Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.061185 Published Online: 10 March 2025

ARTICLE

Smart Contract Vulnerability Detection Using Large Language Models and Graph Structural Analysis

Ra-Yeon Choi[1], Yeji Song[2], Minsoo Jang[1], Taekyung Kim[3], Jinhyun Ahn[4,*] and Dong-Hyuk Im[5,*]

[1]Department of Artificial Intelligence Application, Kwangwoon University, Seoul, 01897, Republic of Korea
[2]Department of Artificial Intelligence Convergence, Kwangwoon University, Seoul, 01897, Republic of Korea
[3]Department of Big Data Analytics, KyungHee University, Seoul, 02447, Republic of Korea
[4]Department of Management Information Systems, Jeju National University, Jeju, 63243, Republic of Korea
[5]School of Information Convergence, Kwangwoon University, Seoul, 01897, Republic of Korea

*Corresponding Authors: Jinhyun Ahn. Email: jha@jejunu.ac.kr; Dong-Hyuk Im. Email: dhim@kw.ac.kr

Received: 19 November 2024; Accepted: 12 February 2025

ABSTRACT: Smart contracts are self-executing programs on blockchains that manage complex business logic with transparency and integrity. However, their immutability after deployment makes programming errors particularly critical, as such errors can be exploited to compromise blockchain security. Existing vulnerability detection methods often rely on fixed rules or target specific vulnerabilities, limiting their scalability and adaptability to diverse smart contract scenarios. Furthermore, natural language processing approaches for source code analysis frequently fail to capture program flow, which is essential for identifying structural vulnerabilities. To address these limitations, we propose a novel model that integrates textual and structural information for smart contract vulnerability detection. Our approach employs the CodeBERT NLP model for textual analysis, augmented with structural insights derived from control flow graphs created using the abstract syntax tree and opcode of smart contracts. Each graph node is embedded using Sent2Vec, and centrality analysis is applied to highlight critical paths and nodes within the code. The extracted features are normalized and combined into a prompt for a large language model to detect vulnerabilities effectivel. Experimental results demonstrate the superiority of our model, achieving an accuracy of 86.70%, a recall of 84.87%, a precision of 85.24%, and an F1-score of 84.46%. These outcomes surpass existing methods, including CodeBERT alone (accuracy: 81.26%, F1-score: 79.84%) and CodeBERT combined with abstract syntax tree analysis (accuracy: 83.48%, F1-score: 79.65%). The findings underscore the effectiveness of incorporating graph structural information alongside text-based analysis, offering improved scalability and performance in detecting diverse vulnerabilities.

KEYWORDS: Blockchain; smart contract; vulnerability detection; large language model

1 Introduction

Blockchain technology, particularly its smart contract functionality, has seen widespread adoption in various applications, including crowdfunding and asset management systems [1–5]. A smart contract is a self-executing program that operates on a blockchain, automating contract execution without the need for intermediaries. Despite the advantages of immutability and transparency, which enhance trust, these very properties also introduce critical risks. Specifically, once deployed, smart contracts are immutable, meaning that any discovered vulnerabilities remain permanently embedded in the code. This immutability transforms minor coding errors into major security risks, making smart contracts attractive targets for malicious actors. The severity of such risks is underscored by real-world incidents. For instance, in the 2020 'Lendf.me' attack,

vulnerabilities in deployed smart contracts were exploited, resulting in significant financial losses [6]. These incidents highlight the limitations of existing vulnerability detection mechanisms, which often fail to provide comprehensive and accurate analysis before deployment. Consequently, there is an urgent need for advanced detection models capable of identifying vulnerabilities pre-deployment to mitigate such risks. To enhance blockchain security, several frameworks have been proposed, including peer-to-peer multiparty transaction schemes and accountable data transmission mechanisms in IoT (Internet of Things) networks [7,8]. These studies provide valuable insights into potential vulnerabilities within blockchain systems and emphasize the necessity for more sophisticated detection frameworks, such as the one proposed in this paper.

