code and associated comments. The process involves the following steps.

(1) Role definition and task description. The Vectorizer agent is defined as an entity dedicated to semantic embedding, tasked with converting smart contract source code and comment text into high-dimensional vector representations. These vectors are required to capture both the semantic and syntactic features of the input text, while also reflecting critical information relevant to vulnerability detection.

(2) Text preprocessing. Before the vectorization step, the source code and comment text undergo a preprocessing phase to standardize the input format and eliminate irrelevant or noisy information that may affect embedding quality. This preprocessing includes tokenization, stop word removal, and stemming. Tokenization splits the text into meaningful units (e.g., subwords or tokens), ensuring the text is standardized for embedding tasks. Through this systematic preprocessing, the Vectorizer agent ensures that the input text is optimized for subsequent embedding tasks.



Commentor's Prompt Template

### Instruction:
You are a smart contract commentor.
[Prompt Strategy]
### Input:
Here is the smart contract code:
'''Solidity
{code}
'''
### Response:

</source code>
</comments>

Vectorizer's Prompt Template

### Instruction:
You are a smart contract Vectorizer.
[Task Description (CoT Prompt)]
### Input:
Here is the smart contract code:
''' {code} ''' (Optional)
Here is the comments of smart contract:
'''{comments} ''' (Optional)
Here is the bytecode blocks:
'''{bytecode blocks} ''' (Optional)
### Response:

</embeddings>

Prompt Strategy		**Prompt**
1	Simple Description	Please provide detailed comments for each line of the smart contract code. Maintain the original code formatting.
2	Detailed Description	Please provide detailed comments for each line of the smart contract code, focusing on the variables used, definitions made, and actions performed by each line. Maintain the original code formatting and ensure that comments are aligned with the respective code lines.
3	Role-Playing	a. Role: Smart Contract Commentator. b. Goals: Generate clear and accurate comments for every line of code in a smart contract. c. Constrains: Maintain the structure of the source code, adding necessary comments to explain the functionality of the code. d. Skills: Proficient in smart contract programming, understanding code logic, and adept at writing comments. e. Output Format: Add comments next to each line of the source code, using long or short comment forms, while maintaining code readability.
4	Chain-of-Thought (CoT)	a. Role definition and task description: Take a deep breath and work on this work step-by-step. b. Code analysis and preliminary comments: Given the following smart contract code fragment, please generate the functional description and logical explanation of it. c. Contextual correlation and optimization: Considering the context of the entire smart contract, please optimize the above comment to make it more coherent and point out potential security vulnerabilities. d. Structure the comments output: Please structure the optimized comments information for subsequent processing.
5	Vulnerability-Customized CoT	a. Role definition and task description: Take a deep breath and work on this work step-by-step. b. Code analysis and preliminary comments: Given the following smart contract code fragment, please generate the functional description and logical explanation of it. c. Contextual correlation and optimization: Considering the context of the entire smart contract, please optimize the above comment to make it more coherent and point out potential security vulnerabilities. [Vulnerability Core Rules] d. Structure the comments output: Please structure the optimized comments information for subsequent processing.

Figure 3 (Color online) Paradigm of FPSL.

(3) Vectorization conversion. After preprocessing, the Vectorizer agent employs a pre-trained LLM specifically designed for embedding tasks to convert the processed source code and comment text into high-dimensional vector representations. Each token is passed through the LLM's embedding layer and Transformer architecture to generate context-aware embeddings. These embeddings are aggregated to produce two vector representations: F_S for the semantic features of the source code, and F_C for the auxiliary explanatory features from the comment text.

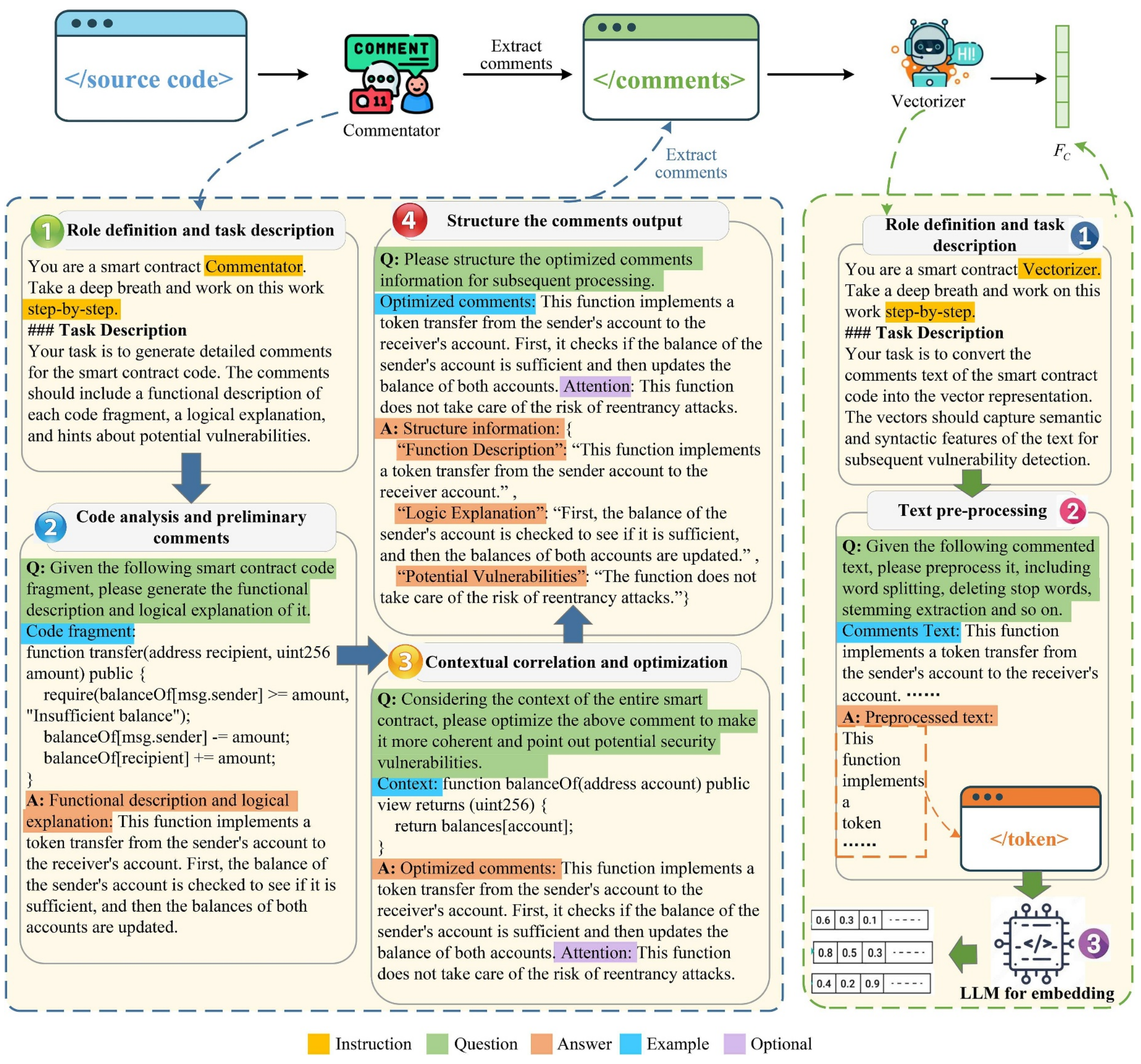
By adopting this step-by-step reasoning approach, the Vectorizer agent ensures the effective transformation of multimodal textual information into semantically enriched vector representations. This CoT-based vectorization process enhances the agent's ability to understand the input text while ensuring the generated embeddings comprehensively capture vulnerability-relevant features. This serves as a crucial foundation for the semantic branch, supporting high-precision vulnerability detection.

