The Journal of Systems and Software 231 (2026) 112635

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

CodeSpeak: Improving smart contract vulnerability detection via LLM-assisted code analysis☆

Shuyu Chang [a,1], Chen Geng [a,1], Haiping Huang [a,b,*], Rui Wang [a,b], Qi Li [a,b], Yang Zhang [a]

[a] School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, 210023, Jiangsu, China
[b] Jiangsu Provincial Key Laboratory of Internet of Things Intelligent Perception and Computing, Nanjing, 210023, Jiangsu, China

ARTICLE INFO

Keywords:
Smart contract vulnerability detection
Prompt engineering
Large language models
Solidity

ABSTRACT

Smart contracts play a crucial role in blockchain technology, but their security remains vulnerable to various threats. While deep learning approaches have shown promise in vulnerability detection, they often require complex graph constructions that complicate the detection process. Large language models (LLMs) offer powerful code comprehension capabilities, but their direct application to vulnerability detection often yields inconsistent or unreliable results. To address these challenges, we introduce CodeSpeak, a novel framework that enhances smart contract vulnerability detection by leveraging LLM-assisted code analysis. Our approach first eliminates redundant code statements to focus on security-critical sections. We then leverage LLMs with designed domain-specific instructions that simulate security expert auditing practices. These instructions serve as intermediate representations that bridge the gap between natural language and vulnerability patterns. CodeSpeak processes this analysis by LLMs and creates structured prompt templates with these results, which are used to train a detection model. Compared to deep learning approaches, this framework offers a more intuitive solution while maintaining high detection effectiveness. Extensive experiments conducted on four types of vulnerabilities (*Reentrancy*, *Timestamp*, *Overflow/Underflow*, and *Delegatecall*) demonstrate the effectiveness of our approach. Our framework also demonstrates strong adaptability to new vulnerability types with minimal training samples, and provides a cost-effective solution for practical deployment. Moreover, a user study with developers shows CodeSpeak reduces detection time by 98.7% compared to manual analysis while maintaining superior accuracy. These improvements highlight the potential of LLM-assisted code analysis in smart contract security assessment.

1. Introduction

Smart contracts are programs that usually run on blockchains without a trusted third party. Due to their traceability, immutability, and automatic execution capabilities, smart contracts are widely employed across various sectors (Mariano et al., 2020; Singh et al., 2020), such as finance, healthcare, and supply chain management. These programs cannot be altered once deployed, which makes any existing vulnerabilities permanent and potentially more exploitable by attackers (Wan et al., 2021; Ren et al., 2021; Cui et al., 2022). These security vulnerabilities have resulted in numerous incidents, causing considerable economic damage (Kushwaha et al., 2022; Zhang et al., 2023a; Storhaug et al., 2023). Therefore, detecting potential vulnerabilities before deploying smart contracts to the blockchain is crucial to prevent such security breaches.

The critical need for vulnerability detection has driven extensive research efforts in smart contract security. Traditional approaches primarily rely on formal verification and symbolic execution techniques. Formal verification translates smart contract code into mathematical models to verify functional correctness and security properties (Bai et al., 2018; Abdellatif and Brousmiche, 2018). Symbolic execution explores potential execution paths by simulating various inputs while maintaining path constraints to identify vulnerabilities (Luu et al., 2016; Mueller, 2017; Torres et al., 2018; Zheng et al., 2022; David et al., 2023). While these methods provide effective analysis capabilities, their practical applicability is limited by high false-positive rates and computational overhead.

To address these limitations, researchers have explored approaches including utilizing graph neural networks (GNNs) and training models

☆ Editor: Dr. Dario Di Nucci.

* Corresponding author at: School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing, 210023, Jiangsu, China.
E-mail addresses: shuyu_chang@njupt.edu.cn (S. Chang), 1022041120@njupt.edu.cn (C. Geng), hhp@njupt.edu.cn (H. Huang), rui_wang@njupt.edu.cn (R. Wang), liqics@njupt.edu.cn (Q. Li), 1023041125@njupt.edu.cn (Y. Zhang).

[1] Equal contribution

https://doi.org/10.1016/j.jss.2025.112635
Received 29 January 2025; Received in revised form 20 August 2025; Accepted 11 September 2025
Available online 17 September 2025

on annotated vulnerability datasets (Qian et al., 2023b; Zhang et al., 2023b; Zhu et al., 2023; Cai et al., 2023; He et al., 2023; Nguyen et al., 2022). These models transform source code into graph representations with carefully designed nodes and edges that capture vulnerability-specific patterns (Zhuang et al., 2020; Liu et al., 2021, 2023; Luo et al., 2024; Li et al., 2023b; Chen et al., 2024; Cai et al., 2023). However, GNN-based approaches struggle with adaptability to new vulnerability types and require substantial effort in graph construction.

The emergence of Large Language Models (LLMs), exemplified by GPT-3.5 (Ouyang et al., 2022), presents new opportunities for software engineering tasks. Trained on vast repositories of code and documentation, these models exhibit remarkable capabilities in code comprehension and analysis (Kojima et al., 2022). Researchers have successfully employed LLMs to advance various software engineering applications, including automated code repair (Jiang et al., 2023) and vulnerability detection (David et al., 2023; Sun et al., 2024a). Through prompt engineering (Kang et al., 2024), LLMs can effectively harness their knowledge of programming patterns to perform complex tasks that traditionally demand human expertise. In the context of smart contract vulnerability detection, LLMs offer a promising paradigm shift (Sun et al., 2024a). Unlike GNNs that depend on explicit structural representations, LLMs excel at simultaneously processing natural language elements and complex code patterns. This capability enables LLMs to comprehensively analyze code syntax and semantics, inferring potential execution paths and identifying vulnerabilities without actual code execution. However, our investigations reveal that directly applying LLMs to vulnerability detection often leads to inconsistent results. It is primarily due to their tendency to generate hallucinations (Li et al., 2023a) or make unfounded assumptions about code behavior.

To address these challenges, we propose CodeSpeak, a novel framework that enhances smart contract vulnerability detection through LLM-assisted code analysis. Our framework consists of four key phases: data preparation, LLM-assisted code analysis, knowledge integration, and vulnerability detection. In the preparation phase, we eliminate redundant code statements to focus the analysis on security-critical sections, improving the efficiency of vulnerability detection. For the LLM-assisted code analysis, we design a set of domain-specific instructions that systematically mimic the process of manual security auditing. These instructions contain specific vulnerability-related conditions, effectively transforming a general-purpose LLM into a specialized expert for smart contract security analysis. CodeSpeak combines processed code with these carefully crafted instructions and leverages LLMs to analyze code structures and potential vulnerability patterns. Unlike traditional methods that rely on complex graph representations, this approach offers a more intuitive and flexible solution for vulnerability detection. Knowledge integration then constructs structured prompt templates based on the security assessment results from LLMs. In the final detection phase, these templates, incorporating knowledge derived from the LLM-assisted code analysis, are used to train a detection model through prompt tuning. We validate our framework through extensive experiments on four prevalent types of Ethereum smart contract vulnerabilities: *Reentrancy*, *Timestamp*, *Overflow/Underflow*, and *Delegatecall*. The experimental results demonstrate that CodeSpeak outperforms existing state-of-the-art approaches in detection effectiveness while offering practical benefits in analysis time and cost-efficient implementation. Moreover, our approach effectively mitigates the limitations of directly applying LLMs to vulnerability detection while maintaining a streamlined analysis process.

The main contributions of our work are summarized as follows:

- We propose CodeSpeak, a novel framework that enhances smart contract vulnerability detection through LLM-assisted code analysis. Our framework offers a more intuitive solution by leveraging the natural language processing capabilities of LLMs.
- We design domain-specific instructions to guide the code analysis process that mimics the manual security auditing process to create precise vulnerability assessments.
- The framework combines LLM-generated security analysis with prompt engineering techniques. By constructing prompt templates from LLM analysis results, we effectively train a detection model that bridges the gap between analysis results and vulnerability detection.
- Extensive experiments are conducted on four types of Ethereum smart contract vulnerabilities, along with user studies involving smart contract developers. The results demonstrate that CodeSpeak outperforms state-of-the-art approaches in performance and reduces detection time by 98.7% compared to manual analysis, while showing strong adaptability to new vulnerability types.

Table 1
Smart contract vulnerabilities and their abbreviations based on SWC standard.

SWC ID	Full name	Abbreviation
SWC-107	Reentrancy	*Reentrancy*
SWC-116	Block values as a proxy for time	*Timestamp*
SWC-101	Integer Overflow and Underflow	*Overflow/Underflow*
SWC-112	Delegatecall to Untrusted Callee	*Delegatecall*

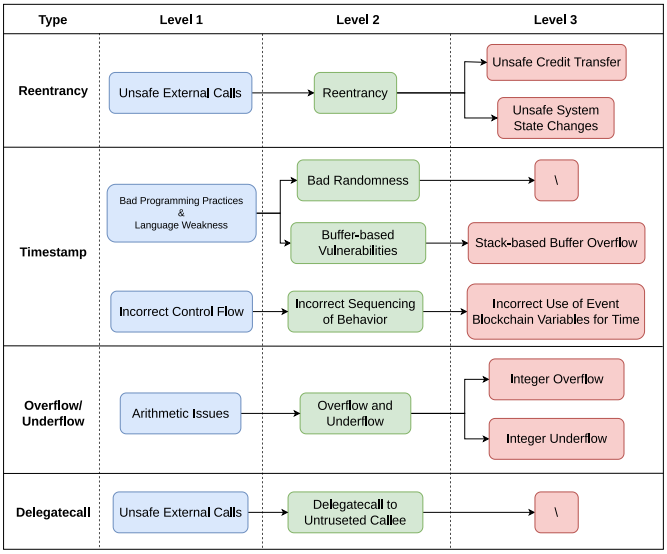

Fig. 1. Classification of the four vulnerability types within the OpenSCV taxonomy.

The remainder of this paper is organized as follows. Section 2 introduces the preliminary knowledge of this paper. We describe the details of our framework CodeSpeak in Section 3. Experiments and analysis of CodeSpeak are presented in Section 4. Section 5 describes threats to validity. Section 6 revisits a review of prior research pertinent to our work. Finally, we conclude our work in Section 7.

2. Preliminaries

Our research primarily focuses on four types of smart contract vulnerabilities. According to the vulnerability classification standards on the SWC official website,[2] the correspondence between the codes and the full names of these four vulnerabilities is shown in Table 1. For ease of presentation, we provide simplified names for the vulnerabilities. We also classify the vulnerabilities discussed in this paper according to the OpenSCV standard (Vidal et al., 2024a), with detailed categorization shown in Fig. 1.

[2] https://swcregistry.io.

These vulnerabilities display characteristics commonly found in Ethereum smart contract issues (Qian et al., 2023b; Zhong et al., 2024), leading to significant financial repercussions (Svyatkovskiy et al., 2020; Xue et al., 2020; Qian et al., 2023a). In this section, we describe the manifestations of these vulnerabilities and present motivating examples.

