

LLM-BSCVM: An LLM-Based Blockchain Smart Contract Vulnerability Management Framework

Yanli Jin¹, Chunpei Li¹, Peng Fan¹, Peng Liu¹, Xianxian Li¹, Chen Liu², Wangjie Qiu³

¹Guangxi Normal University, China

Email: jyl@stu.gxnu.edu.cn, licp@gxnu.edu.cn

²Zhongguancun Laboratory, China

³Beihang University, China

arXiv:2505.17416v1 [cs.CR] 23 May 2025

Abstract—Smart contracts are a key component of the Web 3.0 ecosystem, widely applied in blockchain services and decentralized applications. However, the automated execution feature of smart contracts makes them vulnerable to potential attacks due to inherent flaws, which can lead to severe security risks and financial losses, even threatening the integrity of the entire decentralized finance system. Currently, research on smart contract vulnerabilities has evolved from traditional program analysis methods to deep learning techniques, with the gradual introduction of Large Language Models. However, existing studies mainly focus on vulnerability detection, lacking systematic cause analysis and Vulnerability Repair. To address this gap, we propose LLM-BSCVM, a Large Language Model-based smart contract vulnerability management framework, designed to provide end-to-end vulnerability detection, analysis, repair, and evaluation capabilities for Web 3.0 ecosystem. LLM-BSCVM combines retrieval-augmented generation technology and multi-agent collaboration, introducing a three-stage method of “Decompose-Retrieve-Generate.” This approach enables smart contract vulnerability management through the collaborative efforts of six intelligent agents, specifically: vulnerability detection, cause analysis, repair suggestion generation, risk assessment, vulnerability repair, and patch evaluation. Experimental results demonstrate that LLM-BSCVM achieves a vulnerability detection accuracy and F1 score exceeding 91% on benchmark datasets, comparable to the performance of state-of-the-art (SOTA) methods, while reducing the false positive rate from 7.2% in SOTA methods to 5.1%, thus enhancing the reliability of vulnerability management. Furthermore, LLM-BSCVM supports continuous security monitoring and governance of smart contracts through a knowledge base hot-swapping dynamic update mechanism. It not only provides developers with comprehensive vulnerability management services but also effectively improves the overall security of the Web 3.0 ecosystem. To support the development of Web 3.0 and blockchain security research communities, the code for this framework is open-source and available at <https://github.com/sosol717/LLM-BSCVM>.

Index Terms—Web 3.0, Blockchain, Smart Contract, Vulnerability Management, Large Language Model (LLM)

I. INTRODUCTION

With the rapid development of Web 3.0 and blockchain technology, smart contracts have become a crucial foundation for decentralized applications and the decentralized finance ecosystem. However, due to the automated execution and immutability of smart contracts, once vulnerabilities exist, attackers can exploit the logical flaws in the code, leading to severe security threats. For example, the 2016 DAO attack resulted in a loss of \$60 million [1], and in 2018, the BEC

contract caused the value of tokens to drop to zero due to an integer overflow vulnerability [2]. According to statistics, smart contract vulnerabilities have led to an accumulated economic loss of over \$20 billion [3]. Therefore, comprehensive vulnerability detection, cause analysis, repair suggestion generation, and risk assessment before deploying smart contracts are crucial for ensuring the security of the Web 3.0 ecosystem.

Several methods have been proposed for smart contract vulnerability detection, primarily including the following: (1) **Traditional Methods**: such as formal verification [4], symbolic execution [5], and fuzz testing [6]. These methods rely on expert knowledge and can provide some vulnerability detection capabilities. However, they tend to have a high false positive rate when applied to complex contracts and are difficult to scale [7]. (2) **Deep Learning Methods (e.g., RNN, LSTM, CNN [8], [9], [10])**: These methods can automatically learn vulnerability features in contracts, reducing human intervention. However, they are typically limited to pattern matching for vulnerabilities and lack a deep understanding of code syntax and semantics, making it challenging to generate detailed vulnerability cause analysis and repair solutions. (3) **Generative Large Language Model (LLM) Methods**: Recent research has shown that LLMs possess a deep understanding of code and can capture long-distance dependencies within the code, demonstrating strong generalization capabilities in vulnerability detection [11]. However, current research on LLMs for smart contract vulnerabilities is still limited to the detection stage. There are still significant shortcomings in the areas of vulnerability explainability analysis and automated repair [12], [13], [14].

The research in [15], [16] indicates that smart contract vulnerabilities are a multidimensional issue that requires comprehensive exploration. However, existing smart contract vulnerability detection methods mainly focus on vulnerability identification, while lacking fine-grained cause analysis and automated repair suggestions [12], [14], [17]. This results in detection outcomes that are not directly actionable for contract repair. Furthermore, most methods adopt black-box outputs, making it difficult for developers to understand the root causes of vulnerabilities. The repair process still relies on expert knowledge and manual analysis, which decreases the efficiency and scalability of vulnerability fixes [8], [9], [10]. Although large models have demonstrated strong code

comprehension capabilities in the field of vulnerability detection [13], [14], current research lacks a systematic vulnerability management framework, making it difficult to cover the post-detection processes of analysis, evaluation, and repair. Therefore, building an end-to-end smart contract vulnerability management system that encompasses vulnerability detection, cause analysis, repair suggestion generation, risk assessment, and audit reporting, in order to enhance the automation, explainability, and reliability of vulnerability governance, remains a core challenge that needs to be addressed in the current research landscape. Research [18] further shows that large language models possess great potential in handling vulnerability management tasks, providing new insights for developing interpretable vulnerability management methods and laying the technological foundation for more comprehensive vulnerability management services.

To address the aforementioned issues, this paper proposes LLM-BSCVM, a Large Language Model (LLM)-based smart contract vulnerability management framework, which provides comprehensive capabilities for vulnerability detection, analysis, repair, and evaluation. To build a more robust vulnerability management system, LLM-BSCVM introduces a “Decompose-Retrieve-Generate” three-stage approach specifically for smart contract vulnerability management. This approach aims to systematically address the detection, analysis, and repair of smart contract vulnerabilities, enhancing the automation, explainability, and reliability of vulnerability governance. LLM-BSCVM employs a three-stage vulnerability management method:

(1) **Task Decomposition:** Based on the concept of multi-agent collaboration [19], the vulnerability management task is subdivided into six sub-tasks: vulnerability detection, cause analysis, repair suggestion generation, risk assessment, vulnerability repair, and patch evaluation. Each sub-task is independently handled by different agents, and multiple agents collaborate, with the results of preceding tasks supporting subsequent ones, forming a progressive inference process.