3.3 Graph branch

In the Agent4Vul framework, the graph branch is designed to capture the structural characteristics of the bytecode CFG for vulnerability detection. To construct the CFG, we utilize the BinaryCFGExtractor tool [21], which extracts the control flow graph from the compiled bytecode. The CFG is represented as a graph (V, E, X), where V denotes the set of nodes (representing bytecode blocks), E denotes the set of edges (representing control flow between blocks), and X denotes the node attributes (capturing block-level information).

Once the CFG is constructed, the Vectorizer agent is employed to encode the bytecode blocks. The encoding process mirrors the steps outlined in the semantic branch, enabling the agent to capture both instruction-level and block-level information. This ensures a comprehensive representation of the semantic and structural features embedded in the bytecode.

After the initial encoding of the bytecode blocks, the GAT is applied to learn the graph-structured embeddings of the CFG. The GAT employs attention mechanisms to aggregate information from neighboring nodes, allowing it to model the interactions between bytecode blocks and the overall graph topology effectively. Specifically, the GAT computes the node embeddings by iteratively updating each node's representation using weighted contributions from its neighbors, defined by attention scores. This process enhances the model's ability to capture both local and global structural patterns within the CFG. α_{ij}



</source code>
Commentator
Extract comments
</comments>
Extract comments
Vectorizer
F_c
1 Role definition and task description
2 Code analysis and preliminary comments
3 Contextual correlation and optimization
4 Structure the comments output
1 Role definition and task description
2 Text pre-processing
</token>
3 LLM for embedding
0.6 0.3 0.1 -----
0.8 0.5 0.3 -----
0.4 0.2 0.9 -----
🟨 Instruction 🟩 Question 🟧 Answer 🟦 Example 🟪 Optional

You are a smart contract Commentator. Take a deep breath and work on this work step-by-step.
Task Description
Your task is to generate detailed comments for the smart contract code. The comments should include a functional description of each code fragment, a logical explanation, and hints about potential vulnerabilities.

Q: Given the following smart contract code fragment, please generate the functional description and logical explanation of it.
Code fragment:

```
function transfer(address recipient, uint256 amount) public {
    require(balanceOf[msg.sender] >= amount, "Insufficient balance");
    balanceOf[msg.sender] -= amount;
    balanceOf[recipient] += amount;
}
```

A: Functional description and logical explanation: This function implements a token transfer from the sender's account to the receiver's account. First, the balance of the sender's account is checked to see if it is sufficient, and then the balances of both accounts are updated.

Q: Considering the context of the entire smart contract, please optimize the above comment to make it more coherent and point out potential security vulnerabilities.
Context:

```
function balanceOf(address account) public view returns (uint256) {
    return balances[account];
}
```

A: Optimized comments: This function implements a token transfer from the sender's account to the receiver's account. First, it checks if the balance of the sender's account is sufficient and then updates the balance of both accounts. Attention: This function does not take care of the risk of reentrancy attacks.

Q: Please structure the optimized comments information for subsequent processing.
Optimized comments: This function implements a token transfer from the sender's account to the receiver's account. First, it checks if the balance of the sender's account is sufficient and then updates the balance of both accounts. Attention: This function does not take care of the risk of reentrancy attacks.
A: Structure information: {
"Function Description": "This function implements a token transfer from the sender account to the receiver account." ,
"Logic Explanation": "First, the balance of the sender's account is checked to see if it is sufficient, and then the balances of both accounts are updated." ,
"Potential Vulnerabilities": "The function does not take care of the risk of reentrancy attacks."}

You are a smart contract Vectorizer. Take a deep breath and work on this work step-by-step.
Task Description
Your task is to convert the comments text of the smart contract code into the vector representation. The vectors should capture semantic and syntactic features of the text for subsequent vulnerability detection.

Q: Given the following commented text, please preprocess it, including word splitting, deleting stop words, stemming extraction and so on.
Comments Text: This function implements a token transfer from the sender's account to the receiver's account. ······
A: Preprocessed text:
This
function
implements
a
token
······

Figure 4 (Color online) CoT strategy detail description in the semantic branch.

represents the attention coefficient that is given by

$$\alpha_{ij} = \frac{\exp(\mathcal{T}(a^{\mathrm{T}}[Wh_i \oplus Wh_j]))}{\sum_{k,E_{ik}\neq 0}\exp(\mathcal{T}(a^{\mathrm{T}}[Wh_i \oplus Wh_k]))}, \tag{1}$$

where $\oplus$ denotes concatenation, a is the weight of a single-layer MLP, and $\mathcal{T}$ is the Leaky ReLU function. GAT computes the hidden states of every node by attending to its neighbors as

$$h_i' = \sigma\left(\sum_{j,E_{ij}\neq 0}\alpha_{ij}Wh_j\right), \tag{2}$$

where σ is a nonlinear activation function, $E_{ij} \neq 0$ represents the neighbor relationship between nodes i and j in the graph, and W is a weight matrix. The graph semantic embeddings of the bytecode G_B is computed as

$$G_B = \sum_{i=1}^{V}\sigma\left(P_{\text{gate}}(M_1h_i' + b_1)\right) \odot P(M_2h_i' + b_2), \tag{3}$$

where $\odot$ denotes the element-wise product and σ is an activation function. Matrix M_j and bias vector b_j, with subscript $j \in \{1,2\}$, are trainable network parameters. $|V|$ represents the number of nodes and P is a multi-layer perception.