2.1. Smart contract vulnerabilities

Reentrancy (SWC-107). A *Reentrancy* vulnerability permits external contracts to recursively call back into functions before completing a prior invocation (Liu et al., 2021). This typically occurs during critical operations like asset transfers. In such attacks, a malicious contract re-enters the vulnerable contract mid-execution. This leads to state manipulation, allowing actions like multiple withdrawals or balance inconsistencies.

Timestamp (SWC-116). When a smart contract contains `block.timestamp`, developers often attempt to use it to trigger time-dependent events. As Ethereum is decentralized, the nodes can synchronize time within certain tolerances. Moreover, malicious miners can adjust the timestamp of their blocks within this acceptable range, especially if they can gain advantages by doing so. For instance, attackers might wait for specific timestamps to trigger contract functions that perform malicious operations.

Overflow/underflow (SWC-101). *Overflow/Underflow* happens when arithmetic operations exceed the bounds of a data type. Solidity versions before 0.8.0 handle these cases silently without exceptions. For a uint8 (range 0–255), adding 1 to 255 results in 0. Similarly, subtracting 1 from 0 yields 255. Attackers exploit these behaviors by supplying boundary values. This can lead to logical errors, unauthorized token creation, or bypass of balance checks.

Delegatecall (SWC-112). `Delegatecall` is a specialized variant of a message call, which executes code from another contract within the context of the calling contract. Importantly, it preserves the original `msg.sender` and `msg.value`. This enables a smart contract to dynamically load and execute code from a different address at runtime. However, it gives the called contract access to the storage and balance of the calling contract. Using `delegatecall` with untrusted addresses is highly dangerous. It can result in storage manipulation or complete balance theft.

2.2. Motivating example

Limitations of GNNs. Existing deep learning approaches rely on representing smart contract functions as complex graphs. Researchers construct several types of edges within the graph to effectively model these interactions. Control flow edges outline the logical structure of the code, data flow edges track how data is accessed or modified, and attack edges delineate potential paths an attacker might exploit. Each type of vulnerability demands a specific design of these graph elements, which makes the detection process complex. Furthermore, updating these graph structures becomes increasingly difficult when new vulnerabilities emerge.

Fig. 2 illustrates the construction of such graphs. The contract `LuckyETH` may have a potential *Reentrancy*, while `GameCell` is prone to timestamp dependency issues. Various program elements are categorized into distinct node types within the graph: core nodes, normal nodes, and attack nodes. Core nodes are crucial as they represent the essential calls and variables that are directly involved in vulnerabilities. For instance, in `LuckyETH`, core nodes include the `call.value` function and variables related to user balances, which are beneficial to detect *Reentrancy*. Similarly, in `GameCell`, core nodes involve calls to `block.timestamp` and associated program statements, highlighting the timestamp vulnerability. Normal nodes, by contrast, represent the

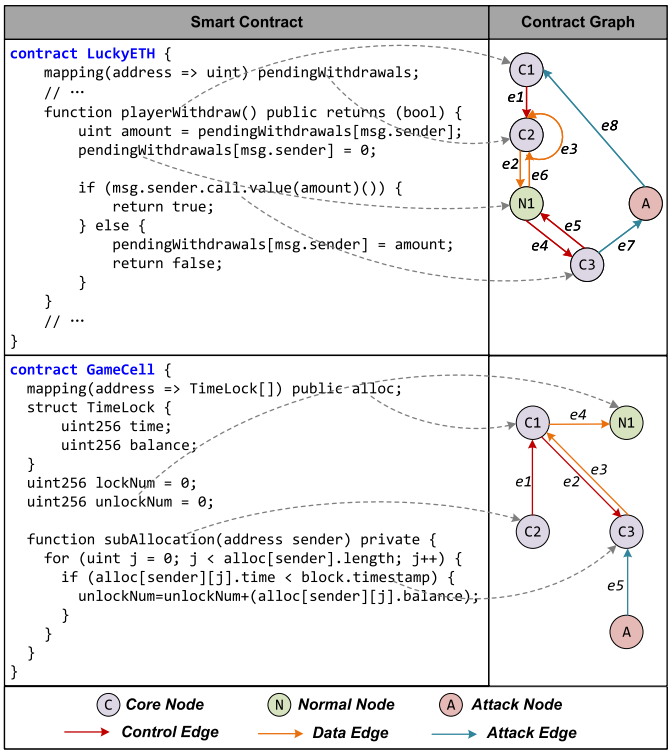

Fig. 2. Examples of contract graph construction.

less critical calls and variables that support the overall structure of the contract but are not directly involved in vulnerabilities. Meanwhile, attack nodes explicitly represent how attackers might exploit the contract. In `LuckyETH`, this includes the use of the fallback function, a special function executed when the contract receives Ether or when an unknown function is called (Atzei et al., 2017). In `GameCell`, the attack node refers to altering the value of the variable `block.timestamp`, which may result in unexpected changes to the value of `unlockNum`.

Limitations of LLMs. Recent research has explored using LLMs such as GPT-3.5 for vulnerability detection (Sun et al., 2024a). However, these models exhibit limitations, particularly in generating hallucinations, which are plausible but incorrect analysis results. Despite their ability to correctly identify vulnerabilities in certain instances, LLMs often exhibit flawed reasoning processes. Researchers (Ullah et al., 2024) have observed that LLMs produce inconsistent results for identical code inputs across multiple executions. This finding indicates that the outputs of LLMs exhibit randomness and lack consistency, which are unacceptable for security vulnerability detection tools. This inconsistency raises significant concerns regarding their reliability in practical applications. LLMs might make unfounded assumptions about code behavior or fail to recognize subtle security implications in state variable updates.

GPT-3.5 incorrectly identifies a *Reentrancy* in the smart contract shown in Fig. 3, even though the contract does not contain such a vulnerability. It predicts that an attacker could exploit the `playerWithdraw` function during an external call transfer operation. According to GPT-3.5, the attacker might repeat the transfer multiple times before the withdrawal amount is set to zero. However, the function sets the pending withdrawal amount to zero before performing the transfer, nullifying the potential for *Reentrancy* from subsequent calls. Our framework utilizes prompt engineering techniques to address the constraints mentioned above.

[Smart Contract]

```
contract LuckyETH {
    mapping(address => uint) pendingWithdrawals;
    address owner;

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    function playerWithdraw() public onlyOwner returns (bool) {
        uint amount = pendingWithdrawals[msg.sender];
        pendingWithdrawals[msg.sender] = 0;

        if (msg.sender.call.value(amount)()) {
            return true;
        } else {
            pendingWithdrawals[msg.sender] = amount;
            return false;
        }
    }
}
```

[Detection Result]
Defective

Analysis:
The player could externally call the playerWithdraw function to withdraw. Since the external call occurs before updating the pendingWithdrawals mapping, an attacker can exploit this by recursively calling the playerWithdraw function. By repeatedly calling before the withdraw amount is set to 0, the attacker can drain funds from the contract.

Fig. 3. Example of LLM-based smart contract vulnerability detection.

3. Methodology

3.1. Overview

This section presents our proposed framework, CodeSpeak, for smart contract vulnerability detection. As illustrated in Fig. 4, our framework consists of four key phases: (1) *Data Preparation*: preprocesses the smart contract code by removing redundant elements, such as comments and version numbers, to focus the analysis on the core functionality. (2) *LLM-Assisted Code Analysis*: employs specific instructions that guide LLMs to mimic the manual security auditing of smart contracts. These vulnerability-specific instructions examine code execution paths and identify potential risky statements associated with each vulnerability type. (3) *Knowledge Integration through Prompt*: constructs prompt templates to integrate the code analysis results with processed code. (4) *Vulnerability Detection*: trains a pre-trained detection model based on the prompts to identify the potential security vulnerability in smart contracts.

3.2. Data preparation

Smart contracts typically contain elements unrelated to code logic, such as function descriptions, parameter definitions, version numbers, and related links. While these elements facilitate development and maintenance, they can interfere with the analysis of code semantics during vulnerability detection. Following research (Yu et al., 2023), we preprocess the code by removing:

- Comment Text: Comments include function descriptions and parameter definitions that explain code functionality but do not participate in execution.

- Version Numbers and Related Links: These elements specify compiler versions and reference documentation but are irrelevant to the core code functionality.

We can precisely identify and eliminate these redundant elements through syntax analysis and regular expression matching. Our rationale for removing these elements stems from several practical considerations. For comments, their quality varies significantly in real-world smart contracts—many contain outdated, misleading, or entirely absent comments, making comment-based analysis unreliable. Additionally, comments may contain annotation biases or vulnerability markers that could artificially influence the model's learning process. By removing comments, we ensure that the model learns consistent patterns from code structure and semantics rather than varying comment styles or annotations. For version numbers and related links, while they provide compilation context, they do not participate in the actual code execution logic. Removing them allows the model to focus on the core vulnerability patterns within the code itself. This standardization preserves the core semantics of the smart contract code, w.r.t. function definitions, variable declarations, and critical control statements, making our approach more robust for deployment scenarios.

3.3. LLM-assisted code analysis

This phase leverages the LLM to analyze smart contract code effectively for vulnerability detection. By assigning the LLM the role of a security expert auditor, we guide the analysis process through a series of targeted questions. These questions are designed from specific vulnerabilities, which help the LLM recognize the corresponding security patterns. The final description put into the prompt template is generated based on the code analysis results for each smart contract.

The LLM learns about smart contracts during its pretraining phase on internet corpora (Devlin et al., 2019). By providing carefully designed system instructions, we can activate this domain-specific knowledge for vulnerability detection tasks (Kang et al., 2024). As shown in Fig. 5, we guide the LLM to analyze static code structure and execution logic, then evaluate the validity of statements related to the potential vulnerability. To further minimize the randomness of the responses generated by LLMs, we instruct the model to simulate multiple rounds of questioning in the background. This approach combines zero-shot chain-of-thought prompting (Kojima et al., 2022) with mimic-in-the-background prompting (Sun et al., 2024a). The LLM provides the most frequent answer from multiple background responses to reduce detection variance.

We design the statements set S for the four types of vulnerabilities respectively based on Qian et al. (2023b). The specific content of the statements is shown below, where the manifestation of each type of vulnerability is illustrated. Note that additional statements can be incorporated when needed for enhanced detection.

- *Reentrancy*. (1) The code contains a call to `call.value`. (2) The code deducts the balance after `call.value`. (3) The code uses `call.value` with an empty or zero parameter.
- *Timestamp*. (1) The code contains a call to `block.timestamp`. (2) `block.timestamp` in the code is either assigned to another variable or passed as a parameter to a conditional statement. (3) `block.timestamp` is used as a variable in a conditional statement.
- *Overflow/Underflow*. (1) There are arithmetic operations between the variables, and the arithmetic operations are constrained by the security library function. (2) The arithmetic operations and corresponding variables appear in the strict conditional statements. (3) The subtraction operation appears in the strict conditional statement (e.g., `assert`, `require`) for comparison, and the conditional statement appears before the subtraction operation.
- *Delegatecall*. (1) An invocation to `delegatecall` exists in the function. (2) The caller of `delegatecall` is the owner account.