This paper introduces a novel approach that combines graph-based centrality analysis and large language models for accurate smart contract vulnerability detection. Traditional verification methods, such as manual code review, are inefficient and error-prone due to the complexity and scale of smart contract code. In response, researchers have developed automated approaches leveraging Natural Language Processing (NLP) and deep learning. While text-based methods have been useful, they often fail to capture the full semantics of smart contract code, particularly its structural and logical dependencies [9–11].

To address these limitations, researchers have explored graph-based representations of smart contract code, such as Abstract Syntax Trees (AST) and Control Flow Graphs (CFG), which better preserve the structural and control flow aspects of the code [12,13]. While these methods have shown promise, challenges remain. Notably, Graph Neural Networks (GNNs), which are widely used for graph-based analysis, struggle to capture long-range dependencies within the code's structure due to issues such as over-smoothing and vanishing gradients in deep network architectures [14–18].

Recent advances in large language models (LLMs), particularly Generative Pretrained Transformers (GPT), have revolutionized numerous fields, including software vulnerability detection [19,20]. These models offer rich contextual understanding and robust learning capabilities. Hybrid approaches that combine LLMs with graph-based analysis methods are emerging as a promising solution to enhance the precision of vulnerability detection models [21–23]. However, applying these models to smart contracts, particularly those written in Solidity, presents unique challenges due to the distinct syntax and structure of the language.

In this paper, we propose a novel LLM-based vulnerability detection model specifically designed for smart contracts. Our approach integrates the strengths of LLMs and graph-based analysis methods to detect vulnerabilities in Solidity-based smart contracts. The key contributions of this work are as follows:

- We utilize the CodeBERT model [24] to analyze smart contract source code, incorporating both contextual and structural features.
- We apply two learning models, AST and CFG, to detect vulnerabilities in smart contracts. Using three centrality analyses, we effectively represent the structural information of the graphs, along with the behavior and control flow of the source code.
- We conduct experiments to evaluate the effectiveness of the proposed approach. Specifically, we use two types of embeddings CodeBERT token embeddings and graph embeddings which are applied separately to the smart contract source code and combined using an effective integration method.

The remainder of this paper is organized as follows. Section 2 provides background information on blockchain and smart contracts, the Solidity language, the compilation of Solidity code, smart contract vulnerabilities, and AST and CFG. In Section 3, we introduce the proposed method for detecting smart contract vulnerabilities. Experimental data, the experimental environment, and results are presented in Section 4. Finally, Section 5 concludes the paper and offers suggestions for future research.

2 Related Works

2.1 Blockchain and Smart Contracts

A blockchain is a transparent and decentralized ledger of transactions. Unlike centralized systems, where a single entity controls the data, all participants in a blockchain share the same ledger. Instead of modifying existing data during updates, new data are appended using a consensus mechanism to ensure the ledger remains up to date. Transactions are processed and stored in blocks, which are sequentially linked to form a chain. Each block typically contains a timestamp, transaction history, and a hash value, although the specific information may vary depending on the blockchain network [25,26]. Key features of blockchain technology include anonymity, decentralization, and immutability [27].

Smart contracts, first proposed by Szabo in the 1990s [28], are programs executed on blockchains. Ethereum, one of the most prominent blockchain platforms, has become widely used for smart contract development. Developers on the Ethereum platform write smart contracts in Solidity, a high-level programming language, which is then compiled into stack-based bytecodes for execution on the Ethereum Virtual Machine (EVM) [29].

2.2 Solidity Language and Compiling Solidity

Solidity is an object-oriented, high-level programming language specifically designed for writing smart contracts. It supports various functions and control structures essential for smart contract development [30]. On the Ethereum blockchain [30], Solidity enables the definition of programmable functions and state variables, similar to those in traditional high-level programming languages. These state variables are updated based on the execution of functions within the contract. To deploy a Solidity-based smart contract on Ethereum, the source code is compiled into Ethereum bytecode, which can be executed on the EVM. This bytecode consists of low-level instructions, referred to as opcodes, that perform specific tasks, such as accessing memory, retrieving information, or interacting with other contracts [31]. Once deployed, each contract is assigned a unique address on the blockchain. Ethereum nodes execute the bytecode to process transactions and maintain network consensus.