The output of the graph branch is a set of graph-structured embeddings G_B that encode both the semantic attributes of individual bytecode blocks and the structural dependencies between them. These embeddings are subsequently used in the downstream classification task to identify vulnerabilities related to control flow anomalies and structural weaknesses within the smart contract's bytecode.

3.4 Feature fusion and vulnerability detection

After processing through the semantic branch and graph branch, the Agent4Vul framework extracts three key feature sets: source code features F_S, comment features F_C, and bytecode CFG features G_B. These features capture complementary information about the smart contract, covering semantic, contextual, and structural characteristics. To integrate these multimodal features into a unified representation F_{Mul}, we apply one of three feature fusion strategies depending on the detection scenario.

- **Decision-level fusion [30].** Each feature set (F_S, F_C, G_B) is processed independently through separate modality-specific classifiers. These classifiers generate preliminary vulnerability predictions based on individual modalities. The final decision is made by aggregating their outputs, ensuring that unique modality-specific characteristics are preserved while leveraging complementary information.
- **Feature-level fusion [31].** The raw feature sets F_S, F_C, and G_B are concatenated to form a single feature vector, allowing the classifier to learn cross-modal relationships directly from the combined input.
- **Hybrid-level fusion [32].** A combination of decision-level and feature-level approaches, where predictions from modality-specific classifiers are integrated with the fused feature representation to capture both individual and cross-modal interactions.

The fused feature representation F_{Mul}, derived from the chosen fusion strategy, is then passed into a classification model for final vulnerability detection. We employ either a LightGBM model [33], which excels at tabular data processing, or a fully connected layer (FC) [34] to handle deep multimodal relationships. During inference, the trained model analyzes the fused features to predict vulnerabilities effectively. By leveraging tailored fusion strategies, the Agent4Vul framework achieves robust and accurate detection across a variety of smart contract vulnerabilities.

4 Evaluation

In this section, we present an extensive experimental evaluation of the Agent4Vul framework, addressing the following research questions (RQs) to assess its effectiveness and performance.

- RQ1: How does the performance of Agent4Vul compare with the current state-of-the-art smart contract vulnerability detection methods? See Subsection 4.2.
- RQ2: How do the individual modules of Agent4Vul contribute to its overall performance? See Subsection 4.3.
- RQ3: What is the impact of different LLM choices on the framework's performance? See Subsection 4.4.
- RQ4: How do prompt strategies from FPSL influence the performance of the framework? See Subsection 4.5.
- RQ5: What effect do different feature fusion strategies have on the framework's performance? See Subsection 4.6.

4.1 Experiment setup

Dataset. We select the largest publicly available smart contract vulnerability dataset [21], comprising 40000 real-world smart contracts with detailed comments for four types of vulnerabilities. Among these contracts, 4290 are identified to have vulnerabilities: 136 with Delegatecall, 1368 with Integer Overflow/Underflow, 680 with Reentrancy, and 2242 with Timestamp dependency vulnerabilities. For evaluation, the dataset is randomly split into 80% for training and 20% for testing. Each experiment was repeated five times, with the average results reported.

Implementations. The system consists of three main components: (1) the semantic branch, (2) the graph branch, and (3) feature fusion and vulnerability detection. Specifically, the Commentator agent utilizes the GLM-4 [35] and Qwen [36] models, which possess advanced natural language understanding and reasoning capabilities, enabling the generation of detailed and accurate code comments. The Vectorizer agent relies on the Voyage-lite-02-instruct [37] and GritLM-7B [38] models, which are specifically

Table 1 LightGBM model hyperparameter tuning.

Hyperparameter	Tuning range	Optimal value
`max_bin`	range(5, 256, 10)	215
`max_depth`	range(3, 8, 1)	5
`num_leaves`	range(5, 100, 5)	10
`bagging_freq`	range(0, 81, 10)	0
`min_data_in_leaf`	range(1, 102, 10)	31
`feature_fraction`	[0.6, 0.7, 0.8, 0.9, 1.0]	0.8
`bagging_fraction`	[0.6, 0.7, 0.8, 0.9, 1.0]	0.6
`lambda_l1`	[1e−5, 1e−3, 1e−1, 0.0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0]	1e−5
`lambda_l2`	[1e−5, 1e−3, 1e−1, 0.0, 0.1, 0.3, 0.5, 0.7, 0.9, 1.0]	0.0
`min_split_gain`	[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]	0.0

Table 2 Performance comparison (%) of Accuracy (ACC), Recall (RE), Precision (PRE) and F1 score (F1). Twenty methods are included in the comparison. 'n/a' indicates that the corresponding tool does not support the detection of the vulnerability type. The best results for each type are in bold.