The LLM evaluates the correctness of these statements based on the provided smart contract code and responds in JSON format. Specifically, we put the smart contract c and S into the prompt instructions

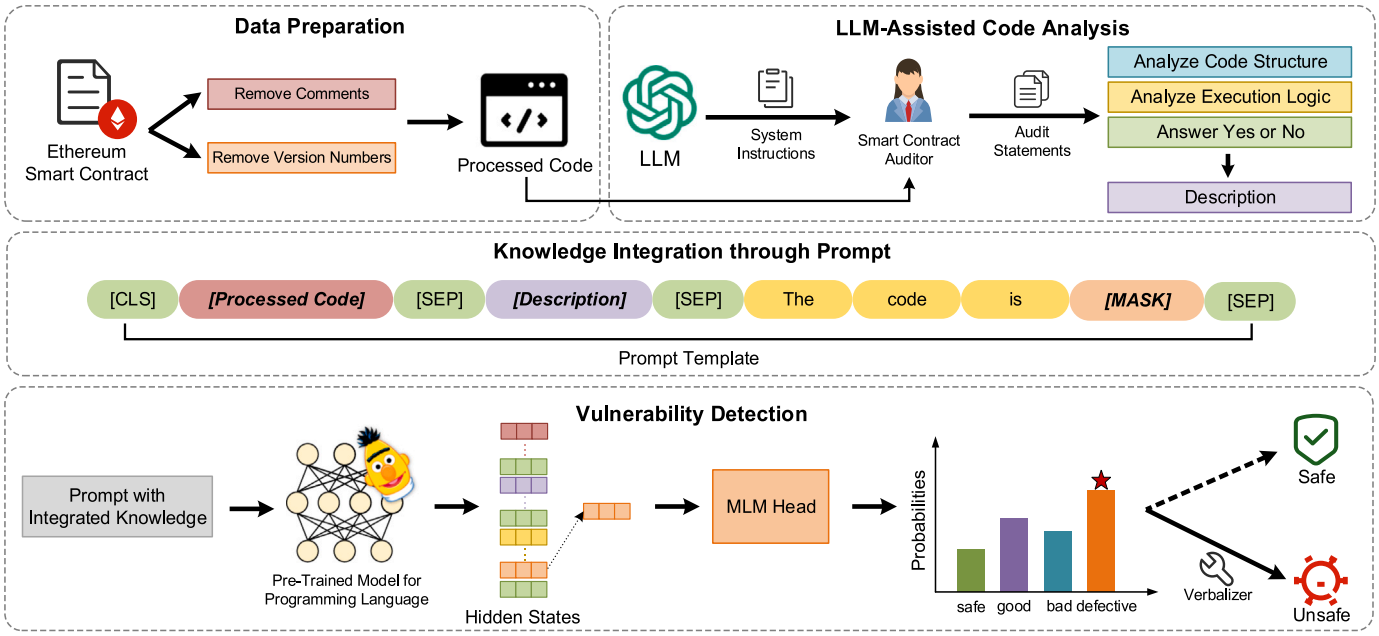

Fig. 4. Overview of our framework.

[System Prompt]
You are a smart contract auditor. You will be asked questions related to code structure and execution logic. You can mimic answering them in the background five times and provide me with the most frequently appearing answer. Please strictly follow the specified output format in the questions. Do not explain your answers.

[Smart Contract Vulnerability Auditor]
Based on the provided smart contract code, judge the statements and give the answer in the JSON format: {"1": yes or no, "2": yes or no, "3": yes or no}.

"1": ***Statement 1***
"2": ***Statement 2***
"3": ***Statement 3***

Code

Fig. 5. Instructions for LLM-assisted code analysis.

$f(\cdot)$ in Fig. 5. The LLM generates the answer set $\mathcal{A} = \{a_1, a_2, \ldots, a_N\}$ of $\mathcal{S}$ by estimating the conditional probability $\mathrm{P}_{\mathrm{LLM}}$:

$$\mathrm{P}_{\mathrm{LLM}}\left(a_n \mid f(c,\mathcal{S})\right) = \prod_{i=1}^{|a_n|} \mathrm{P}_{\mathrm{LLM}}\left(a_n^{(i)} \mid f(c,\mathcal{S}), a_n^{(<i)}\right), \tag{1}$$

where $a_n^{(i)} (n < N)$ indicates the ith token in the answer a_n, and $a_n^{(<i)}$ indicates the text before the ith token.

The generated answer set $\mathcal{A}$ is formatted in JSON and is then converted into the corresponding description of the smart contract c by a mapping function $d = k(a_1, a_2, \ldots, a_N)$. The a_n represents either "yes" or "no". When a_n is "yes", the mapping function $k(\cdot)$ appends the corresponding statement in $\mathcal{S}$ to the description d.

This LLM-assisted code analysis approach allows CodeSpeak to adapt dynamically to new and evolving vulnerabilities. As the smart contract attacks change, our framework can easily incorporate textual knowledge into the model without complex graph construction.

3.4. Knowledge integration through prompt

After transforming the analysis results from the LLM into corresponding code descriptions, our framework employs prompt tuning to integrate these descriptions into the detection model. Prompt tuning is a technique (Wang et al., 2022) that aligns the downstream task with the pre-training tasks of language models. In this context, we transform the smart contract vulnerability detection task into a pre-training task of predicting masked tokens within a structured prompt designed for programming language models.

To facilitate this integration, we embed processed code and description within a natural language prompt, which helps the pre-trained model learn the relationships between textual elements. We construct a prompt template $g(c,d)$ that encapsulates the smart contract code c and corresponding description d. The integration is achieved by arranging these elements around a masked token prediction task within the prompt template, as defined below:

$$g(c,d) = [CLS]\ c\ [SEP]\ d\ [SEP]\ The\ code\ is\ [MASK]\ [SEP]. \tag{2}$$

In this template, $[CLS]$ is a special token utilized at the beginning of the input. $[MASK]$ acts as a placeholder for the predicted word that the model outputs, allowing it to focus on filling in the optimal word based on its understanding of the surrounding context. The output represents the likely vulnerability of the smart contract code based on the presented integrated knowledge. $[SEP]$ is applied to separate text segments within the input data, clearly distinguishing between various pieces of information. This separation is crucial to help the model recognize each prompt segment as a distinct yet related unit, enhancing its ability to respond to the composite input corresponding to smart contracts. A specific example of $g(c,d)$ can be seen in Fig. 6.

This structured approach to knowledge integration enables our model to leverage its capabilities more directly for vulnerability detection without graph construction.

3.5. Vulnerability detection

To effectively detect vulnerabilities in smart contracts, we train the pre-trained programming language model through prompts. This process concentrates on predicting the vulnerability label through the $[MASK]$ token prediction task.

When a prompt $p = g(c,d)$ is input into the pre-trained model $\mathcal{M}$, we obtain the hidden states h for each token in p. The hidden state vector h_m corresponding to the $[MASK]$ token is then processed by a Masked Language Model (MLM) head network. This head network computes

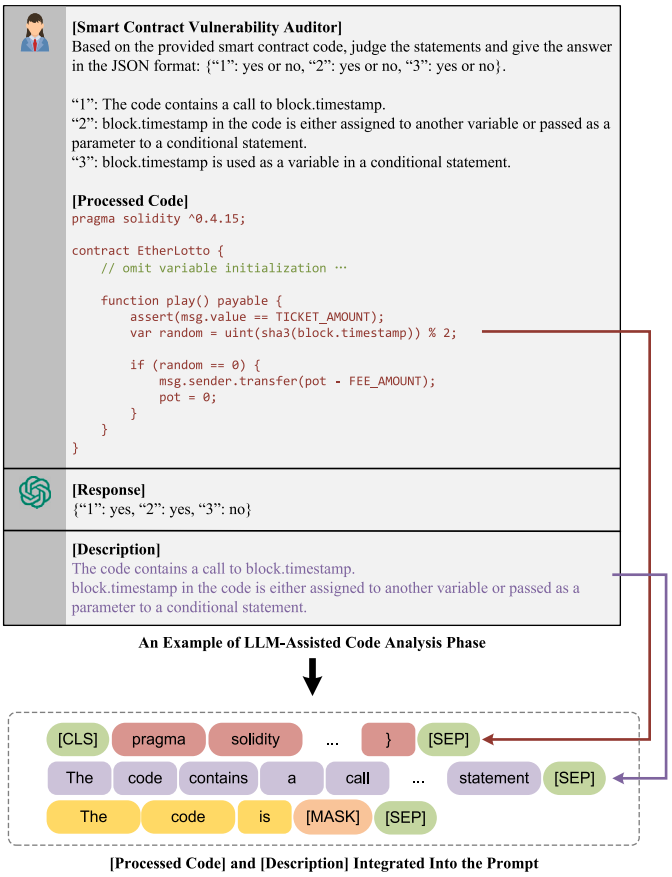

Fig. 6. A prompting example of the LLM-assisted code analysis phase and Knowledge Integration through Prompt phase.

probabilities for potential tokens at the $[MASK]$ position, identifying possible vulnerabilities. The computation follows:

$$(h_{CLS}, h_1, \ldots, h_m, \ldots, h_{SEP}) = \mathcal{M}(p), \tag{3}$$

$$o_m = \text{MLMHead}(h_m). \tag{4}$$

Here, $o_m \in \mathbb{R}^T$ represents the probability distribution over the model's vocabulary, where T denotes the vocabulary size of model $\mathcal{M}$.

To map token probabilities o_m into label probabilities, we implement a verbalizer that maps specific label words to the label space $\mathcal{Y} = \{0, 1\}$. We use $\mathcal{V}_y$ to represent the subset of vocabulary mapped to a label $y \in \mathcal{Y}$, where $y = 1$ indicates a vulnerability and $y = 0$ indicates a secure contract. The label words are defined as:

$$\mathcal{V}_y = \begin{cases} [\text{"bad"}, \text{"defective"}], & y = 1, \\ [\text{"good"}, \text{"safe"}], & y = 0. \end{cases} \tag{5}$$

For final classification, we examine the predicted probabilities of tokens $\mathrm{P}_{\mathcal{M}}([MASK] = v \mid p)$ for each label word v. The predicted label $\hat{y}_i$ is determined by averaging the scores across each label's vocabulary subset:

$$\hat{y} = \underset{y \in \mathcal{Y}}{\text{argmax}} \frac{\sum_{v \in \mathcal{V}_y} \mathrm{P}_{\mathcal{M}}([MASK] = v \mid p)}{|\mathcal{V}_y|}. \tag{6}$$

The training objective minimizes the cross-entropy loss computed from the predicted probabilities. Through this learning process, the model is tailored to distinguish between vulnerable and secure smart contracts based on our structured prompts and integrated knowledge.