2.3 Smart Contract Vulnerabilities

Flaws in smart contract code can lead to exploitable vulnerabilities, resulting in blockchain-based attacks [31]. This study focuses on six well-known vulnerabilities [32].

2.3.1 Integer Overflow and Underflow

In Solidity, In Solidity, integer types have predefined maximum and minimum values. Overflow occurs when an integer value exceeds its maximum limit, while underflow happens when a value falls below its minimum limit. These issues can result in unintended behavior and are critical vulnerabilities in smart contract development. Solidity version 0.8 and above includes built-in overflow checks for arithmetic operations, automatically reverting transactions if an overflow or underflow occurs. For older versions, the SafeMath library provides explicit checks for these issues. Using these features ensures contract integrity and mitigates risks of integer overflows and underflows.

2.3.2 Transaction-Ordering Dependence (TOD)

Blockchain blocks contain sets of transactions. The state of the smart contract intended to be called by a user may differ from its actual state when executed, as the blockchain state is updated. Miners determine the order of transactions, which can lead to vulnerabilities if the update order changes. Transaction-ordering

dependence arises when a contract's behavior is affected by such changes in transaction order [33]. To mitigate transaction-ordering dependence, contracts should avoid relying on transaction sequences. A commit-reveal scheme, where users submit a commitment first and reveal it later, reduces miner manipulation risks. Using non-miner-dependent randomness, such as cryptographic methods or external oracles, further enhances contract fairness and integrity.

2.3.3 Timestamp Dependence

This vulnerability occurs when a contract relies on block timestamps to execute critical operations [33]. Miners set block timestamps during mining and can manipulate this value by up to 900 s. Contracts that depend on timestamps for key functionalities are susceptible to exploitation by miners. To address timestamp dependence, avoid using block timestamps for critical decisions. For time-sensitive operations, use alternatives like block numbers or add checks and constraints to limit exploitation risks. Thresholds or ranges can reduce miners' ability to manipulate outcomes, enhancing contract robustness against timestamp vulnerabilities.

2.3.4 Mishandled Exception (Call Stack's Depth Attack)

When the call stack depth exceeds a predefined threshold, the contract is terminated, and the state is reverted, returning a 'false' value. To prevent exploitation, caller contracts must handle exceptions properly by verifying the successful execution of the call. To prevent exploitation, ensure caller contracts handle exceptions by verifying the success of low-level calls (e.g., call or send) and using require or assert statements to enforce these checks.

2.3.5 Reentrancy Vulnerability

Reentrancy vulnerabilities occur when a smart contract calls another contract and delays its execution until the call is completed. If a smart contract has this vulnerability, attackers can exploit it by repeatedly executing a code segment, akin to a recursive function call; to prevent this, use the checks-effects-interactions pattern: perform checks (e.g., input validation, authorization) first, update the contract state before external calls, avoid calling untrusted external contracts when possible, and apply protections like Solidity's ReentrancyGuard modifier.

2.4 AST and CFG

An AST is a hierarchical representation of the structure of source code written in a programming language. It provides a tree-like abstraction where each node signifies a specific code construct, focusing on the logical structure of the code rather than its exact syntax. By simplifying syntactic details, the AST enables analyses such as code optimization and error detection. Parsers typically generate the AST during the compilation process, and additional semantic details may be incorporated through contextual analysis. ASTs play a vital role in program analysis, transformation, and code generation systems [34]. In the context of this study, ASTs are generated using tree-sitter-solidity, which transforms Solidity smart contract source code into a structured tree. This tree represents key constructs, such as functions, variables, and control structures, forming the foundation for identifying logical relationships within the code.