(2) **Knowledge Retrieval:** By integrating retrieval-augmented generation [20], during the execution of each agent’s task, the vulnerability knowledge base and external data sources are dynamically retrieved to enhance the model’s understanding of vulnerability causes and repair strategies, improving the accuracy of inference.

(3) **Result Generation:** The agents, in conjunction with the retrieved relevant knowledge, generate vulnerability detection reports, cause analyses, repair suggestions, and final audit reports, thus improving the explainability and automation of vulnerability management.

In the vulnerability detection phase, we use a separately fine-tuned detection model (CodeLlama) to provide initial results, ensuring detection accuracy. Subsequent tasks are progressively inferred by the foundational large model (CodeLlama), and finally, after vulnerability repair is completed, a more computationally expensive advanced LLM is used for final audit evaluation to ensure the reliability of the output results. The main contributions of this paper are as follows:

- **LLM-BSCVM:** The first smart contract vulnerability management framework, integrating vulnerability detection, cause analysis, risk assessment, vulnerability repair, repair verification, and report generation, thereby constructing a complete vulnerability governance system suitable for the Web 3.0 ecosystem.
- **Decompose-Retrieve-Generate Method:** The proposed “Decompose-Retrieve-Generate” three-stage method for smart contract vulnerability management, combining multi-agent collaboration and retrieval-augmented generation. This method enhances LLM’s explainability and accuracy in smart contract vulnerability management through task decomposition, dynamic knowledge expansion, and inference enhancement.
- **Experimental Validation:** Experiments on a smart contract dataset validate the effectiveness of LLM-BSCVM. The results show that the framework’s vulnerability detection accuracy and F1 score exceed 91%, comparable to state-of-the-art (SOTA) methods. At the same time, the false positive rate is reduced from 7.2% in SOTA methods to 5.1%, significantly decreasing the error alarm rate and improving the precision and feasibility of vulnerability repair.

This study not only advances the application of AI in smart contract vulnerability management but also offers a novel approach to Web 3.0 and blockchain security governance. The code is open-sourced (<https://github.com/sosol717/LLM-BSCVM>) to support the development of the Web 3.0 and blockchain security research community.

II. RELATED WORK

A. Smart Contract Vulnerability Detection

Traditional methods are widely used for detecting vulnerabilities in smart contracts, including fuzz testing, symbolic execution tools, and formal verification methods. Fuzz testing tools such as Contractfuzzer [6], Reguard [17], and Soliaudit [21] discover vulnerabilities by simulating various inputs at runtime. However, these tools typically only cover a subset of code paths, potentially missing latent vulnerabilities, and often have a high false positive rate. Symbolic execution tools like Oyente [5], Manticore [22], and WANA [23] conduct boundary checks by analyzing the execution paths of contracts. While they can identify more complex vulnerabilities, they are computationally expensive and are often limited to smaller or simpler contracts. Formal verification tools, such as Zeus [4] and VeriSmart [24], provide rigorous mathematical proofs, but their comprehensive verification of complex contracts remains challenging.

With the development of deep learning technologies, researchers have started using deep learning models to learn vulnerability features from contract samples for detection. Diversevul [8] employs deep learning algorithms to extract vulnerability features from contract samples, and then uses neural network models to detect different types of vulnerabilities. Contractward [9] analyzes the bytecode and opcodes

of contracts, leveraging deep neural network models to detect potential vulnerabilities, including common issues like reentrancy attacks. DA-GNN [10], on the other hand, constructs a contract’s control flow graph (CFG) and combines graph-based features to capture the complex relationships between nodes within the contract, thereby more accurately identifying and detecting vulnerabilities in smart contracts.

In recent years, Large Language Models (LLMs) have provided new approaches for vulnerability detection. Researchers have not only utilized traditional deep learning models but also applied LLMs for vulnerability detection. GPTScan [12] attempts to combine large models with static analysis for detecting logical vulnerabilities in smart contracts. GPTLENS [14] is a two-stage adversarial framework that uses GPT-4 to mine potential vulnerabilities in smart contracts, with the goal of identifying as many real vulnerabilities as possible. TrustLLM [13] conducts intuitive smart contract audits and generates audit explanations through majority voting and multiple-prompt fine-tuning. LLMSmartSec [25] leverages GPT-4 to understand smart contracts and trains an LLMGraphAgent to achieve low-cost automated security auditing. LLM4Vuln [26] accurately evaluates the performance of LLMs in vulnerability detection by separating the active search for additional information, employing relevant vulnerability knowledge, and generating structured results.

However, all of these studies primarily explore vulnerabilities from a single dimension and mainly focus on vulnerability detection tools, which presents two major limitations. On one hand, detection tools lack fine-grained interpretability analysis [8], [9], [10], failing to effectively reveal the root causes of vulnerabilities, and they do not provide reliable repair suggestions, leading developers to still rely heavily on manual analysis and expert experience during the vulnerability repair process. On the other hand, these tools do not cover the full lifecycle of vulnerabilities [12], [14], especially in terms of interpretability analysis and automated repair, where there are significant shortcomings. This results in an inability to address diverse security needs, leading to issues such as inefficiency, insufficient accuracy, and increased security risks.

B. Large Language Models (LLMs)

Large Language Models (LLMs) are a class of deep learning models based on the Transformer architecture [27]. They undergo pre-training on large-scale text data through self-supervised learning, thereby acquiring rich language knowledge. As the training data increases, LLMs are capable of processing longer context information, demonstrating enhanced abilities in language understanding and generation. In addition to capturing subtle nuances in text, LLMs can also generate grammatically correct and semantically coherent natural language, improving the quality and fluency of text generation. In the domain of code understanding and generation, LLMs have undergone several evolutions. From the GPT series, which supports multilingual code understanding [28], to the specially optimized CodeLlama [29], code-specific large lan-

guage models have shown tremendous potential in program understanding, analysis, and generation.

Fine-tuning is the process of domain-specific optimization of a pre-trained large language model. During the pre-training phase, LLMs learn extensive language knowledge by training on large-scale general datasets. However, for certain specific tasks or domains, the pre-trained model may not be fully applicable. Through fine-tuning, LLMs can be optimized for specific application scenarios, thereby improving their performance on targeted tasks. Compared to training from scratch, fine-tuning can significantly enhance model performance in a shorter time frame while effectively saving computational resources.

C. Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) [20] introduces an external knowledge retrieval mechanism into the generation process of large language models, effectively addressing the knowledge gap that large models may have in specific domains. In traditional large pretrained language models, generation methods typically rely on the parameters learned during pretraining to generate text. However, this approach may produce inaccurate outputs when handling complex tasks or scenarios requiring real-time updates. The RAG technique enables the generation model to dynamically retrieve relevant documents from an external knowledge base while processing the input, merging the retrieved information with the input, thereby enhancing the accuracy and reliability of the generated results.

III. DETAILED DESIGN OF LLM-BSCVM

A. Framework Overview

The overall framework we propose is shown in Figure 1. The LLM-BSCVM framework implements smart contract vulnerability management through a three-stage “Decompose-Retrieve-Generate” approach. This method organically combines retrieval-augmented generation (RAG) with a multi-agent collaborative task decomposition mechanism, breaking down the complex vulnerability management process into several subtasks. At each subtask, relevant specialized knowledge is dynamically retrieved, ensuring that the model’s output of repair suggestions and evaluation results is based on reliable and accurate knowledge. We will discuss each stage in more detail in the following chapters.

- **Task Decomposition Stage:** The vulnerability management task is subdivided into six subtasks, each handled independently by a different agent. Multiple agents work collaboratively, with the results of previous tasks providing support for subsequent tasks, forming a progressive reasoning process.
- **Knowledge Retrieval Stage:** Each agent uses retrieval-augmented generation (RAG) technology to access relevant information in real-time from the vulnerability knowledge base and external data sources.

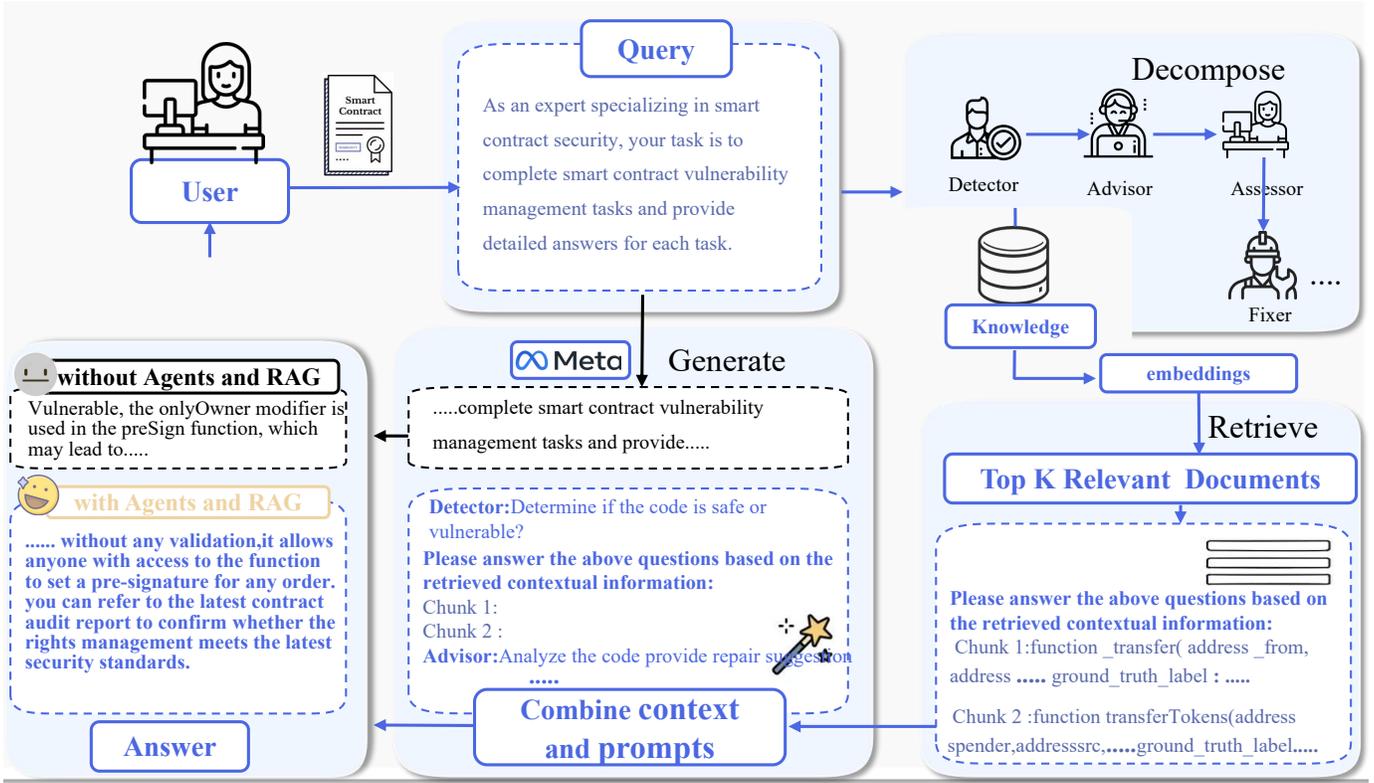


Fig. 1. Framework of our proposed approach LLM-BSCVM.

- **Result Generation Stage:** The LLM integrates the knowledge obtained from the retrieval stage into the prompts and generates interpretable results for vulnerability detection, evaluation, analysis, and repair. Finally, an objective audit report is generated.

B. Task Decomposition Stage

Smart contract vulnerability management is a multi-step, complex process that covers various stages, including vulnerability detection, repair suggestions, risk assessment, and vulnerability repair. To ensure the efficient execution and accuracy of each task, we need to break the entire process down into multiple independent yet interrelated subtasks. Each subtask has its own clear objectives and responsibilities. While minimizing interference between tasks, these subtasks must also provide necessary support for subsequent tasks.