	Method	Delegatecall				Overflow/Underflow				Reentrancy				Timestamp			
		ACC	RE	PRE	F1	ACC	RE	PRE	F1	ACC	RE	PRE	F1	ACC	RE	PRE	F1
1	sFuzz [9]	64.37	47.22	58.62	52.31	45.50	25.97	25.88	25.92	55.69	14.95	10.88	12.59	33.41	27.01	23.15	24.93
2	Smartcheck [10]	62.41	56.21	45.56	50.33	53.91	68.54	42.81	52.70	54.65	16.34	45.71	24.07	47.73	79.34	47.89	59.73
3	Osiris [11]	n/a	n/a	n/a	n/a	68.41	34.18	60.83	43.77	56.73	63.88	40.94	49.90	66.83	55.42	59.26	57.28
4	Oyente [12]	n/a	n/a	n/a	n/a	69.71	57.55	58.05	57.80	65.07	63.02	46.56	53.55	68.29	57.97	61.04	59.47
5	Mythril [13]	75.06	62.07	72.30	66.80	n/a	n/a	n/a	n/a	64.27	75.51	42.86	54.68	62.40	49.80	57.50	53.37
6	Securify [14]	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	72.89	73.06	68.40	70.41	n/a	n/a	n/a	n/a
7	Slither [15]	68.97	52.27	70.12	59.89	n/a	n/a	n/a	n/a	74.02	73.50	74.44	73.97	68.52	67.17	69.27	68.20
8	Vanilla-RNN [16]	64.33	67.26	63.77	65.47	68.12	70.19	67.00	68.56	65.90	72.89	67.39	70.03	64.41	65.17	64.16	64.66
9	ReChecker [17]	67.98	70.66	66.47	68.50	70.49	71.59	70.56	71.07	70.95	72.92	70.15	71.51	66.65	54.53	73.37	62.56
10	GCN [18]	65.76	69.74	69.01	69.37	67.53	70.93	69.52	70.22	73.21	73.18	74.47	73.82	75.91	77.55	74.93	76.22
11	TMP [19]	69.11	70.37	68.18	69.26	70.85	69.47	70.26	69.86	76.45	75.30	76.04	75.67	78.84	76.09	78.68	77.36
12	AME [20]	72.85	69.40	70.25	69.82	73.24	71.59	71.36	71.47	81.06	78.45	79.62	79.03	82.25	80.26	81.42	80.84
13	SMS [21]	78.82	73.69	76.97	75.29	79.36	72.98	78.14	75.47	83.85	77.48	79.46	78.46	89.77	91.09	89.15	90.11
14	DMT [21]	82.76	77.93	84.61	81.13	85.64	74.32	85.44	79.49	89.42	81.06	83.62	82.32	94.58	96.39	93.60	94.97
15	GPTLENS [25]	76.32	76.32	84.11	77.69	74.36	74.36	69.09	67.37	48.15	48.15	70.69	47.29	57.89	57.89	64.94	49.96
16	GPT-4o	65.00	65.00	84.60	66.24	85.45	85.45	89.17	86.26	29.09	29.09	8.46	13.11	61.43	61.43	78.50	55.34
17	o1	70.00	70.00	85.65	71.38	94.50	94.55	94.48	94.47	78.18	78.18	87.53	79.17	62.86	62.86	62.86	62.86
18	DeepSeek-V3	62.50	62.50	84.13	63.56	83.64	83.64	82.73	82.85	29.09	29.09	8.46	13.11	54.29	54.29	58.21	49.82
19	DeepSeek-R1	67.50	67.50	80.97	69.11	94.55	94.55	94.48	94.47	76.36	76.36	86.96	77.41	64.29	64.29	64.33	64.29
20	Agent4Vul	**97.50**	**97.50**	**97.58**	**97.45**	**98.18**	**98.18**	**98.29**	**98.20**	**92.73**	**92.73**	**92.68**	**92.53**	**98.57**	**98.57**	**98.62**	**98.58**

designed for feature embedding to capture the semantic information of the code. The feature classification model FC and LightGBM [33] are implemented in Python, based on the PyTorch framework.

Parameter settings. All experiments are conducted on a computer equipped with an Intel Core i9 CPU operating at 3200 MHz, an NVIDIA GeForce RTX 4090 GPU, and 64 GB of RAM. The software environment utilizes Ubuntu 20.04 LTS as the operating system. For the FC layer networks, the learning rate is adjusted within the range of {0.5, 0.1, 0.05, 0.01, 0.005, 0.001}. Regarding the hyperparameters of the LightGBM model, we employ a step-by-step tuning method for optimization. The adjustment range and the ultimately confirmed optimal parameter combination are presented in Table 1.

4.2 RQ1: effectiveness of Agent4Vul

Table 2 presents the performance evaluation of 20 smart contract vulnerability detection methods, focusing on four distinct types of vulnerabilities.

Comparison with rule-based techniques. We initially compare Agent4Vul with seven rule-based techniques: sFuzz [9], Smartcheck [10], Osiris [11], Oyente [12], Mythril [13], Securify [14], and Slither [15] (rows 1–7 in Table 2). The results demonstrate that Agent4Vul significantly outperforms all existing rule-based tools across all four vulnerability types. Specifically, Agent4Vul surpasses Mythril by 30.65% for Delegatecall, and outperforms Oyente by 40.40% for Overflow/Underflow. For Reentrancy and Times-

Table 3 Ablation experiments for agents. The impact of the designed Commentator and Vectorizer modules on the vulnerability detection performance (%) of Agent4Vul.

Agent	Delegatecall			overflow/underflow			Reentrancy			Timestamp		
	ACC	F1	ACC-Decrease	ACC	F1	ACC-Decrease	ACC	F1	ACC-Decrease	ACC	F1	ACC-Decrease
Agent4Vul (semantic)	92.50	92.34	–	96.36	96.36	–	89.09	89.32	–	84.29	84.44	–
Commentator-w/o	90.00	89.53	−2.50	78.18	76.44	−18.18	80.00	74.71	−9.09	80.00	80.20	−4.29
Vectorizer-w/o	72.50	73.77	−20.00	69.09	67.16	−27.27	87.27	86.26	−1.82	75.71	75.94	−8.58
Two agents-w/o	70.00	71.00	−22.50	63.64	61.93	−32.72	81.82	80.73	−7.27	72.86	73.11	−11.43

tamp, Agent4Vul achieves F1 scores of 92.53% and 98.58%, respectively, significantly improving over Slither by 18.56% and 30.38%.

Comparison with deep learning-based methods. Next, we compare Agent4Vul with seven state-of-the-art deep learning-based methods: Vanilla-RNN [16], ReChecker [17], GCN [18], TMP [19], AME [20], SMS [21], and DMT [21] (rows 8–14 in Table 2). The results indicate that Agent4Vul significantly outperforms all deep learning-based methods across the four vulnerability types. The best-performing deep learning-based method, DMT, achieves F1 scores of 81.13%, 79.49%, 82.32%, and 94.97% for the four vulnerability types, respectively. Agent4Vul surpasses DMT in all four categories, with F1 score improvements of 16.32%, 18.71%, 10.21%, and 3.61%, respectively.

Comparison with LLM methods. We benchmark Agent4Vul against the latest LLMs, GPT-4o[1], o1[2], DeepSeek-V3[3], and DeepSeek-R1[4]. Although these LLMs perform better on Overflow/Underflow vulnerabilities, they exhibit relatively weaker performance on the other three types of vulnerabilities. Agent4Vul achieves F1 score improvements of 26.07%, 3.73%, 13.36%, and 34.29% over the LLMs for the four vulnerability types, respectively. GPTLens [25] outperforms LLMs in Delegatecall vulnerability detection but struggles with other types. This highlights LLMs' cognitive limits in smart contract auditing, making the direct application of LLMs for smart contract auditing somewhat restrictive. In contrast, Agent4Vul leverages multimodal LLM agents, effectively utilizing the reasoning and semantic understanding capabilities of LLMs to transform code comprehension into vulnerability detection.