4. Evaluation

4.1. Research questions

To evaluate our approach, we design experiments to answer the following research questions:

RQ1: How does CodeSpeak perform compared to existing smart contract vulnerability detection methods? In the Vulnerability Detection phase of CodeSpeak, we train a model using prompt templates that integrate LLM analysis results. We compare CodeSpeak with 10 detection methods on four common vulnerability types to explore whether our approach outperforms state-of-the-art detection tools.

RQ2: How do different design choices affect the effectiveness of CodeSpeak? We examine the Knowledge Integration through Prompt phase and the LLM-assisted code analysis phase of our framework. We investigate how different prompt templates, LLM selection, and analysis techniques impact the detection effectiveness, helping to understand the sensitivity of the framework to various design decisions.

RQ3: How does CodeSpeak perform in detecting new types of vulnerabilities? We evaluate the adaptability of CodeSpeak to new vulnerability types beyond its initial training. This tests the ability to handle emerging security threats through the flexible LLM-assisted code analysis phase, which can be extended with new statement sets for new vulnerability patterns.

RQ4: How well does CodeSpeak apply to real-world scenarios? We analyze token costs and financial expenses, and evaluate the usability of CodeSpeak with 2 smart contract developers in practical scenarios. We also present case studies demonstrating successful detections and limitations on complex real-world contracts.

The following sections detail our experimental setup and answer each research question above.

4.2. Experimental setup

Datasets. We perform experiments based on the dataset used in Yu et al. (2023), which is collected from the SmartBugs Wild Dataset (Durieux et al., 2020) and the ESC (Ethereum Smart Contracts) Dataset (Liu et al., 2021), containing *Reentrancy* and *Timestamp* vulnerabilities. The *Reentrancy* dataset includes 20,141 smart contracts with 1140 vulnerabilities. The *Timestamp* dataset contains 5010 smart contracts on Ethereum, including 2169 vulnerabilities. We also carry out experiments on another dataset (Qian et al., 2023b) containing two new types of vulnerabilities. This dataset includes *Overflow/Underflow* and *Delegatecall*. The *Overflow/Underflow* dataset contains 275 Ethereum smart contracts, including 90 vulnerabilities. The *Delegatecall* contains 196 Ethereum smart contracts, including 62 vulnerabilities. We perform a 3:1:1 split for training, validation, and testing to evaluate CodeSpeak. Each experiment is repeated three times, and the final average results are recorded.

Baselines. We evaluate CodeSpeak against existing state-of-the-art approaches, categorized into conventional detection tools and deep learning detection methods. The following tools selected as baselines for CodeSpeak are based on a recent survey paper (Vidal et al., 2024b). Since we primarily focus on LLM-based detection methods, we additionally incorporate the state-of-the-art LLM-based detection method GPTScan (Sun et al., 2024a) into baselines.

The conventional detection tools are shown as follows:

- Mythril (Mueller, 2017) employs control flow verification to detect smart contract vulnerabilities.
- Slither (David et al., 2023) translates smart contracts into an intermediate representation known as SlithIR for analysis.
- Osiris (Torres et al., 2018) integrates symbolic execution with taint analysis to identify smart contract vulnerabilities.
- Oyente (Luu et al., 2016) utilizes symbolic execution based on the control flow graph to uncover potential vulnerability patterns.

Table 2
Vulnerability detection tool output mapping to standard classifications. "\" means the corresponding tool does not support detecting the vulnerability type.

Tools	Reentrancy	Timestamp	Overflow/Underflow	Delegatecall
Mythril	External Calls, State Change External Calls	Dependence on Predictable Variables	Integer	Delegate Call to Untrusted Contract
Slither	Reentrancy-Eth, Reentrancy-No-Eth, Reentrancy-Events, Reentrancy-Unlimited-Gas	Timestamp	\	Controlled-Delegatecall, Delegatecall-Loop
Osiris	Reentrancy	\	Underflow, Overflow	\
Oyente	Re-Entrancy Vulnerability	Timestamp Dependency	Integer Underflow, Integer Overflow	\

Table 3
Effectiveness comparison (%) in four type vulnerabilities. "F1" is short for "F1-score", and "(3.5)" is short for "GPT-3.5" and "(4o)" is short for "GPT-4o". "\" means the corresponding tool does not support detecting the vulnerability type.

Method	Reentrancy			Timestamp			Overflow/Underflow			Delegatecall		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Mythril	50.24	51.79	49.74	50.00	41.72	45.49	58.62	68.00	62.96	62.07	72.30	66.80
Slither	51.97	65.41	52.60	67.25	72.38	69.72	\	\	\	52.27	70.12	59.89
Osiris	59.01	53.82	55.33	\	\	\	34.18	60.83	43.77	\	\	\
Oyente	65.63	54.11	56.44	45.16	38.44	41.53	57.55	58.05	57.80	\	\	\
TMP	74.88	83.69	79.04	74.43	85.12	79.42	69.47	70.26	69.86	70.37	68.18	69.26
AME	86.25	89.69	87.94	82.07	86.23	84.10	71.59	71.36	71.47	69.40	70.25	69.82
MHSA	77.78	51.22	61.76	60.00	69.23	64.29	83.06	79.03	80.99	44.44	61.53	51.61
CGE	85.24	87.62	86.41	87.41	88.10	87.75	80.00	85.71	82.76	78.57	68.75	73.33
PSCVFinder	97.73	90.53	93.83	94.12	92.87	93.49	83.33	78.95	81.08	87.50	70.00	77.78
GPTScan (3.5)	92.31	85.71	88.89	91.67	84.62	88.00	82.48	75.37	78.96	82.11	78.16	80.08
GPTScan (4o)	90.22	91.08	90.22	90.91	83.33	86.96	80.00	85.71	82.76	84.24	78.98	81.52
CodeSpeak	**97.93**	**93.26**	**95.54**	**97.06**	**94.29**	**96.14**	**90.21**	**87.42**	**88.79**	**87.87**	**91.17**	**89.49**

Since detection tools report vulnerabilities using their custom nomenclature, we map them to the four vulnerabilities shown in Table 2. The mapping is established through systematic analysis of tool documentation, source code examination, and validation against known vulnerability examples. When tools report vulnerability identifiers that cannot be confidently mapped based on Table 2, we conservatively treat these as detection failures to ensure evaluation integrity. For cases where tools encounter execution errors (such as "MISSING VALIDATION" without a clear vulnerability context), we conduct repeated detection attempts. If errors persist across multiple attempts, we regard this as the tool failing to detect the vulnerability. This conservative approach ensures fair comparison while preventing false attribution of detection capabilities.

We further compare our methods with deep learning detection methods, which are summarized as follows:

- TMP (Zhuang et al., 2020) transforms crucial functions and variables into core nodes within the contract graph. It uses the derived graph features for vulnerability detection.
- AME (Liu et al., 2021) combines graph neural networks with expert mode in an interpretable way.
- MHSA (Li et al., 2023b) combines knowledge-driven and data-driven algorithms to achieve smart contract vulnerability detection effectively.
- CGE (Liu et al., 2023) combines graph neural networks and customized expert rules for smart contract vulnerability detection.
- GPTScan (Sun et al., 2024a) breaks down each type of smart contract vulnerability into different scenarios and attributes, leveraging the matching capability of GPT-3.5 and GPT-4o to perform vulnerability detection.
- PSCVFinder (Yu et al., 2023) learns the representation of programming language through the pre-trained model, then exploits the capacity of the model to detect vulnerabilities.

All the baselines mentioned above are executed with their recommended hyperparameters. We adopt established evaluation metrics, including precision, recall, and F1-score, to assess the effectiveness of our model and other baselines.

Implementation details. The experiments are conducted on the CodeBERT (Feng et al., 2020) pre-trained programming language model. We set the maximum input length to 512 tokens, the maximum output length to 32 tokens, and the batch learning size to 16. An NVIDIA GeForce RTX 3090 GPU is utilized for the experiments, and the training is executed within the PyTorch framework. We utilize *gpt-3.5-turbo* as the LLM in CodeSpeak, and experiment with *gpt-4o* and *Gemini 1.0 Pro* as the LLM in the ablation study. The code of CodeSpeak is available at https://github.com/GengarSix2/CodeSpeak.

Effectiveness measures. Our experiments employ accuracy, precision, recall, and F1-score to assess the effectiveness of our approach. For binary classification tasks, the positive samples are the contract samples with vulnerabilities, and the negative samples are the safe contract samples. The correct prediction of positive samples is denoted as TP, the incorrect prediction of positive samples as FN, the correct prediction of negative samples as TN, and the incorrect prediction of negative samples as FP. The equations for calculating metrics are as follows.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}, \tag{7}$$

$$\text{Precision} = \frac{TP}{TP + FP}, \tag{8}$$

$$\text{Recall} = \frac{TP}{TP + FN}, \tag{9}$$

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{10}$$

4.3. Effectiveness comparison with existing methods (RQ1)

Comparison with conventional detection tools. To investigate the effectiveness of CodeSpeak, we compare CodeSpeak with 10 vulnerability detection methods. Firstly, we benchmark our proposed method against traditional vulnerability detection tools, including Mythril (Mueller, 2017), Slither (David et al., 2023), Osiris (Torres et al., 2018), and Oyente (Luu et al., 2016). As shown in Table 3, traditional detection tools achieve relatively low F1-scores on the four types of vulnerabilities. For detecting *Reentrancy*, Mythril and Slither only achieve

Table 4
Different prompt templates are used in the experiment. c and d indicate the smart contract code and corresponding description, respectively. $[C]$ represents the $[CLS]$ token and $[S]$ denotes the $[SEP]$ token.

Template	Prompt template
CP-1	$[C]$ c $[S]$ d $[SOFT]$ $[SOFT]$ $[MASK]$ $[S]$
CP-2	$[C]$ c $[S]$ $[SOFT]$ $[SOFT]$ d $[SOFT]$ $[MASK]$ $[S]$
CP-3	$[C]$ c $[S]$ d $[SOFT]$ $[SOFT]$ $[SOFT]$ $[MASK]$ $[S]$
DP-1	$[C]$ d $[S]$ $The\ code\ c\ is$ $[MASK]$ $[S]$
DP-2	$[C]$ c $[S]$ d $[S]$ $It\ is\ a$ $[MASK]$ $code$ $[S]$
CodeSpeak	$[C]$ c $[S]$ d $[S]$ $The\ code\ is$ $[MASK]$ $[S]$

49.74% and 52.60% F1-score, respectively. While Osiris and Oyente obtain 43.77% and 57.80% F1-score in *Overflow/Underflow*. This limited performance can be attributed to their focus on low-level instruction analysis while overlooking high-level semantic information from the source code.

Comparison with deep learning detection methods. We further compare CodeSpeak with advanced GNN-based detection methods, including TMP, AME, and CGE. GNN-based methods show improvement over traditional tools, attributed to GNNs' ability to capture complex structural information within smart contracts. However, these methods still face challenges in fully understanding the semantics of code. PSCVFinder, which utilizes the programming language model, achieves the best effectiveness among traditional deep learning detection methods.