Multi-agent collaboration is an effective method for solving complex problems, particularly those that require multiple independent entities to work together. In the smart contract vulnerability management process, the entire workflow is broken down into six subtasks, each assigned to an independent agent. Each agent, based on its specific task objectives and expertise, performs independent reasoning and decision-making, passing its output to subsequent agents. This collaborative working mechanism ensures efficient task interconnection and information flow, thereby improving the overall accuracy and reliability of the management process. As shown in Figure 2, the entire vulnerability management process is divided into six

subtasks: vulnerability detection, repair suggestion generation, risk assessment, vulnerability repair, patch correctness evaluation, and detection report generation. Each subtask is handled by a different agent, corresponding to its specific task:

Vulnerability Detection Agent (Detector): As the core of smart contract auditing, the vulnerability detection agent is responsible for identifying potential vulnerabilities in the contract and providing accurate detection results for subsequent tasks. To improve the accuracy of vulnerability detection, the results from three dimensions are integrated at this stage. (1) **Static Analysis**, Based on a predefined pattern library, common vulnerabilities, such as reentrancy attacks and arithmetic overflows, are detected; (2) **Using Retrieval-Augmented Generation (RAG) technology**, top-k contracts similar to the target contract are retrieved in real time, and relevant information is sourced from the contract library; (3) **Inference Analysis**: A fine-tuned model deeply understands the business logic of the contract, assessing potential security issues.

Each dimension's analysis results are marked as "safe" or "vulnerable." Based on this, we design two decision-making approaches to combine the results from all dimensions: one is through weighted fusion, combining a dynamic threshold mechanism to determine the final security; the other is through a voting mechanism, where the contract's final security is decided by majority vote. It should be noted that the analysis results of the first two dimensions are not directly provided

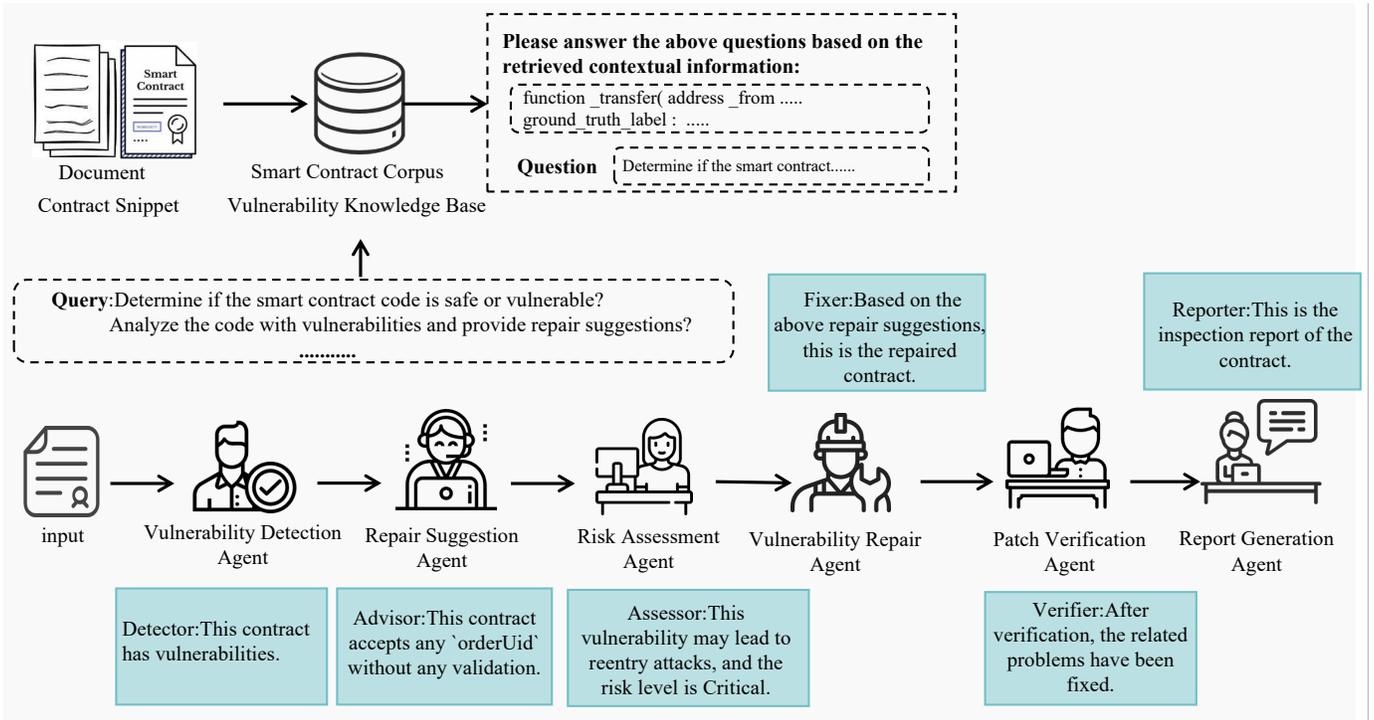


Fig. 2. The specific of task decomposition.

to the model for the final decision, as experiments show that excessive external information may introduce noise, affecting the accuracy of detection.

Repair Suggestion Agent (Advisor): After vulnerability detection, the repair suggestion agent is responsible for providing targeted repair solutions for the detected vulnerabilities. Using RAG technology, it retrieves real-time information from the vulnerability knowledge base and, combined with the model’s generative capabilities, provides specific repair suggestions for each vulnerability. The repair solutions include root cause analysis of the vulnerability, potential impact assessment, repair steps, and preventive measures, ensuring the comprehensiveness and effectiveness of the repair solutions.

Risk Assessment Agent (Assessor): The risk assessment agent systematically evaluates the risk level of each vulnerability by analyzing audit reports from major security audit agencies, vulnerability disclosure data, and the CVSS score standard from the CVE vulnerability database. Based on a four-level risk assessment system (Critical, High, Medium, Low), and leveraging the model’s reasoning capabilities, the agent assigns a risk level to each vulnerability, which provides a basis for prioritizing the subsequent repair tasks.

Vulnerability Repair Agent (Fixer): The vulnerability repair agent is responsible for fixing the vulnerabilities in the smart contract based on the repair suggestions and risk assessment results. The agent first sorts vulnerabilities according to their repair priority, then, considering contextual information and dependencies, generates repair code that complies with programming standards, ensuring the security and effective-

ness of the repair process.

Patch Verification Agent (Verifier): The repaired code must undergo verification to ensure no new security issues are introduced. We adopt the concept of multi-agent debate, using independent evaluation models to verify the repaired code. The verification process mainly includes two aspects: first, ensuring that the repair successfully eliminates the vulnerability, and second, ensuring that no new security issues are introduced during the repair process.