Analysis of Agent4Vul's superiority. The superior performance of Agent4Vul can be attributed to several key factors. First, the integration of reasoning enhances vulnerability detection by enabling the Commentator agent to generate detailed code comments, while the Vectorizer agent effectively extracts meaningful features. Second, the use of multimodal learning enables the effective combination of diverse data sources, such as source code, comments, and CFGs, thereby providing a comprehensive representation of the contract's behavior. Lastly, the adoption of advanced models, such as LightGBM, ensures robust classification capabilities, leading to more accurate vulnerability detection.

Answer to RQ1. The proposed Agent4Vul outperforms state-of-the-art methods across all four types of vulnerabilities, achieving F1 score improvements ranging from 3.61% to 16.32% compared to the best existing methods.

4.3 RQ2: impact of different modules

To answer RQ2, we conduct a comprehensive ablation study to evaluate the impact of different modules on the overall performance of Agent4Vul. In Section 3, we described the design of the Agent4Vul agents and the multimodal framework. Accordingly, we design separate ablation tests for the agents and the multimodal features. The results of all ablation experiments are presented in Tables 3 and 4.

Ablation test for dual agents. In Agent4Vul, we introduced two LLM-based agents: Commentator and Vectorizer. To evaluate the effectiveness of these agents, we modified the semantic branch of the framework and designed the following ablation tests. (1) Removing the Commentator agent: The smart contract source code bypasses Commentator for comments generation and is directly encoded by Vectorizer to obtain embedded feature vectors for vulnerability detection. (2) Removing the Vectorizer agent: The smart contract source code is commented by the Commentator, but the Vectorizer is bypassed. Instead, word2vec [39] is used to generate the embedded feature vectors for vulnerability detection. (3) Removing both agents: Both the Commentator and Vectorizer are bypassed, and the smart contract source code uses word2vec to directly generate the embedded feature vectors for vulnerability detection.

1) https://openai.com/index/hello-gpt-4o/.
2) https://openai.com/index/introducing-openai-o1-preview/.
3) https://github.com/deepseek-ai/DeepSeek-V3.
4) https://github.com/deepseek-ai/DeepSeek-R1.

Table 4 Ablation experiments for the multimodal features. The impact of source code modality F_S, comment modality F_C, and bytecode CFG modality G_B on the vulnerability detection performance (%) of Agent4Vul.

modality	Delegatecall			Overflow/Underflow			Reentrancy			Timestamp		
	ACC	F1	ACC-Decrease	ACC	F1	ACC-Decrease	ACC	F1	ACC-Decrease	ACC	F1	ACC-Decrease
Agent4Vul	97.50	97.45	–	98.18	98.20	–	92.73	92.53	–	98.57	98.58	–
F_S-w/o	92.50	92.34	−5.00	98.18	98.20	−0.00%	89.09	88.80	−3.64	97.14	97.16	−1.43
F_C-w/o	90.00	89.53	−7.50	83.64	83.09	−14.54	87.27	86.26	−5.46	95.71	95.74	−2.86
G_B-w/o	85.00	85.50	−12.50	96.36	96.36	−1.82	89.09	89.32	−3.64	84.29	84.44	−14.28

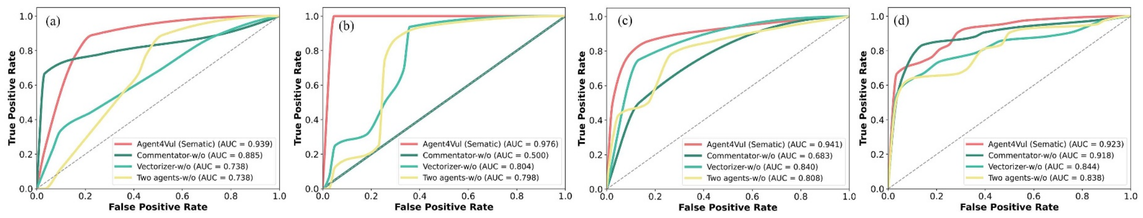


Figure 5 (Color online) Ablation experiment AUC curves. Impact of Commentator and Vectorizer on Agent4Vul's vulnerability detection performance. (a) Delegatecall; (b) Overflow/Underflow; (c) Reentrancy; (d) Timestamp.

The experimental results are presented in Table 3, with the AUC curve shown in Figure 5. Notably, the absence of the Commentator agent leads to a decrease in detection accuracy across all four vulnerability types, with the most significant drop observed in Overflow/Underflow, which decreases by 18.18%. This suggests that the Commentator agent plays a crucial role in translating the understanding of smart contract code into vulnerability detection capability, thereby significantly improving performance. Furthermore, in the absence of the Vectorizer agent, the accuracy decreases by 1.82% to 27.27%. Compared to word2vec, the Vectorizer agent, which leverages pre-trained large language models for embedding tasks, achieves more sophisticated and accurate feature representations, leading to enhanced detection performance. Finally, when both agents are removed, the accuracy for all four vulnerability types further declines. The AUC curve further demonstrates that Agent4Vul exhibits superior classification performance, reinforcing the effectiveness of both the Commentator and Vectorizer agents in vulnerability detection.

Ablation test for multimodal features. In Agent4Vul, the smart contract modalities include the source code modality F_S, the comment modality F_C, and the bytecode CFG modality G_B. To assess the effectiveness of multimodal features, we design ablation tests where one modality is removed at a time (i.e., F_S, F_C, or G_B). The results are shown in Table 4 and Figure 6.

After removing the source code modality F_S, the accuracy decreases by 5.00%, 3.64%, and 1.43% for Delegatecall, Reentrancy, and Timestamp, respectively. For Overflow/Underflow, the accuracy and F1 score remain unchanged, but the AUC score drops by 0.006. This is because, for Overflow/Underflow, the feature quality extracted by F_C and G_B is exceptionally high. The most significant drop is observed when removing F_C, which leads to a 14.54% decrease in accuracy, indicating that F_C plays a crucial role in extracting key features for this vulnerability type, while F_S contributes minimal additional information. After removing G_B, the accuracy of Delegatecall and Timestamp decreases most significantly, by 12.50% and 14.28%, respectively. Therefore, the experimental results demonstrate that each modality contributes significantly to the performance of Agent4Vul, and the absence of any modality leads to a notable decline in overall performance.