CodeSpeak demonstrates superior performance, surpassing PSCVFinder with F1-score gains of 1.71% and 2.65% for *Reentrancy* and *Timestamp*, respectively. For *Overflow/Underflow* and *Delegatecall* with relatively small training samples, CodeSpeak outperforms PSCVFinder with F1-score gains of 7.71% and 11.71%. Unlike GNN-based methods that rely on predefined rules for graph construction, CodeSpeak leverages LLMs in comprehending code context and logic, allowing for a better analysis of potential vulnerabilities. Furthermore, CodeSpeak eliminates the need to redesign graph construction methods for different vulnerability types. This enables the model to learn and adapt to various vulnerability patterns more efficiently.

Comparison with LLM-based methods. We also compared CodeSpeak with GPTScan (Sun et al., 2024a). GPTScan is the most innovative detection method that utilizes LLMs, which breaks the vulnerability detection process into matching vulnerability scenarios and properties.

The final results show that GPTScan using GPT-3.5 achieves an F1-score of 88.89% for *Reentrancy* and 88.00% for *Timestamp*. The results indicate that LLMs have the potential to perform vulnerability detection tasks, but there is still a gap compared to state-of-the-art detection approaches. As shown in Section 2.2, LLMs may produce false positives, such as failing to recognize security policies in code. It is noteworthy that research (Sun et al., 2024a) has already demonstrated that GPT-4 does not significantly improve over GPT-3.5 in vulnerability detection tasks. The experimental results in Table 3 indicate that code detection tasks do not necessarily require more powerful GPT models. Therefore, CodeSpeak uses GPT-3.5 to implement code analysis and obtains the best precision, recall, and F1-score compared to the 10 baseline approaches.

Answer to RQ1: CodeSpeak outperforms the state-of-the-art approaches across all metrics. Compared to the best detection method, PSCVFinder, CodeSpeak shows average improvements of 2.59% in precision, 8.44% in recall, and 3.44% in F1-score.

4.4. Impact of different design choices (RQ2)

Impact of different prompt templates. To investigate the impact of different prompt templates on the effectiveness of CodeSpeak, we conduct a series of experiments. We designed various prompt templates, including both discrete prompts (DP) and continuous prompts (CP), as detailed in Table 4. It is noted that DP templates use natural language to describe tasks and CP templates use $[SOFT]$ to indicate a virtual token that needs to be optimized. $[MASK]$ denotes the label word to be predicted.

Table 5
Effectiveness comparison (%) between different prompt templates.

Template	Reentrancy			Timestamp		
	Precision	Recall	F1	Precision	Recall	F1
CodeSpeak	**97.93**	**93.26**	**95.54**	**97.06**	**94.29**	**96.14**
CP-1	95.63	92.11	93.84	96.06	92.62	94.31
CP-2	96.76	92.18	94.42	96.53	91.75	94.08
CP-3	96.43	92.98	94.68	96.58	92.09	94.28
DP-1	97.50	92.72	95.05	96.59	91.66	94.06
DP-2	97.84	93.17	95.45	96.99	94.21	95.58
Template	Overflow/Underflow			Delegatecall		
	Precision	Recall	F1	Precision	Recall	F1
CodeSpeak	**90.21**	87.42	**88.79**	87.87	**91.17**	**89.49**
CP-1	88.09	86.73	87.40	88.51	86.69	87.59
CP-2	86.02	**88.19**	87.09	87.84	87.00	87.42
CP-3	85.85	87.60	86.72	88.73	83.44	86.01
DP-1	86.44	87.66	87.05	89.99	87.63	88.79
DP-2	85.41	87.68	86.53	**90.09**	87.42	88.74

From Table 5, we can find that using different prompt templates in the knowledge integration phase has a limited impact on the effectiveness of CodeSpeak. There are differences among the prompt templates, such as the number and placement of $[SOFT]$ tokens in the CP template, the natural language text in the DP template, and the placement of c and d. However, the final metrics achieved by each template are similar, and the CodeSpeak template used by CodeSpeak achieves optimal results for both vulnerability types. The impact of prompt templates on the detection performance of CodeSpeak is minimal, suggesting that the design of these templates is not heavily restricted. Different templates can achieve strong effectiveness, allowing for flexible design and adjustment of prompts within the framework.

Contribution of LLM-assisted code analysis. We conduct comprehensive ablation studies to examine the contribution of LLM-assisted code analysis to CodeSpeak. The result is shown in Fig. 7. We use GPT-3.5, GPT-4o, and Gemini Pro to generate code analysis results, which are added as descriptions into the structured prompt template in Eq. (2). Prompt Tuning indicates that the prompt template does not contain the description d. We use CodeSpeak (Gemini Pro) to denote the implementation where Gemini Pro serves as the assisted LLM in the code analysis phase, while CodeSpeak (GPT-3.5) refers to our originally proposed framework. When LLM-assisted code analysis is removed from CodeSpeak (GPT-3.5), the F1-score decreases in all vulnerabilities. This indicates that LLM-assisted code analysis helps CodeSpeak better understand the execution process of smart contracts during prompt tuning.

To further investigate the impact of different LLMs in the code analysis phase, we compare the effectiveness of GPT-3.5, GPT-4o, and Gemini Pro using the same prompt template. It is evident that using different LLMs has a relatively minor impact on the detection effectiveness. This disparity highlights GPT-3.5's strength in code comprehension and suggests that it is the proper choice for code analysis.

By comparing the effectiveness of GPT-3.5, GPT-4o, and Gemini Pro with the same prompt template, it is found that different LLMs have a relatively minor impact on detection effectiveness. CodeSpeak finally selects GPT-3.5 as the LLM for the LLM-assisted code analysis phase.

Stability analysis of different LLMs. To quantitatively evaluate the impact of mimic-in-the-background prompting on overall system stability, we execute the entire detection pipeline multiple times on the complete test dataset. We independently repeat the testing process 10 times for

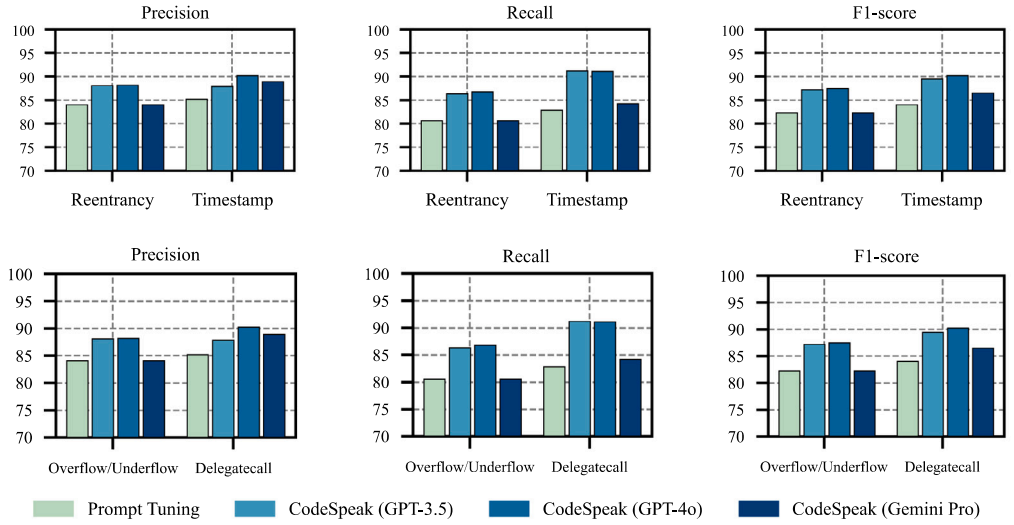

Fig. 7. Effectiveness comparison (%) between various modules of CodeSpeak.

Table 6
Performance metrics with 95% confidence intervals (%). – *w/o mimic* means tests without mimic-in-the-background prompting. Reen., Times., Overf., and Deleg. are short for Reentrancy, Timestamp, Overflow/Underflow, and Delegatecall, respectively.

Method	Reen.	Times.	Overf.	Deleg.
Precision				
CodeSpeak	97.58 ± 0.35	97.16 ± 0.33	90.10 ± 0.40	87.50 ± 0.40
– *w/o mimic*	94.20 ± 0.94	95.69 ± 0.74	89.54 ± 0.84	83.75 ± 0.62
Recall				
CodeSpeak	93.82 ± 0.33	93.68 ± 0.36	87.50 ± 0.38	90.06 ± 0.34
– *w/o mimic*	86.08 ± 1.12	92.60 ± 0.96	84.62 ± 0.73	86.46 ± 0.66
F1-score				
CodeSpeak	95.19 ± 0.50	95.88 ± 0.25	88.14 ± 0.37	89.89 ± 0.31
– *w/o mimic*	92.77 ± 0.74	94.38 ± 0.72	86.20 ± 0.63	88.10 ± 0.88

Table 7
Percentage of smart contracts with inconsistent classifications across multiple tests (%).

Method	Reen.	Times.	Overf.	Deleg.
CodeSpeak	3.11	1.99	4.72	3.78
– *w/o mimic*	5.98	6.12	9.13	8.76

Table 8
Code analysis accuracy (%) of different LLMs in the stability tests.

LLM	Reen.	Times.	Overf.	Deleg.
GPT-3.5	90.67	90.66	90.33	90.00
– *w/o mimic*	85.33	87.00	83.66	82.33
GPT-4o	91.99	89.00	90.33	90.00
– *w/o mimic*	87.66	85.00	88.33	83.33
Gemini Pro	87.33	86.33	90.66	87.00
– *w/o mimic*	81.00	80.00	81.33	85.00

all vulnerability types and record the test results, reporting standard deviations and 95% confidence intervals for all performance metrics. The results are presented in Table 6, where – *w/o mimic* indicates that the LLM does not use mimic-in-the-background prompting.

The results demonstrate that mimic-in-the-background prompting significantly enhances overall system stability. With this technique, performance metrics show substantially reduced variance across all vulnerability types. For instance, Precision variance for *Overflow/Underflow* decreases from 1.39 to 0.30, while the F1-score of *Timestamp* variance reduces from 0.99 to 0.12. Additionally, we calculate the percentage of contracts with inconsistent classifications across multiple tests, shown in Table 7. We consider a contract inconsistent if CodeSpeak fails to produce identical labels across all 10 repeated experiments. The mimic-in-the-background prompting significantly reduces the proportion of inconsistent contracts, with the average percentage decreasing by 4.09% across the four vulnerability types.

To further validate these findings and examine the stability improvements at the LLM component level, we conduct a targeted analysis on a subset of 40 smart contracts randomly selected from our dataset, with 10 cases per vulnerability type. Two independent security experts with 3+ years of Solidity development experience manually analyze the correct answers for each contract using a standardized protocol: (1) line-by-line code review, (2) execution path analysis, and (3) comparison with known vulnerability patterns from the SWC taxonomy. Inter-rater agreement achieved Cohen's $\kappa = 0.87$, indicating strong agreement.