Report Generation Agent (Reporter): Finally, the report generation agent integrates the analysis results from the previous stages into a complete audit report. The report includes seven key sections: contract basic information overview, executive summary, audit methodology explanation, vulnerability discovery summary, in-depth analysis report, improvement suggestions, and compliance disclaimer, providing a comprehensive reference for smart contract developers.

C. Knowledge Retrieval Stage

In terms of knowledge base construction, as shown in Figure 3, we have utilized RAG technology to build two knowledge bases aimed at smart contract vulnerability management.

- **Smart Contract Corpus:** This corpus contains a large volume of smart contract code, primarily used for similarity retrieval in the vulnerability detection stage. The data is sourced from [13], and collected from the well-known auditing website Solodit [30], analyzing a total of 263 smart contract audit reports.
- **Vulnerability Knowledge Base:** This knowledge base stores documents related to smart contract vulnerabilities,

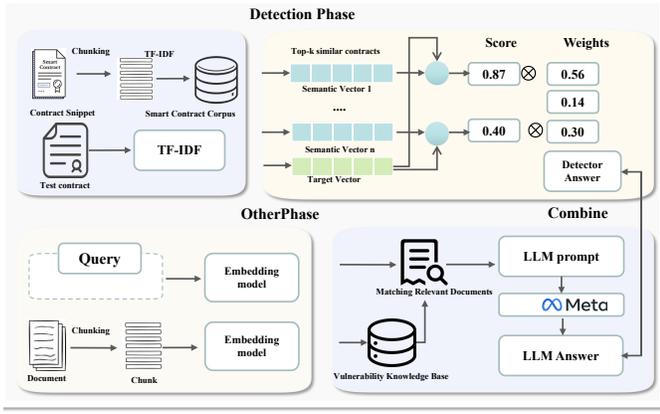


Fig. 3. The specific of knowledge retrieval.

including smart contract audit reports [31] from renowned security institutions, security best practices [32], and programming standard documents [33]. These documents are primarily sourced from leading companies in the industry, such as Solidit [30] and Smart Contract Weakness Classification (SWC) [34]. The vulnerability knowledge base provides relevant documents and background information for tasks at all stages.

For processing the smart contract corpus, we applied the TF-IDF algorithm to vectorize the contract code. By computing the product of Term Frequency (TF) and Inverse Document Frequency (IDF), we are able to capture the key features within the contract code. We then used cosine similarity to measure the similarity between the target contract and other contracts in the corpus, selecting the top-k most similar contracts for further analysis. To improve the accuracy of similarity retrieval, we assigned different weights to contracts based on their similarity rankings (the higher the rank, the greater the weight) and calculated the final probability of vulnerability presence through weighted computation. As shown in Formula 1, where V_a and V_b represent the contract encoding vectors to be compared.

For processing the vulnerability knowledge base data, we employed vector embedding techniques to convert unstructured text into vector representations in a high-dimensional semantic space. This enables LLM-BSCVM to perform semantic similarity retrieval based on the specific requirements of the analysis stage (e.g., vulnerability detection or repair recommendation generation), providing the model with specialized knowledge support.

$$\text{cosine similarity}(V_a, V_b) = \frac{V_a \cdot V_b}{\|V_a\| \|V_b\|} \quad (1)$$

D. Result Generation Stage

In the process of smart contract vulnerability management, each intelligent agent retrieves and integrates relevant domain knowledge, embedding it into the task execution process. Based on this, it generates explainable vulnerability detection,

evaluation, analysis, and repair results. Ultimately, based on the collaborative outcomes of all intelligent agents, a comprehensive audit report is produced, offering developers practical and feasible vulnerability repair strategies and improvement recommendations.

The basic prompt template consists of four core components: role-playing, task description, expected output, and background information. Taking vulnerability detection as an example, as shown in Figure 4, the input in the template is the smart contract code to be analyzed, with “code” serving as a placeholder. The background information includes best security practices, Solidity programming guidelines, etc. This information is dynamically adjusted based on task requirements to help the model better understand the task objectives and provide structured feedback.

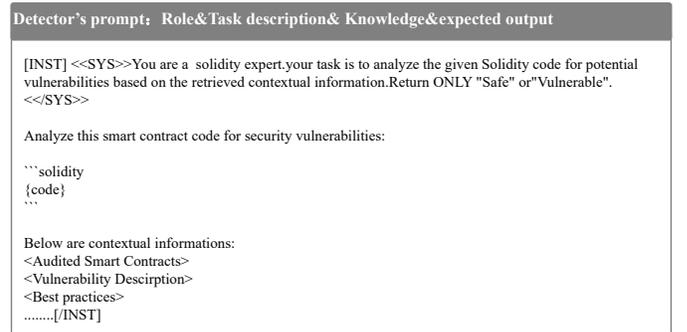


Fig. 4. Prompt Template of LLM-BSCVM: An Example for Vulnerability Detection Tasks.

IV. EVALUATION AND RESULTS

A. Experimental Setup

Our dataset is compiled from two sources:

- (1) TrustLLM [13], which originates from the renowned auditing website Solodit [30], comprising a total of 263 smart contract audit reports.
- (2) smart contract audit reports from Dappscan [31], which includes analysis of 1,199 open-source audit reports from 29 security teams.

In this study, we set the value of k in similarity retrieval to 5, and we also designed a flexible interface to integrate various large language models (LLMs), including Codellama [29], CodeBERT [35], CodeT5 [36], and Llama [37]. Within our framework, we employed different model configurations according to task requirements: the vulnerability detection task uses Codellama fine-tuned with LoRA; tasks such as vulnerability fix suggestion generation, risk level assessment, vulnerability repair, and report generation use the base Codellama model. For patch correctness evaluation, we adopted the multi-agent debate concept [38], introducing GPT-4 as an independent verifier to assess whether the patch successfully fixed the target vulnerability.

B. Experimental Results

Vulnerability Detection. To evaluate the performance of LLM-BSCVM in vulnerability detection, we employed ac-

curacy, precision, recall, and F1-score as evaluation metrics. The experiments were conducted using the same dataset as TrustLLM. LLM-BSCVM (W) refers to the weighted fusion approach, with weight distribution as follows: model-based detection (70%), static analysis (10%), and retrieval-based detection (20%). In contrast, LLM-BSCVM (V) adopts a majority voting mechanism, making decisions based on model predictions, static analysis results, and similarity matching outcomes.