Answer to RQ2. Every module in Agent4Vul is indispensable. Both the Commentator and Vectorizer agents, as well as the multimodal features, are crucial for the performance of Agent4Vul.

4.4 RQ3: the impact of different LLMs on Agent4Vul

To investigate the impact of different LLMs on the performance of Agent4Vul, we conduct experiments using two sets of LLMs. Specifically, the Commentator agent utilizes the GLM-4 and Qwen models to implement the CoT prompts designed in our FPSL and generate comments for smart contracts. The Vectorizer agent uses the Voyage-lite-02-instruct and GritLM-7B models to convert the comments and source code into embedding vector representations. The experimental results are shown in Figure 7.

Commentator agent: GLM-4 vs. Qwen. Qwen outperforms GLM-4 in Delegatecall, achieving

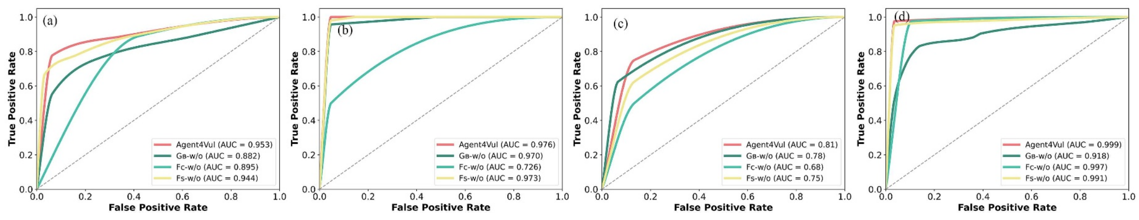


Figure 6 (Color online) Ablation experiment AUC curves. Impact of source code modality F_S, comment modality F_C, and bytecode CFG modality G_B modalities on Agent4Vul's vulnerability detection performance. (a) Delegatecall; (b) Overflow/Underflow; (c) Reentrancy; (d) Timestamp.

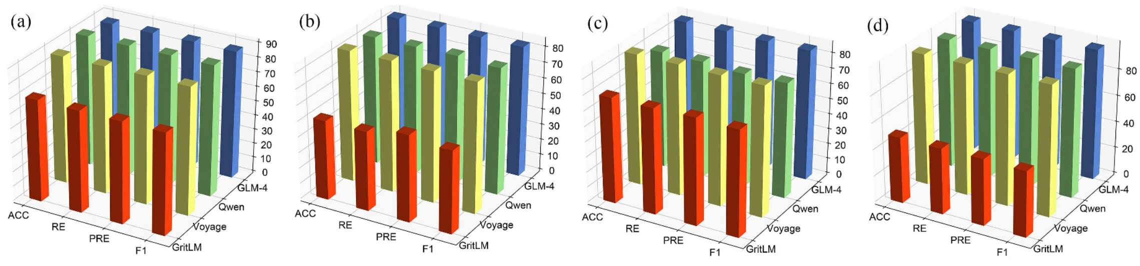


Figure 7 (Color online) Performance (%) comparison of different LLMs in Agent4Vul. (a) Delegatecall; (b) Overflow/Underflow; (c) Reentrancy; (d) Timestamp.

an F1 score of 2.34% higher. Conversely, GLM-4 shows a slight advantage in Overflow/Underflow and Timestamp, with F1 scores higher by 2.45% and 1.42%, respectively. In Reentrancy, GLM-4 significantly surpasses Qwen with an F1 score difference of 9.71%. The experimental results demonstrate that GLM-4 possesses superior capabilities in understanding and reasoning about smart contracts compared to Qwen. GLM-4 generates comments that are more precise in terms of functional descriptions, logical explanations, and potential vulnerabilities. This advantage likely arises from GLM-4's enhanced ability to manage complex logic and conditional judgments, leading to better performance in detecting vulnerabilities involving state transitions and conditional jumps. In contrast, Qwen excels in handling straightforward text features but falls short in complex logical reasoning tasks.

Vectorizer agent: Voyage-lite-02-instruct vs GritLM-7B. Voyage-lite-02-instruct significantly outperforms GritLM-7B in terms of Accuracy, Recall, Precision, and F1 scores across all four vulnerability types. Specifically, Voyage-lite-02-instruct shows improvements in F1 scores by 17.66% for Delegatecall, 30.23% for Overflow/Underflow, 15.54% for Reentrancy, and 48.63% for Timestamp. The experimental results indicate that Voyage-lite-02-instruct is more adept at capturing comprehensive feature representations for smart contract vulnerability detection. Technically, Voyage-lite-02-instruct uses an embedding vector dimension of 1024, compared to GritLM-7B's 4096. Larger vector dimensions typically entail higher computational complexity and storage requirements, which can degrade performance in practical applications. The smaller vector dimension of Voyage-lite-02-instruct enables it to efficiently capture critical features in smart contracts while maintaining lower computational overhead and storage needs. This efficiency likely contributes to its superior performance over GritLM-7B in vulnerability detection.

Answer to RQ3. The selection of different LLMs has a substantial impact on the performance of the Agent4Vul framework. Our evaluations demonstrate that GLM-4 and Voyage-lite-02-instruct are the more effective models, significantly enhancing the performance of smart contract vulnerability detection.

4.5 RQ4: the impact of prompt strategies from FPSL

Based on the five prompt strategies designed in FPSL (Figure 3), we explore the impact of different prompt strategies on the detection performance across four contract vulnerability types. Among them, the vulnerability-customized CoT prompt integrates the vulnerability core rules for each of the four vulnerability types, as detailed below. (1) Delegatecall: Focusing on the usage of the Delegatecall function, checking the relationship between msg.sender and the contract owner before executing any delegated calls. (2) Overflow/Underflow: Focusing on how balance-related variables are defined and manipulated, as well as the arithmetic operations performed on them. (3) Reentrancy: Focusing on balance-related variables, functions, and the usage of the call.value function. (4) Timestamp: Paying special attention to the usage

Table 5 Performance comparison (%) of Accuracy (ACC), Recall (RE), Precision (PRE), and F1 score (F1) for different prompt strategies from FPSL on vulnerabilities. The best results are in bold.