We then instruct each LLM to perform code analysis following our prompt pattern, with the process repeated 10 times per contract to measure consistency. The LLM evaluates the correctness of the vulnerability statements presented in Section 3.3, and we consider an LLM response correct only if it is entirely consistent with the manually verified ground truth. The results are presented in Table 8. We find that all LLMs exhibit improved accuracy when utilizing the mimic-in-the-background prompting technique. GPT-4o demonstrates the best overall performance, with accuracy above 89% across all four vulnerability categories, while Gemini Pro shows the most improvement in the *Overflow/Underflow* dataset, achieving an increase of 9.33%. These results validate the effectiveness of our mimic-in-the-background prompting technique in reducing randomness in LLM outputs and ensuring greater consistency, which directly contributes to the improved end-to-end pipeline stability observed in our comprehensive analysis.

Impact of comment removal. To evaluate the effect of removing comments on our framework, we conduct experiments comparing LLM outputs with and without comments across all test datasets. Specifically, we process each smart contract in two versions: one with comments removed (our standard approach) and one with comments retained. We then instruct the LLM to analyze both versions and record whether the LLM's answers (yes/no) for each statement were identical between the two versions. As shown in Table 9, the agreement rates between

Table 9
Consistency analysis of LLM outputs with and without comments.

LLM	Reen.	Times.	Overf.	Deleg.
GPT-3.5	98.42	97.83	97.47	94.71
GPT-4o	97.34	97.11	97.86	95.35
Gemini Pro	98.12	97.22	98.27	95.85

Table 10
Analysis of inconsistent cases between with and without comments.

Analysis	Reen.	Times.	Overf.	Deleg.
Total inconsistent cases	83	38	18	12
Correct w/o comments	47 (56.6%)	12 (31.6%)	8 (44.4%)	6 (50.0%)
Correct w/ comments	35 (42.2%)	10 (26.3%)	7 (38.9%)	4 (33.3%)
Both incorrect	1 (1.2%)	16 (42.1%)	3 (16.7%)	2 (16.7%)

both settings were remarkably high, ranging from 94.71% to 98.42% across different vulnerability types and LLMs. This high consistency indicates that comments have minimal impact on the LLM's code analysis capabilities for the majority of cases.

To understand the impact on the remaining inconsistent cases where outputs differed, we perform manual analysis by having two security experts independently verify the correct answers for each vulnerability statement. We compare these ground truth answers with the LLM outputs from both versions to determine which approach produces more accurate results. Table 10 presents our findings from analyzing 151 total inconsistent cases across all vulnerability types. In cases where the LLM outputs differ between the two versions, removing comments leads to correct predictions in 73 cases compared to only 56 cases when comments are retained. This analysis reveals that removing comments provides a positive effect on LLM-assisted code analysis, particularly for *Reentrancy* vulnerability, where the improvement is most pronounced.

Answer to RQ2: Prompt templates used in the knowledge integration phase have a limited impact on the effectiveness of CodeSpeak. By comparing GPT-3.5, GPT-4o, and Gemini Pro with the same prompt template, we find that different LLMs have a relatively minor impact on detection effectiveness. Our framework demonstrates considerable flexibility in both prompt template design and LLM selection. The experiments also validate the effectiveness of our mimic-in-the-background prompting technique and comment removal choice.

4.5. Performance in detecting new types of vulnerabilities (RQ3)

To evaluate the capability of CodeSpeak in few-shot scenarios, we select three vulnerability types with limited available samples. Table 11 presents these vulnerabilities according to the SWC standard classification, including their IDs, full names, and abbreviations.

For the LLM-assisted code analysis phase, we define the following vulnerability statements for the three types of vulnerabilities:

- *Suicide*. (1) The code contains a self-destruct function. (2) The self-destruct function lacks an access modifier.
- *Transaction*. (1) Critical operations of a code are based on temporary global variables. (2) The code lacks protections for transaction order-sensitive logic.
- *Txorigin*. (1) The code contains the variable `tx.origin`. (2) The code implements authorization verification through `tx.origin`.

We source our extended dataset from HuangGai,[3] which consists entirely of real-world smart contracts. For each vulnerability type, we construct a balanced few-shot training set with four vulnerable samples and four manually patched samples. All remaining contracts were reserved for testing. For comparison, we select Slither, which supports the detection of these vulnerabilities, and GPTScan, a state-of-the-art LLM-based detection method. We exclude GNN-based methods due to their limited generalizability to novel vulnerability types.

[3] https://github.com/xf97/HuangGai.

Table 11
SWC classification of three additional vulnerabilities.

SWC ID	Full name	Abbreviation
SWC-106	Unprotected SELFDESTRUCT Instruction	*Suicide*
SWC-114	Transaction Order Dependence	*Transaction*
SWC-115	Authorization through tx.origin	*Txorigin*

Table 12
Detection performance across three new vulnerability types. Values represent detected vulnerabilities/total test cases.

Method	Suicide	Transaction	Txorigin
Slither	41/56	30/42	35/42
GPTScan	43/56	34/42	32/42
CodeSpeak	47/56	40/42	39/42

Table 12 lists the detection results for each vulnerability type. Although Slither and GPTScan showed comparable performance, CodeSpeak consistently identified more vulnerabilities in all three categories. This superior performance demonstrates the effective integration of LLM reasoning capabilities with pre-trained models in CodeSpeak. The results confirm that CodeSpeak maintains strong detection performance even in few-shot learning scenarios.

Answer to RQ3: CodeSpeak addresses diverse security vulnerabilities through targeted statements, even with minimal training data. Our experiments show it outperforms traditional tools and other LLM-based methods in few-shot scenarios.

4.6. Application to real-world scenarios (RQ4)

Token costs and financial analysis. This part evaluates the token costs and financial expenses of CodeSpeak when utilizing various LLM APIs. These costs primarily stem from the LLM-assisted code analysis phase. For the dataset used in our experiments, smart contracts contain 155 lines on average. Table 13 presents the token costs and financial expenses across all four vulnerability types, while Table 14 details the total token consumption and corresponding overall experimental costs. This also serves as a reference for the estimation of enterprise-scale deployment costs, facilitating the transition to real-world applications.

Our results reveal that GPT-3.5 maintains consistent performance across vulnerability types, consuming approximately 7800 tokens per thousand lines of code at USD 0.008 per thousand lines. Gemini Pro shows competitive costs for *Overflow/Underflow* and *Delegatecall* detection, while GPT-4o incurs 3–4 times higher financial costs despite comparable detection performance.

Human verification. To further validate the practical applicability of CodeSpeak, we conducted a controlled user study with two Solidity developers who had limited prior exposure to vulnerability detection tools. This evaluation assesses detection accuracy and efficiency on a dataset separate from our previous experiments. We select 200 previously unused smart contracts from the SmartBugs dataset, equally distributed across four vulnerability types (50 contracts per type). All contracts are manually verified to ensure ground truth. Participants analyze these contracts using three approaches: manual inspection, Mythril (a popular detection tool), and CodeSpeak.

Table 13
Token costs and financial expenses of CodeSpeak across different LLM backends. T/KL indicates tokens per thousand lines of code, and E/KL indicates expenses (USD) per thousand lines of code.

LLM	Reentrancy		Timestamp	
	T/KL	E/KL	T/KL	E/KL
GPT-3.5	7729	0.0079	7929	0.0081
GPT-4o	7802	0.0253	8049	0.0271
Gemini Pro	7834	0.0079	7929	0.0081
LLM	Overflow/Underflow		Delegatecall	
	T/KL	E/KL	T/KL	E/KL
GPT-3.5	7842	0.0079	7911	0.0082
GPT-4o	7908	0.0311	7956	0.0297
Gemini Pro	7881	0.0043	7796	0.0041

Table 14
The total token number processed by the LLMs and the corresponding overall experimental expense (USD).

LLM	Token	Expense
GPT-3.5	30,383,210	15.19
GPT-4o	30,774,571	64.67
Gemini Pro	40,552,665	7.78

Table 15
Performance comparison (%) and average detection time per contract (seconds) across four vulnerability types. User A/B represents manual detection, while Mythril A/B and CodeSpeak A/B indicate tool-assisted detection by respective users.

Method	Reentrancy		Timestamp	
	F1	Time	F1	Time
User A	82.14	271.48	81.63	282.21
User B	82.35	254.05	83.64	277.42
Mythril A	56.52	4.78	44.90	3.86
Mythril B	58.00	4.13	44.00	3.01
CodeSpeak A	88.37	3.62	88.14	3.23
CodeSpeak B	89.29	3.04	87.50	3.50
Method	Overflow/Underflow		Delegatecall	
	F1	Time	F1	Time
User A	80.85	299.42	80.85	281.42
User B	82.76	301.07	80.77	284.28
Mythril A	71.70	4.01	69.39	4.44
Mythril B	68.18	3.61	69.23	4.52
CodeSpeak A	83.72	3.88	86.21	3.96
CodeSpeak B	84.00	3.97	86.36	4.13

Table 15 presents the performance comparison across all approaches and vulnerability types. The reported times represent the average per-contract analysis duration, including all operational overheads (compilation and execution for Mythril; API invocation and model inference for CodeSpeak). The results demonstrate that CodeSpeak achieves superior detection accuracy, maintaining F1-scores above 83% across all vulnerability types. This represents an improvement over manual analysis and Mythril. Notably, the advantage of CodeSpeak is most pronounced for complex vulnerability types like *Reentrancy* and *Timestamp*, where manual and traditional tool-based detection frequently struggle. Moreover, CodeSpeak completes vulnerability detection in under 4.5 s per contract, comparable to Mythril but dramatically faster than manual analysis. This 98.7% reduction in analysis time compared to manual inspection represents an efficiency improvement for practical development workflows.

Case studies of success cases and limitations. We examine the case where CodeSpeak successfully detected vulnerabilities that other methods missed. Fig. 8 shows a contract where the `random` function generates pseudorandom numbers based on manipulable blockchain variables (`block.timestamp` and `block.number`). Both `block.timestamp` and `block.number` are public and predictable blockchain variables. This contract creates a *Timestamp* vulnerability, as miners can influence block timestamps to predict or manipulate the generated random values. Traditional tools like Oyente fail to detect this vulnerability due to limitations in analyzing complex computational relationships. These tools primarily focus on direct dependencies in their data flow analysis, making it difficult to track how `block.timestamp` ultimately influences the random number generation through multiple intermediate calculations. GNN-based methods (CGE, AME) similarly struggle, as their graph structures typically capture only direct variable relationships and cannot effectively model complex mathematical operations. GPTScan fails due to constraints in its predefined scenario templates, which cannot accommodate vulnerabilities with complex logical chains. In contrast, CodeSpeak leverages the analytical capabilities of both LLMs and pre-trained code models to accurately identify this vulnerability by tracing the complete propagation path of `block.timestamp` across lines 7–11.