As shown in Table I, the weighted approach, LLM-BSCVM (W), achieved the highest detection accuracy of 91.11% and precision of 94.95%. These results indicate that LLM-BSCVM (W) effectively identifies and localizes vulnerabilities in smart contracts, demonstrating strong detection capability. Comparatively, although LLM-BSCVM (V) also performed well with an F1-score of 89%, recall of 86%, precision of 93%, and accuracy of 89%, it exhibited slightly inferior performance. We hypothesize that this difference arises due to the majority voting method employed in LLM-BSCVM (V), which aggregates results from different components through a simple voting mechanism. However, this approach may amplify the influence of weaker components (such as static analysis or retrieval), negatively impacting the final decision and leading to a slight decrease in overall performance. In contrast, LLM-BSCVM (E) incorporates retrieved similar contracts and security documentation as contextual information to theoretically enhance detection accuracy. However, the actual results show a decline in performance. We speculate that the introduction of excessive contextual information may have distracted the model’s attention, thereby reducing its ability to accurately detect specific vulnerabilities.

Additionally, as presented in Table II, LLM-BSCVM

TABLE I
COMPARISON OF DETECTION PERFORMANCE OF LLM-BSCVM USING DIFFERENT METHODS

Approach	F1	Recall	Precision	Accuracy
LLM-BSCVM(E)	0.7890	0.7125	0.8467	0.8042
LLM-BSCVM(V)	0.8996	0.8689	0.9326	0.8999
LLM-BSCVM(W)	0.9104	0.8743	0.9506	0.9111

achieves performance comparable to TrustLLM [13] in terms of F1-score and precision, reaching high scores of 91% and 91%, respectively. An F1-score of 91% indicates that the model maintains a strong balance between precision and recall in vulnerability detection. A precision of 91% suggests that 91% of the contracts predicted as vulnerable are indeed true positives. These metrics demonstrate that LLM-BSCVM effectively distinguishes between vulnerable and non-vulnerable contracts, ensuring both efficiency and accuracy in detection.

Regarding the false positive rate, LLM-BSCVM achieved a false positive rate of 5.1%, which is 2.2 percentage points lower than TrustLLM’s 7.2%. The false positive rate (FPR) is a crucial indicator of a model’s detection accuracy, where a lower FPR implies a more precise identification of normal contracts, reducing the risk of misclassification. By integrating a

fine-tuned large language model, static analysis, and retrieval-augmented techniques, LLM-BSCVM identifies and verifies vulnerabilities at multiple levels, mitigating errors caused by insufficient knowledge coverage and significantly reducing the false positive rate.

We further compared LLM-BSCVM with both base models and fine-tuned models. The base models include Codellama 13B, Codellama 7B, CodeBERT, CodeT5, and Llama, while the fine-tuned models consist of their respective fine-tuned versions. As shown in Table II, LLM-BSCVM outperforms all baseline models across all evaluation metrics, particularly in accuracy, where it surpasses Codellama 13B by approximately 48 percentage points. This result suggests that base models (e.g., Codellama and CodeBERT) lack sufficient contextual understanding and domain knowledge when handling complex smart contracts, leading to significantly lower accuracy compared to LLM-BSCVM.

TABLE II
COMPARISON OF DETECTION PERFORMANCE BETWEEN LLM-BSCVM AND ZERO-SHOT LEARNING LLMs

Approach	F1	Recall	Precision	Accuracy
Codellama 7B	0.5278	0.6749	0.4333	0.3766
Codellama 13B	0.5791	0.8708	0.4338	0.4255
CodeT5	0.6183	0.7568	0.5226	0.5176
CodeBERT	0.5208	0.5464	0.4975	0.4810
Llama 8B	0.5938	0.8087	0.4691	0.4288
LLM-BSCVM(W)	0.9104	0.8743	0.9506	0.9111

As illustrated in Table Table III, fine-tuned models exhibit substantial performance improvements over their base counterparts across all metrics, particularly in precision, recall, and F1-score, highlighting their enhanced capability in vulnerability detection. For instance, the fine-tuned versions of Codellama 13B and CodeBERT show significant improvements in precision and recall, indicating that the fine-tuning process enhances the model’s understanding of smart contract contexts and its ability to accurately localize vulnerabilities. However, despite these improvements, LLM-BSCVM consistently outperforms all fine-tuned models, particularly in accuracy, where even the fine-tuned versions of Codellama 13B and CodeBERT fall short of LLM-BSCVM’s performance.

Furthermore, to validate the effectiveness of each component in the vulnerability detection method, we conducted an ablation study by systematically removing different components to evaluate their contributions to overall performance. The LLM-BSCVM vulnerability detection framework consists of three core components: (1) a LoRA fine-tuned CodeLlama-13B model, (2) a static analysis module based on predefined vulnerability patterns, and (3) a retrieval-augmented module leveraging historical vulnerability knowledge. In our experiments, we designed two variants: w/o Static, which removes the static analysis module, and w/o RAG, which eliminates the retrieval-augmented component. As shown in Table IV, compared to the complete LLM-BSCVM framework, the removal of the static analysis module led to a significant decline in F1-score, accuracy, and precision. This result indicates

TABLE III
COMPARISON OF DETECTION PERFORMANCE BETWEEN LLM-BSCVM
AND LoRA FINE-TUNED LLMs

Approach	F1	Recall	Precision	Accuracy
Codellama 7B(Lora)	0.8451	0.8445	0.9211	0.8954
Codellama 13B(Lora)	0.8918	0.8661	0.9388	0.9027
CodeT5(Lora)	0.8543	0.7887	0.9411	0.8543
CodeBERT(Lora)	0.8121	0.7230	0.9111	0.8564
Ilama 8B(Lora)	0.8231	0.7554	0.9233	0.8422
TrustLLM	0.9121	0.8934	0.9316	0.9111
LLM-BSCVM(W)	0.9104	0.8743	0.9506	0.9111