Prompt strategy		Delegatecall				Overflow/Underflow				Reentrancy				Timestamp			
		ACC	RE	PRE	F1	ACC	RE	PRE	F1	ACC	RE	PRE	F1	ACC	RE	PRE	F1
1	Simple description	90.00	90.00	89.93	89.67	83.64	83.45	83.64	83.26	85.45	85.72	85.45	83.97	**88.57**	**89.12**	**88.57**	**88.55**
2	Detailed description	**90.00**	**90.00**	**90.88**	**90.20**	78.18	77.70	78.18	77.47	76.36	75.89	76.36	76.10	84.13	84.12	84.13	84.04
3	Role-playing	87.50	87.50	87.63	86.92	81.82	81.64	81.82	81.23	81.82	81.47	81.82	81.52	78.57	78.61	78.57	78.58
4	CoT	82.50	82.50	82.04	81.69	**87.27**	**87.95**	**87.27**	**86.72**	80.00	79.50	80.00	79.48	82.86	82.86	82.86	82.86
5	Vulnerability-customized CoT	87.50	87.50	87.63	86.92	78.18	78.84	78.18	76.36	**89.09**	**89.46**	**89.09**	**88.69**	85.71	86.86	85.71	85.64

of the block.timestamp variable to ensure its correct interpretation. In the experiment, we utilize the F_C feature from the semantic branch of Agent4Vul, combined with the FC classifier to classify the F_C feature. The experiment focuses solely on the effect of different prompt strategies on the generated comments. The experimental results are shown in Table 5.

Based on the experimental results, we found that the vulnerability-customized CoT strategy performs better in reasoning quality for Reentrancy vulnerabilities. This indicates that the designed vulnerability core rules play a positive role in the comment generation process, effectively enhancing the detection of Reentrancy vulnerabilities. However, the vulnerability-customized CoT strategy does not yield optimal reasoning results for Overflow/Underflow, Delegatecall, and Timestamp vulnerabilities. When the prompt strategy does not provide vulnerability core rules, the use of CoT strategies, simple descriptions, or detailed descriptions instead generates higher-quality comments. Through an in-depth analysis of the generated comments, we observe that in contracts without the corresponding vulnerabilities, the LLM tends to make preemptive judgments. For instance, in contracts without a Delegatecall vulnerability, the comment might mention, "If a delegatecall function is present, and msg.sender is not authorized, there will be a security risk," which impacts the quality of the comments and interferes with the vulnerability detection. In contrast, the use of prompt strategies that do not include the vulnerability core rules but are more generalized allows for a better restoration of the actual semantics of the code, leading to more accurate reasoning comments.

Answer to RQ4. The choice of prompt strategies from FPSL significantly affects the reasoning and comment generation quality of the Commentator across different vulnerability types. Therefore, the design of FPSL is essential.

4.6 RQ5: the impact of feature fusion strategies on Agent4Vul

We conducted an in-depth study on the impact of the three feature fusion strategies, decision-level, feature-level, and hybrid-level, proposed in Subsection 3.4 on the performance of Agent4Vul. The experimental results, shown in Figure 8, indicate that the feature-level fusion strategy outperforms both decision-level and hybrid-level fusion strategies in detecting Delegatecall and Timestamp vulnerabilities, with F1 scores of 97.45% and 98.58%, respectively. This suggests that performing fusion at the feature level allows for better capture of important information relevant to vulnerability detection, thereby enhancing the model's detection capability. On the other hand, the hybrid-level fusion strategy shows superior performance in detecting Overflow/Underflow and Reentrancy vulnerabilities, indicating that this strategy effectively combines its own features with the decision outputs from other modalities, leading to improved reasoning accuracy. This result demonstrates that different types of vulnerabilities require different fusion strategies to maximize the efficiency of feature information utilization.

Answer to RQ5. Different feature fusion strategies have a modest impact on the vulnerability detection performance of Agent4Vul. Therefore, it is necessary to select an appropriate feature fusion strategy.

5 Discussion

5.1 Selection of LLMs

In the Agent4Vul framework, we currently combine the best performance results of the four LLMs used in our evaluation. Since the choice of different LLMs can significantly affect the detection performance for

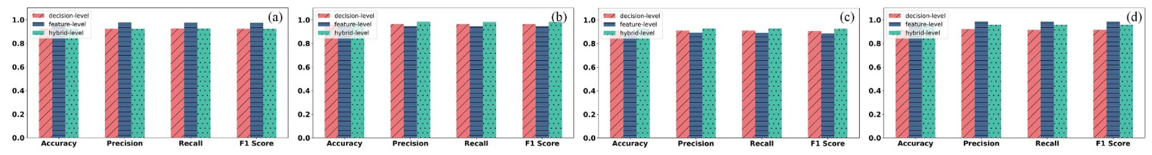


Figure 8 (Color online) Feature fusion strategies on vulnerability detection performance metrics of Agent4Vul. (a) Delegatecall; (b) Overflow/Underflow; (c) Reentrancy; (d) Timestamp.

various vulnerability types in smart contracts, our future work will aim to further exploit the potential of LLMs within the Commentator and Vectorizer agents. We plan to employ an ensemble learning approach that links multiple LLMs to enhance overall detection performance, such as weighted voting or stacking.

5.2 Zero-shot vs. fine-tuning

In the Agent4Vul framework, the LLMs are employed in a zero-shot manner without fine-tuning for smart contract vulnerability detection tasks. Our designed multimodal agents effectively transform the LLMs' code understanding capabilities into vulnerability detection capabilities, thereby enhancing detection performance. Compared to fine-tuning, the zero-shot approach is more cost-effective and conserves significant computational resources. While fine-tuning can yield good results [26], it is limited for LLMs with fewer parameters. Fine-tuning LLMs like GPT-4 may be effective but pose challenges due to high costs and computational demands.

6 Conclusion

In this paper, we present Agent4Vul, a novel smart contract vulnerability detection framework that integrates LLM-based agents with multimodal learning. Our approach leverages the strengths of LLMs to effectively transform smart contract understanding capabilities into vulnerability detection capabilities. By introducing the Commentator and Vectorizer agents for the semantic and graph branches in the multimodal processing, we have achieved significant improvements in detection performance. Extensive experimental results demonstrate that Agent4Vul outperforms 19 state-of-the-art detection methods, with F1 score improvements ranging from 3.61% to 16.32%. This underscores the framework's robustness and efficacy in accurately identifying vulnerabilities across different contexts. In the future, we will enhance the capabilities of LLMs within the Commentator and Vectorizer agents by employing ensemble learning, such as weighted voting and stacking, to improve detection performance. We also recognize the computational overhead of LLMs, particularly for large-scale data, and plan to explore trade-offs like model distillation to balance efficiency and performance.