We also analyze a case where CodeSpeak and other methods produced false positives. Fig. 9 presents code from a real-world Ethereum gaming platform with extensive arithmetic operations (lines 10–12). All tested methods, including CodeSpeak, incorrectly flag an *Overflow/Underflow* vulnerability in this contract. However, this contract is actually secure because it is compiled with Solidity 0.8.0, which introduces automatic overflow/underflow checks at the compiler level. The compiler of this version checks arithmetic operations that may cause integer overflow. If an operation results in overflow or underflow, an exception will be raised, and the operation will not be executed. Traditional tools like Oyente are limited by their compatibility with Solidity versions and often fail to keep pace with such updates. Similarly, learning-based methods (CGE, PSCVFinder, and GPTScan) continue to

Timestamp Vulnerability

```
1  pragma solidity ^0.4.19;
2
3  uint256 constant private salt = block.timestamp;
4
5  function random(uint Max) constant private returns (uint256 result) {
6      uint256 x = salt * 100 / Max;
7      uint256 y = salt * block.number / (salt % 5);
8      uint256 seed = block.number/ 3 + (salt % 300) + Last_Payout + y;
9      uint256 h = uint256(block.blockhash(seed));
10
11     return uint256((h / x)) % Max + 1;
12 }
```

Fig. 8. An example where CodeSpeak successfully analyzes complex variable propagation paths.

Overflow/Underflow Vulnerability

```
1  pragma solidity ^0.8.0;
2
3  function Participate(uint deposit) private {
4      //Every 10th participant gets a 10% bonus, play smart!
5      If ((players.length % 10)==0 && players.length>1 ) {
6          total_multiplier+=100;
7      }
8
9      //Core part of the game. A lot of calculations.
10     WinningPot += (deposit * PotFrac) / 1000;
11     fees += (deposit * feeFrac) / 1000;
12     Balance += (deposit * (1000 - ( feeFrac + PotFrac ))) / 1000;
13
14     if (( deposit > 1 ether ) && (deposit > players[Payout_id].payout)) {
15         uint roll = random(100);
16         if( roll % 10 == 0 ){
17             //Win the game! Bravo!
18             msg.sender.send(WinningPot);
19             WinningPot=0;
20         }
21     }
22 }
```

Fig. 9. An example of detection challenges with evolving language features.

flag vulnerabilities based on outdated detection patterns from their training data. Both categories of tools demonstrate limited adaptability to language updates.

To address this limitation, we develop an enhanced version of our framework that preserves version information as metadata during preprocessing while still removing comments. This metadata is then passed to the LLM during the analysis phase, allowing us to incorporate version-aware statements into the vulnerability detection logic. Specifically, for *Overflow/Underflow*, we add the statement in the LLM prompt: "If the smart contract's Solidity version is 0.8.0 or higher, no integer overflow/underflow vulnerability exists".

To validate this improvement, we test all misclassified *Overflow/Underflow* cases from our original experiments. Among the test contracts that are incorrectly flagged as vulnerable in our base implementation, 8 are using Solidity version 0.8.0 or higher. After applying our version-aware detection method, only 1 contract remains misclassified, reducing false positives.

This evolution demonstrates the flexibility of LLM-based approaches. When Solidity introduces new security features in future versions, we can quickly update our analysis logic through simple modifications to the LLM prompts and metadata handling, without redesigning the entire model.

Answer to RQ4: CodeSpeak offers a cost-effective vulnerability detection solution, with GPT-3.5 providing the optimal balance between performance and expense. The controlled user study confirms efficiency gains, with detection time reduced by 98.7% compared to manual analysis while maintaining superior accuracy. Our case studies demonstrate the superior ability to analyze complex code patterns while clarifying the adaptability over existing tools.

4.7. Discussion about real-world applicability

To ensure the effective and scalable deployment of CodeSpeak in diverse development environments, a combination of preprocessing techniques, resource management strategies, and seamless tool integrations is essential.

Performance characteristics and integration strategies. Based on our experimental evaluation, CodeSpeak processes smart contracts, averaging 155 lines in under 4 s per contract, making it suitable for real-time development feedback. For large-scale deployments, preprocessing with syntax parsers can identify high-risk code patterns (e.g., `call.value`, `delegatecall`) before detailed analysis, potentially reducing analysis overhead for contracts lacking vulnerability-relevant patterns. Resource scheduling systems can prioritize computational resources to critical detection tasks and release resources during idle periods to manage costs effectively.

Development environment integration. CodeSpeak can be integrated into popular development environments like Remix or VS Code through existing plugin architectures. During smart contract development, the framework can provide real-time vulnerability warnings through code highlighting, hover tooltips, and interactive guidance. Our cost analysis shows that processing ranges from $7.78 to $64.67 for complete experimental datasets, depending on LLM selection (Table 14), making it economically viable for development workflows.

Current limitations and future work. While our evaluation demonstrates CodeSpeak's effectiveness on real-world smart contracts, several important limitations must be acknowledged. Our user study with 2 developers on 200 contracts, though using authentic blockchain-deployed contracts, represents insufficient scale for validating enterprise-level usability. Critical deployment considerations, including training requirements for development teams, integration with existing security audit workflows, handling of exceptionally large contracts (1000+ lines), and latency analysis under production loads, require dedicated field studies and industry partnerships beyond our current research scope. Future work should focus on large-scale user studies with professional security auditors and comprehensive enterprise deployment validation to fully establish CodeSpeak's production readiness.

5. Threats to validity

Threats to Internal Validity. The threats to internal validity mainly lie in our implementation of baselines and CodeSpeak. To mitigate these threats, we adopted the same hyperparameters as those in the original papers and trained the models on the publicly available source code, similar to that used in the studied methods. The effectiveness of CodeSpeak may vary under different settings, such as input length and maximum output length. We configured the input length of CodeBERT (Feng et al., 2020) as 512 tokens to ensure it processes the entire code snippet. We also set the maximum output length to 32 tokens to allow the model to generate complete predicted words. Additionally, our decision to remove comments and version information during preprocessing represents a design choice that could impact detection results. While our experiments demonstrate that removing comments actually improves detection accuracy in most cases, we acknowledge that high-quality, well-maintained comments might provide valuable context in certain scenarios. Future work could explore adaptive approaches that assess comment quality and selectively incorporate reliable documentation into the analysis process.

Threats to External Validity. The primary threats to external validity stem from the dataset used in our experiments, which was created from real-world Ethereum smart contracts. However, experimental results may not generalize to all scenarios, as unknown factors in real-world environments can affect outcomes. The vulnerabilities selected for this study are common in Ethereum smart contracts, which may not represent less frequent but potentially severe vulnerabilities. Further experimentation on datasets encompassing a broader range of smart contract vulnerabilities, including less common ones, is necessary for future work.

Threats to Construct Validity. The primary threats to construct validity arise from the metrics used in our experiments. To reduce such threats, following the prior work (Yu et al., 2023), we employed three widely recognized metrics to assess effectiveness, i.e., accuracy, precision, recall, and F1-score.

6. Related work

6.1. Smart contract vulnerability detection

In recent years, security vulnerabilities in smart contracts have caused significant losses. Researchers have tried various methods to prevent malicious attackers from exploiting vulnerabilities in smart contracts. Symbolic execution is a commonly used method for detecting vulnerabilities in smart contracts. The main idea is to symbolize program inputs and execution to maintain a set of constraints for all execution paths. For example, Oyente (Luu et al., 2016) was one of the earliest smart contract vulnerability detection methods that leveraged symbolic execution. It aimed to identify vulnerabilities by analyzing the contract's control flow graph using symbolic execution. Similarly, Mythril (Mueller, 2017) used control flow to analyze code. It carried out conceptual and taint analyzes based on control flow verification to detect common vulnerability types in Ethereum smart contracts. Osiris (Torres et al., 2018) also used taint ss. It integrated symbolic execution with taint analysis to identify integer errors in smart contracts. Slither (David et al., 2023) was a static analysis framework that transforms smart contract source code into an intermediate representation known as SlithIR, which was then used to identify smart contract vulnerabilities. When detecting code with complex logic, symbolic execution does not perform well.

Deep learning networks have been extensively applied to the task of smart contract vulnerability detection. DR-GCN (Zhuang et al., 2020) converted the source code of smart contracts into a contract graph with a high level of semantic representation. Further, TMP (Zhuang et al., 2020) expanded upon the DR-GCN approach by transforming crucial functions and variables into core nodes within the contract graph. AME (Liu et al., 2021) combined the contract graph with expert patterns and cast the code into a semantic graph. CGE (Liu et al., 2023) extracted the graph feature from the normalized graph and combined the graph feature with designed expert patterns. Recently, SCVHunter (Luo et al., 2024) has proposed a new way for graph feature extraction. It allowed users to point out more important nodes in the graph freely, leveraging expert knowledge in a simpler way to aid the automatic capture of more information related to vulnerabilities. Different from combining contract graphs and expert patterns, MHSA (Li et al., 2023b) combined knowledge-driven and data-driven algorithms to achieve smart contract vulnerability detection effectively. With the development of contrastive learning, Clear (Chen et al., 2024) employed a contrastive learning model to capture the fine-grained correlation information among contracts and generate correlation labels based on the relationships between contracts.

However, it should be noted that all the GNN-based methods mentioned above predominantly require the transformation of code into graphs and feature extraction during the training phase. This process introduces additional overhead. Moreover, these methods increase the complexity of the detection process due to the need for defining graph nodes and edge types.

6.2. Large language models

The field of natural language processing has undergone notable advancements, especially with the rise of large language models (LLMs). These LLMs are characterized by their extensive parameter count and training on large-scale datasets, such as LLaMA (Cui et al., 2024), T5 (Raffel et al., 2020), and GPT (Ouyang et al., 2022). LLMs have demonstrated strong capabilities in solving reasoning tasks and can well use knowledge in the training corpus for logical reasoning and judgment. Such models have been widely used in software engineering for tasks such as code repair and vulnerability detection. For instance, the work in Jiang et al. (2023) applied code language models on automated program repair benchmarks and fine-tuned models with training data. In the field of smart contracts, researchers have tried to use LLMs to complete vulnerability detection tasks. GPTScan (Sun et al., 2024a) has effectively aligned candidate vulnerabilities with GPT-3.5 by decomposing each type of logical vulnerability into distinct scenarios and properties. All these methods mentioned above highlight the capabilities of LLMs for coding tasks. However, LLMs still have significant limitations before they can fully replace humans in code auditing. Researchers provided LLMs with vulnerability descriptions (David et al., 2023) and used them to detect flaws in source code. The final experimental results showed that LLMs tend to produce high false-positive rates, necessitating ongoing involvement from human reviewers.

During their training phase, LLMs are exposed to extensive code datasets. LLMs can leverage their latent expertise which arises from extensive pretraining on large corpora, without the need for extensive human annotations (Sun et al., 2024b). This enables them to learn about the syntax, semantics, and prevalent execution patterns of code. We designed a series of instructions specific to smart contract vulnerability detection, guiding LLMs to generate task-appropriate context. LLMs can comprehend embedded conditional statements, loop structures, and function calls within the code to be analyzed. In this paper, we explore the new possibilities of LLMs in the task of detecting vulnerabilities in smart contracts, leveraging the latent knowledge embedded in LLMs (Kang et al., 2024).