Smart contract code

```
pragma solidity ^0.4.22;
/Solidity
function preSign(bytes calldata orderUid) external onlyOwner {
    ICowSettlement(COW_SETTLEMENT).setPreSignature(orderUid, true);
}

Ground truth:The 'preSign' function in the 'Staking' contract could use some basic validations to ensure the security of the system. Currently, the function accepts any 'orderUid' without any validation. This can be a potential vulnerability as it allows anyone with access to the function to set a pre-signature for any order, regardless of its validity. One possible attack vector is a rug-pull scenario. Since the 'orderUid' is not validated, an attacker can create a malicious order with a receiver address that they control. By setting a pre-signature for this order, the attacker can later execute the order and transfer funds to their own address.
```

Fig. 5. Example Contract: preSign Contract.

that static analysis effectively identifies common and easily detectable vulnerability patterns, such as reentrancy attacks and integer overflows, which can often be recognized through simple pattern matching. Consequently, the removal of this module resulted in degraded model performance. Similarly,

TABLE IV
RESULTS OF THE ABLATION EXPERIMENTS ON LLM-BSCVM

Approach	F1	Recall	Precision	Accuracy
W/o Static	0.8848	0.8497	0.9228	0.8858
W/o RAG	0.8440	0.7541	0.9483	0.8561
LLM-BSCVM(W)	0.9104	0.8743	0.9506	0.9111

the w/o RAG variant also exhibited a performance decline. The RAG module dynamically retrieves knowledge from a vulnerability database, supplementing the model’s knowledge gaps in vulnerability detection. This component is particularly beneficial when encountering previously unseen vulnerability types, as it allows the model to access the latest relevant data in real time, thereby enhancing both accuracy and robustness.

Other Tasks. Figure 5 illustrates the preSign contract used to evaluate the effectiveness of LLM-BSCVM in four additional tasks: repair suggestion generation, risk level assessment, and vulnerability repair.

Figure 6 illustrates the repair suggestions generated by LLM-BSCVM, covering five aspects: vulnerability name, cause analysis, potential impact assessment, specific repair steps, and preventive measures recommendations. These suggestions provide a detailed description of the vulnerabilities in the preSign contract and their corresponding repair solutions. Compared to the ground truth repair suggestions, the suggestions generated by LLM-BSCVM exhibit high semantic

LLM-BSCVM's Repair Suggestion

Vulnerability:Unprotected Function

Description: The preSign function is marked as external and onlyOwner, but it does not perform any access control checks to ensure that only the owner of the contract can call it. This means that any address can call the function and set the pre-signature for any order.

Impact: An attacker can call the preSign function and set the pre-signature for any order, potentially allowing them to manipulate the settlement process and steal funds.

Fix: Add access control checks to the preSign function to ensure that only the owner of the contract can call it. For example:

```
function preSign(bytes calldata orderUid) external onlyOwner {
    require(msg.sender == owner, "Only the owner can call this function");
    ICowSettlement(COW_SETTLEMENT).setPreSignature(orderUid, true);
}
```

Prevention: Always use access control checks to ensure that only authorized addresses can call functions that modify the state of the contract.

Fig. 6. repair suggestion Results of LLM-BSCVM.

LLM-BSCVM's Risk Assessment

1
Critical

0
High

0
Medium

0
Low

Unprotected Function Critical

The preSign function is marked as external and onlyOwner, but it does not perform any access control checks to ensure that only the owner of the contract can call it. This means that any address can call the function and set the pre-signature for any order.

Fig. 7. Risk Assessment Results of LLM-BSCVM.

consistency, both pointing out that an attacker could exploit the preSign function to set pre-signatures for any order, thereby leading to the risk of asset theft. Although the phrasing differs, LLM-BSCVM offers a comprehensive analysis of the vulnerability’s impact and presents the same essential repair solution.

Figure 7 presents the risk level assessment results for the preSign contract. For the identified vulnerability, “Unprotected Function,” LLM-BSCVM correctly assigns a “Critical” risk level and provides a statistical distribution of different risk levels.

Figure 8 displays the results after the vulnerability was repaired. The repaired contract introduces access control checks in the preSign function, ensuring that only the contract owner can invoke this function, along with the necessary validation logic, effectively eliminating the vulnerability.

Vulnerability repair. To evaluate the effectiveness of

LLM-BSCVM's Patch

Original_code:

```
function preSign(bytes calldata orderUid) external onlyOwner {
    ICowSettlement(COW_SETTLEMENT).setPreSignature(orderUid, true);
}

Fixed_code:
```

```
function preSign(bytes memory orderuid) public onlyowner returns (bool) {
    require(msgsender == owner, "Only the Owner Can Call This Function");
    ICowsettlement(cowsettlement).setpresignature(ordersuid, true);
    return true;
}
```

Fig. 8. Vulnerability repair Results of LLM-BSCVM.

LLM-BSCVM in vulnerability repair, we utilized another independent large model to validate the repaired smart contracts. The results show that approximately 21% of the contracts successfully passed the validation, indicating that the model is capable of vulnerability repair and can effectively reduce the impact of certain high-risk vulnerabilities. For the contracts that did not pass the validation, further analysis revealed that, during the repair process, some contracts might have adjusted their code structure or security mechanisms, leading to the detection of potential risks under certain evaluation criteria. Additionally, discrepancies in evaluation standards can lead to different validation results. Therefore, further optimization of the repair strategy is needed to improve the success rate of repairs.

Report Generation. Although the automatically generated reports have not yet fully reached the reliability level of human expert reports, their advantages are significant. They not only provide detailed repair suggestions but also include the repaired contract code as a reference. Furthermore, compared to traditional manual audit reports, the generation time is significantly shortened, greatly improving efficiency.

V. CONCLUSION

The widespread application of smart contracts in the Web 3.0 ecosystem is accompanied by significant security challenges, where vulnerabilities can lead to substantial economic losses and systemic risks. To address this, this paper proposes LLM-BSCVM, the first end-to-end vulnerability management framework for smart contracts, designed to provide comprehensive functions for vulnerability detection, root cause analysis, repair recommendations, risk assessment, and audit reporting. The core innovation of LLM-BSCVM lies in its “Decompose-Retrieve-Generate” three-stage approach, which includes: (1) Task Decomposition, based on the concept of multi-agent collaboration, breaking down the vulnerability management process to facilitate progressive reasoning in vulnerability detection, repair suggestions, and risk assessment; (2) Knowledge Retrieval, integrating the vulnerability knowledge base with external data sources in real-time to enhance contextual understanding; (3) Result Generation, where agents combine retrieved relevant knowledge to generate explainable vulnerability analysis, repair plans, and final security audit reports. Experimental evaluation shows that LLM-BSCVM achieves a vulnerability detection accuracy and F1 score of 91% on benchmark datasets, while the false positive rate decreases from the state-of-the-art (SOTA) 7.2% to 5.1%, enhancing the reliability and feasibility of vulnerability repair while maintaining high detection performance. Future work will focus on (1) integrating symbolic execution and formal verification to improve detection accuracy, and (2) optimizing LLM reasoning transparency through human-computer interaction. LLM-BSCVM is expected to advance the application of AI in smart contract security and provide more intelligent, automated security guarantees for the Web 3.0 ecosystem.