Acknowledgements This work was supported by National Key R&D Program of China (Grant No. 2022ZD0116800), National Natural Science Foundation of China (Grant Nos. 62141605, 62372493), Beijing Natural Science Foundation (Grant No. Z230001), Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing (Grant Nos. GJJ-23-001, GJJ-23-002), China Postdoctoral Fellowship Fund (Grant No. 373500), and Beihang Dare to Take Action Plan (Grant No. KG16336101).

References

1 Wood G. Ethereum: a secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper, 2014, 151: 1–32
2 Zou W, Lo D, Kochhar P S, et al. Smart contract development: challenges and opportunities. IEEE Trans Software Eng, 2021, 47: 2084–2106
3 Ante L. Smart contracts on the blockchain—a bibliometric analysis and review. Telematics Inf, 2021, 57: 101519
4 Victor F, Weintraud A M. Detecting and quantifying wash trading on decentralized cryptocurrency exchanges. In: Proceedings of the Web Conference, New York, 2021. 23–32
5 DefiLlama. Defillama hacks, 2024. Accessed: June 2024. https://defillama.com/hacks
6 Zheng Z, Xie S, Dai H N, et al. An overview on smart contracts: challenges, advances and platforms. Future Generation Comput Syst, 2020, 105: 475–491
7 Peng K, Li M, Huang H, et al. Security challenges and opportunities for smart contracts in Internet of Things: a survey. IEEE Internet Things J, 2021, 8: 12004–12020
8 Wang S, Ouyang L, Yuan Y, et al. Blockchain-enabled smart contracts: architecture, applications, and future trends. IEEE Trans Syst Man Cybern Syst, 2019, 49: 2266–2277
9 Nguyen T D, Pham L H, Sun J, et al. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020. 778–788
10 Tikhomirov S, Voskresenskaya E, Ivanitskiy I, et al. Smartcheck: static analysis of Ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, 2018. 9–16
11 Torres C F, Schütte J, State R. Osiris: hunting for integer bugs in Ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, 2018. 664–676
12 Luu L, Chu D H, Olickel H, et al. Making smart contracts smarter. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2016. 254–269
13 Mueller B. Smashing Ethereum smart contracts for fun and real profit. HITB SECCONF Amsterdam, 2018, 9: 54

14 Tsankov P, Dan A, Drachsler-Cohen D, et al. Securify: practical security analysis of smart contracts. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, 2018. 67–82

15 Feist J, Grieco G, Groce A. Slither: a static analysis framework for smart contracts. In: Proceedings of IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 2019. 8–15

16 Tann W J W, Han X J, Gupta S S, et al. Towards safer smart contracts: a sequence learning approach to detecting security threats. 2018. ArXiv:1811.06632

17 Qian P, Liu Z, He Q, et al. Towards automated reentrancy detection for smart contracts based on sequential models. IEEE Access, 2020, 8: 19685–19695

18 Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks. 2016. ArXiv:1609.02907

19 Zhuang Y, Liu Z, Qian P, et al. Smart contract vulnerability detection using graph neural networks. In: Proceedings of the 29th International Conference on International Joint Conferences on Artificial Intelligence, 2021. 3283–3290

20 Liu Z, Qian P, Wang X, et al. Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. 2021. ArXiv:2106.09282

21 Qian P, Liu Z, Yin Y, et al. Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode. In: Proceedings of the ACM Web Conference, 2023. 2220–2229

22 Jie W, Chen Q, Wang J, et al. A novel extended multimodal AI framework towards vulnerability detection in smart contracts. Inf Sci, 2023, 636: 118907

23 Chen C, Su J, Chen J, et al. When ChatGPT meets smart contract vulnerability detection: how far are we? 2023. ArXiv:2309.05520

24 Du Y, Tang X. Evaluation of ChatGPT's smart contract auditing capabilities based on chain of thought. 2024. ArXiv:2402.12023

25 Hu S, Huang T, Ilhan F, et al. Large language model-powered smart contract vulnerability detection: new perspectives. In: Proceedings of the 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA), 2023. 297–306

26 Ma W, Wu D, Sun Y, et al. Combining fine-tuning and LLM-based agents for intuitive smart contract auditing with justifications. 2024. ArXiv:2403.16073

27 Sun Y, Wu D, Xue Y, et al. GPTScan: detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, 2024. 1–13

28 Contro F, Crosara M, Ceccato M, et al. EtherSolve: computing an accurate control-flow graph from Ethereum bytecode. In: Proceedings of IEEE/ACM 29th International Conference on Program Comprehension (ICPC), 2021. 127–137

29 Luo F, Luo R, Chen T, et al. SCVHUNTER: smart contract vulnerability detection based on heterogeneous graph attention network. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, New York, 2024

30 Meng H, Huang D, Wang H, et al. Depression recognition based on dynamic facial and vocal expression features using partial least square regression. In: Proceedings of the 3rd ACM International Workshop on Audio/Visual Emotion Challenge, New York, 2013. 21–30

31 Fan W, He Z, Xing X, et al. Multi-modality depression detection via multi-scale temporal dilated CNNs. In: Proceedings of the 9th International on Audio/Visual Emotion Challenge and Workshop, New York, 2019. 73–80

32 Morales M, Scherer S, Levitan R. A linguistically-informed fusion approach for multimodal depression detection. In: Proceedings of the 5th Workshop on Computational Linguistics and Clinical Psychology: From Keyboard to Clinic, New Orleans, 2018. 13–24

33 Ke G, Meng Q, Finley T, et al. LightGBM: a highly efficient gradient boosting decision tree. In: Proceedings of Advances in Neural Information Processing Systems, 2017

34 Sun D, Wulff J, Sudderth E B, et al. A fully-connected layered model of foreground and background flow. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2013. 2451–2458

35 Zeng A, Liu X, Du Z, et al. GLM-130B: an open bilingual pre-trained model. In: Proceedings of the 11th International Conference on Learning Representations, Kigali, 2023

36 Bai J Z, Bai S, Chu Y F, et al. Qwen technical report. 2023. ArXiv:2309.16609

37 Pinecone. Voyage-lite-02-instruct: optimized text embedding model. 2024. https://www.pinecone.io/models/voyage-lite-02-instruct/

38 Muennighoff N, Su H, Wang L, et al. Generative representational instruction tuning. 2024. ArXiv:2402.09906

39 Church K W. Word2Vec. Nat Lang Eng, 2017, 23: 155–162