7. Conclusion

In this paper, we proposed a smart contract vulnerability detection framework named CodeSpeak, which enhances smart contract vulnerability detection by leveraging LLM-assisted code analysis. CodeSpeak combines processed code with carefully crafted instructions and utilizes the LLM to analyze code structures and identify potential vulnerability patterns. The framework further processes the analysis result of the LLM and creates the structured prompt template, which is used to train a detection model. To validate the effectiveness of CodeSpeak, we conducted experiments across datasets containing four types of smart contract vulnerabilities. The experimental results demonstrated that CodeSpeak outperforms existing state-of-the-art approaches in detecting vulnerabilities. In future work, we plan to further evaluate the effectiveness of our framework in few-shot scenarios and investigate its applicability to additional types of smart contract vulnerabilities.

CRediT authorship contribution statement

Shuyu Chang: Writing – original draft, Methodology, Formal analysis, Conceptualization. **Chen Geng:** Writing – original draft, Validation, Software, Data curation. **Haiping Huang:** Writing – review & editing, Supervision, Project administration, Conceptualization. **Rui Wang:** Writing – review & editing, Methodology. **Qi Li:** Writing – review & editing, Supervision. **Yang Zhang:** Software, Data curation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors thank the editors and anonymous reviewers for their insightful comments and suggestions, which can substantially improve the quality of this work. This work was supported by the National Natural Science Foundation of China (Grant Nos. 62072252, 62372250), the Open Fund of Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation (Grant No. TK224013), and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (Grant Nos. KYCX23_1077, KYCX24_1234).

Data availability

Data will be made available on request. The code will be made publicly available upon paper acceptance.

References

Abdellatif, T., Brousmiche, K., 2018. Formal verification of smart contracts based on users and blockchain behaviors models. In: Proceedings of the 9th International Conference on New Technologies, Mobility and Security. IEEE, pp. 1–5.

Atzei, N., Bartoletti, M., Cimoli, T., 2017. A survey of attacks on ethereum smart contracts (SoK). In: Proceedings of the 6th International Conference on Principles of Security and Trust. Vol. 10204, Springer, pp. 164–186.

Bai, X., Cheng, Z., Duan, Z., Hu, K., 2018. Formal modeling and verification of smart contracts. In: Proceedings of the 7th International Conference on Software and Computer Applications. ACM, pp. 322–326.

Cai, J., Li, B., Zhang, J., Sun, X., Chen, B., 2023. Extended abstract of combine sliced joint graph with graph neural networks for smart contract vulnerability detection. In: Proceedings of the 30th IEEE International Conference on Software Analysis, Evolution and Reengineering. IEEE, pp. 851–852.

Chen, Y., Sun, Z., Gong, Z., Hao, D., 2024. Improving smart contract security with contrastive learning-based vulnerability detection. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. ACM, pp. 156:1–156:11.

Cui, Y., Yang, Z., Yao, X., 2024. Efficient and effective text encoding for Chinese LLaMA and Alpaca. arXiv:2304.08177.

Cui, S., Zhao, G., Gao, Y., Tavu, T., Huang, J., 2022. VRust: Automated vulnerability detection for solana smart contracts. In: Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 639–652.

David, I., Zhou, L., Qin, K., Song, D., Cavallaro, L., Gervais, A., 2023. Do you still need a manual smart contract audit?. arXiv:2306.12338.

Devlin, J., Chang, M., Lee, K., Toutanova, K., 2019. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 17th North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics, pp. 4171–4186.

Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P., 2020. Empirical review of automated analysis tools on 47, 587 ethereum smart contracts. In: Proceedings of the 42nd International Conference on Software Engineering. ACM, pp. 530–541.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, pp. 1536–1547.

He, L., Zhao, X., Wang, Y., Yang, J., Sun, X., 2023. GraphSA: Smart contract vulnerability detection combining graph neural networks and static analysis. In: Proceedings of the 26th European Conference on Artificial Intelligence. Vol. 372, IOS Press, pp. 1020–1027.

Jiang, N., Liu, K., Lutellier, T., Tan, L., 2023. Impact of code language models on automated program repair. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering. IEEE, pp. 1430–1442.

Kang, J., Luo, H., Zhu, Y., and, J.A.H., 2024. Self-specialization: Uncovering latent expertise within large language models. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, pp. 2681–2706.

Kojima, T., (Shane), S.G., Reid, M., Matsuo, Y., Iwasawa, Y., 2022. Large language models are zero-shot reasoners. In: Proceedings of the 36th Annual Conference on Neural Information Processing Systems. pp. 22199–22213.

Kushwaha, S.S., Joshi, S., Singh, D., Kaur, M., Lee, H., 2022. Systematic review of security vulnerabilities in ethereum blockchain smart contract. IEEE Access 10, 6605–6621.

Li, J., Cheng, X., Zhao, X., Nie, J., Wen, J., 2023a. HaluEval: A large-scale hallucination evaluation benchmark for large language models. In: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, pp. 6449–6464.

Li, M., Ren, X., Fu, H., Li, Z., Sun, J., 2023b. ConvMHSA-SCVD: Enhancing smart contract vulnerability detection through a knowledge-driven and data-driven framework. In: Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering. IEEE, pp. 578–589.

Liu, Z., Qian, P., Wang, X., Zhu, L., He, Q., Ji, S., 2021. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. In: Proceedings of the 30th International Joint Conference on Artificial Intelligence. ijcai.org, pp. 2751–2759.

Liu, Z., Qian, P., Wang, X., Zhuang, Y., Qiu, L., Wang, X., 2023. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Trans. Knowl. Data Eng. 35 (2), 1296–1310.

Luo, F., Luo, R., Chen, T., Qiao, A., He, Z., Song, S., Jiang, Y., Li, S., 2024. SCVHunter: Smart contract vulnerability detection based on heterogeneous graph attention network. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. ACM, pp. 170:1–170:13.

Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A., 2016. Making smart contracts smarter. In: Proceedings of the ACM on Web Conference 2024. ACM, pp. 254–269.

Mariano, B., Chen, Y., Feng, Y., Lahiri, S.K., Dillig, I., 2020. Demystifying loops in smart contracts. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 262–274.

Mueller, B., 2017. A framework for bug hunting on the ethereum blockchain. https://github.com/ConsenSys/mythril.

Nguyen, H.H., Nguyen, N., Doan, H., Ahmadi, Z., Doan, T., Jiang, L., 2022. MANDO-GURU: vulnerability detection for smart contract source code by heterogeneous graph embeddings. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp. 1736–1740.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al., 2022. Training language models to follow instructions with human feedback. In: Proceedings of the 36th Annual Conference on Neural Information Processing Systems. pp. 27730–27744.

Qian, P., He, J., Lu, L., Wu, S., Lu, Z., Wu, L., Zhou, Y., He, Q., 2023a. Demystifying random number in ethereum smart contract: Taxonomy, vulnerability identification, and attack detection. IEEE Trans. Softw. Eng. 49 (7), 3793–3810.

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

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J., 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. 21 (140), 1–67.

Ren, M., Ma, F., Yin, Z., Fu, Y., Li, H., Chang, W., Jiang, Y., 2021. Making smart contract development more secure and easier. In: Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp. 1360–1370.

Singh, A., Parizi, R.M., Zhang, Q., Choo, K.R., Dehghantanha, A., 2020. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. Comput. Secur. 88, 101654.

Storhaug, A., Li, J., Hu, T., 2023. Efficient avoidance of vulnerabilities in auto-completed smart contract code using vulnerability-constrained decoding. In: Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering. IEEE, pp. 683–693.

Sun, Z., Shen, Y., Zhou, Q., and, H.Z., 2024b. Principle-driven self-alignment of language models from scratch with minimal human supervision. Adv. Neural Inf. Process. Syst. 36, 2511–2565.

Sun, Y., Wu, D., Xue, Y., Liu, H., Wang, H., Xu, Z., Xie, X., Liu, Y., 2024a. GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. ACM, pp. 166:1–166:13.

Svyatkovskiy, A., Deng, S.K., Fu, S., Sundaresan, N., 2020. IntelliCode compose: code generation using transformer. In: Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp. 1433–1443.

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

Ullah, S., Han, M., Pujar, S., Pearce, H., 2024. LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In: Proceedings of the 45th Symposium on Security and Privacy. IEEE, pp. 862–880.

Vidal, F.R., Ivaki, N., Laranjeiro, N., 2024a. OpenSCV: an open hierarchical taxonomy for smart contract vulnerabilities. Empir. Softw. Eng. 29 (4), 101.

Vidal, F.R., Ivaki, N., Laranjeiro, N., 2024b. Vulnerability detection techniques for smart contracts: A systematic literature review. J. Syst. Softw. 217, 112160.

Wan, Z., Xia, X., Lo, D., Chen, J., Luo, X., Yang, X., 2021. Smart contract security: a practitioners' perspective. In: Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering. IEEE, pp. 1410–1422.

Wang, C., Yang, Y., Gao, C., Peng, Y., Zhang, H., Lyu, M.R., 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, pp. 382–394.

Xue, Y., Ma, M., Lin, Y., Sui, Y., Ye, J., Peng, T., 2020. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp. 1029–1040.

Yu, L., Lu, J., Liu, X., and, L.Y., 2023. PSCVFinder: A prompt-tuning based framework for smart contract vulnerability detection. In: Proceedings of the 34th IEEE International Symposium on Software Reliability. IEEE, pp. 556–567.

Zhang, Y., Ma, J., Liu, X., Ye, G., Jin, Q., Ma, J., Zhou, Q., 2023b. An efficient smart contract vulnerability detector based on semantic contract graphs using approximate graph matching. IEEE Internet Things J. 10 (24), 21431–21442.

Zhang, W., Wei, L., Cheung, S., Liu, Y., Li, S., Liu, L., Lyu, M.R., 2023a. Combatting front-running in smart contracts: Attack mining, benchmark construction and vulnerability detector evaluation. IEEE Trans. Softw. Eng. 49 (6), 3630–3646.

Zheng, P., Zheng, Z., Luo, X., 2022. Park: accelerating smart contract vulnerability detection via parallel-fork symbolic execution. In: Proceedings of the 31st ACM International Symposium on Software Testing and Analysis. ACM, pp. 740–751.

Zhong, Z., Zheng, Z., Dai, H., Xue, Q., Chen, J., Nan, Y., 2024. PrettySmart: Detecting permission re-delegation vulnerability for token behaviors in smart contracts. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. ACM, pp. 168:1–168:12.

Zhu, H., Yang, K., Wang, L., Xu, Z., Sheng, V.S., 2023. GraBit: A sequential model-based framework for smart contract vulnerability detection. In: Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering. IEEE, pp. 568–577.

Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., He, Q., 2020. Smart contract vulnerability detection using graph neural networks. In: Proceedings of the 29th International Conference on International Joint Conferences on Artificial Intelligence. pp. 3283–3290.