REFERENCES

- [1] B. Ghaleb, A. Al-Dubai, E. Ekonomou, M. Qasem, I. Romdhani, and L. Mackenzie, “Addressing the dao insider attack in rpl’s internet of things networks,” *IEEE Communications Letters*, vol. 23, no. 1, pp. 68–71, 2018.
- [2] S. Hesseauer, “Batch overflow bug on ethereum ERC20 token contracts and safemath,” 2018.
- [3] S. Sayeed, H. Marco-Gisbert, and T. Caira, “Smart contract: Attacks and protections,” *Ieee Access*, vol. 8, pp. 24 416–24 427, 2020.
- [4] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “Zeus: analyzing safety of smart contracts.” in *Ndss*, 2018, pp. 1–12.
- [5] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.
- [6] B. Jiang, Y. Liu, and W. K. Chan, “Contractfuzzer: Fuzzing smart contracts for vulnerability detection,” in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 259–269.
- [7] H. Liu, Y. Fan, L. Feng, and Z. Wei, “Vulnerable smart contract function locating based on multi-relational nested graph convolutional network,” *Journal of Systems and Software*, vol. 204, p. 111775, 2023.
- [8] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, “Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection,” in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 654–668.
- [9] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, “Contractward: Automated vulnerability detection models for ethereum smart contracts,” *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2020.
- [10] Z. Zhen, X. Zhao, J. Zhang, Y. Wang, and H. Chen, “Da-gnn: A smart contract vulnerability detection method based on dual attention graph neural network,” *Computer Networks*, vol. 242, p. 110238, 2024.
- [11] J. Wang and Y. Chen, “A review on code generation with llms: Application and evaluation,” in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 2023, pp. 284–289.
- [12] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, “Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
- [13] W. Ma, D. Wu, Y. Sun, T. Wang, S. Liu, J. Zhang, Y. Xue, and Y. Liu, “Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications,” *arXiv preprint arXiv:2403.16073*, 2024.
- [14] S. Hu, T. Huang, F. Ilhan, S. F. Tekin, and L. Liu, “Large language model-powered smart contract vulnerability detection: New perspectives,” in *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2023, pp. 297–306.
- [15] H. Chu, P. Zhang, H. Dong, Y. Xiao, S. Ji, and W. Li, “A survey on smart contract vulnerabilities: Data sources, detection and repair,” *Information and Software Technology*, vol. 159, p. 107221, 2023.
- [16] N. K. Kumar, N. V. Honnunar, M. S. Prakash, and J. Lohith, “Vulnerabilities in smart contracts: A detailed survey of detection and mitigation methodologies,” in *2024 International Conference on Emerging Technologies in Computer Science for Interdisciplinary Applications (ICETCS)*. IEEE, 2024, pp. 1–7.
- [17] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, “Reguard: finding reentrancy bugs in smart contracts,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, 2018, pp. 65–68.
- [18] P. Liu, J. Liu, L. Fu, K. Lu, Y. Xia, X. Zhang, W. Chen, H. Weng, S. Ji, and W. Wang, “Exploring {ChatGPT’s} capabilities on vulnerability management,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 811–828.
- [19] W. Chen, Y. Su, J. Zuo, C. Yang, C. Yuan, C. Qian, C.-M. Chan, Y. Qin, Y. Lu, R. Xie *et al.*, “Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents,” *arXiv preprint arXiv:2308.10848*, vol. 2, no. 4, p. 6, 2023.
- [20] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, “Retrieval-augmented-generation for knowledge-intensive nlp tasks,” *Advances in neural information processing systems*, vol. 33, pp. 9459–9474, 2020.

- [21] J.-W. Liao, T.-T. Tsai, C.-K. He, and C.-W. Tien, "Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing," in *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. IEEE, 2019, pp. 458–465.
- [22] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.
- [23] B. Jiang, Y. Chen, D. Wang, I. Ashraf, and W. K. Chan, "Wana: Symbolic execution of wasm bytecode for extensible smart contract vulnerability detection," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 926–937.
- [24] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1678–1694.
- [25] V. Mothukuri, R. M. Parizi, and J. L. Massa, "Llmsmartsec: Smart contract security auditing with llm and annotated control flow graph," in *2024 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2024, pp. 434–441.
- [26] Y. Sun, D. Wu, Y. Xue, H. Liu, W. Ma, L. Zhang, Y. Liu, and Y. Li, "Llm4vuln: A unified evaluation framework for decoupling and enhancing llms' vulnerability reasoning," *arXiv preprint arXiv:2401.16185*, 2024.
- [27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
- [28] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [29] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.
- [30] Solodit, "Solodit - all findings from popular audit platforms," <https://solodit.xyz/>, 2024, accessed: 2024-03-07.
- [31] Z. Zheng, J. Su, J. Chen, D. Lo, Z. Zhong, and M. Ye, "Dappscan: building large-scale datasets for smart contract weaknesses in dapp projects," *IEEE Transactions on Software Engineering*, 2024.
- [32] Y. Riady, "Best practices for smart contract development," <https://yos.io/2019/11/10/smart-contract-development-best-practices/>, 2019, accessed: 2024-03-07.
- [33] Ethereum Enterprise Alliance, "Eea Ethtrust Security Levels Specification Version 2," <https://www.eea.ethereum.org/ethtrust/>, 2024, accessed: 2024-03-07.
- [34] SmartContractSecurity, "Smart Contract Weakness Classification and Test Cases," <https://github.com/SmartContractSecurity/SWC-registry>, 2020, accessed: 2024-03-07.
- [35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
- [36] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.
- [37] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.
- [38] C. Wang, J. Zhang, J. Gao, L. Xia, Z. Guan, and Z. Chen, "Contract-tinker: Llm-empowered vulnerability repair for real-world smart contracts," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2350–2353.