A CFG is a directed graph represented as G = (V, E), where V = $\{v_1, v_2, \ldots, v_n\}$ denotes the set of vertices, and E = $\{(v_i, v_j), (v_k, v_l), \ldots\}$ denotes the set of edges In a CFG, each vertex represents a basic block, which is a sequence of program instructions. Each basic block has an entry point, corresponding to the first executed instruction, and an exit point, corresponding to the last executed instruction. The directional edges in the

graph illustrate the control flow paths [35]. During execution on the EVM, control flow in the compiled smart contract source code is managed via the stack because the EVM lacks the concept of functions [12]. Instead, all operations are handled using jumps, with each valid jump targeting the destination specified by the 'jumpdest' instruction [35]. This characteristic allows bytecode to be segmented into fragments, which can then be transformed into basic CFG blocks. However, determining the destination of a jump is challenging due to the absence of explicit arguments in the EVM jump instruction [36]. EtherSolve can be employed to extract CFGs from Ethereum smart contract bytecodes [12]. It utilizes a static analysis technique known as symbolic stack execution to dynamically calculate and infer the destinations of jump opcodes based on preceding code segments. In this study, CFG extraction is performed using EtherSolve, as described in Section 3.

3 Methods

The framework in this study comprises three modules: code extraction, graph structure extraction, and vulnerability detection, as illustrated in Fig. 1. Each module is detailed in the following subsections.

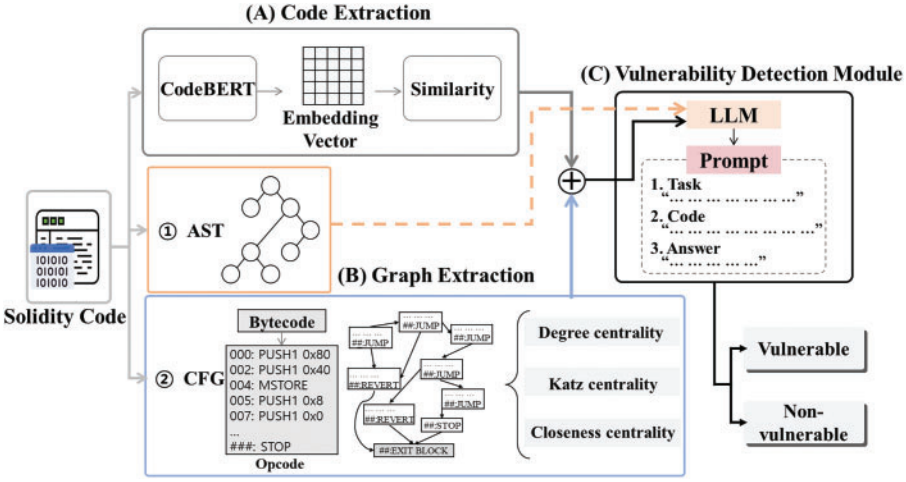

Figure 1: General architecture of the proposed framework, comprising three key modules: (A) Code Extraction Module, which preprocesses Solidity code by parsing and transforming it into meaningful representations; (B) Graph Extraction Module, which captures structural relationships using AST and CFG to generate graph-based features; and (C) Vulnerability Detection Module, which integrates these features using an LLM for vulnerability analysis

3.1 Code Extraction

The code extraction module, depicted in Fig. 1A, preprocesses the source code of smart contracts. Smart contract code, akin to natural language, has context-dependent meanings, and vulnerabilities often emerge from these dependencies. Traditional methods typically analyze code line-by-line or use static pattern matching, which can miss subtle, context-dependent vulnerabilities. To address this, NLP techniques are employed to transform the source code into token sequences, which are subsequently converted into vectors for analysis. The CodeBERT model [24], a Transformer-based model pretrained for both code and natural language, is utilized to convert code tokens into embedding vectors. It captures contextual and structural features of the code, effectively distinguishing between vulnerable and non-vulnerable code using labeled data. The similarity between the learned embedding vectors is measured in terms of cosine similarity [37]:

$$\text{sim}\,(\text{A},\text{B}) = \cos(\theta) = \frac{\text{A}\cdot\text{B}}{\|A\|\,\|B\|} \tag{1}$$

here, values closer to 1 indicate a higher degree of similarity between vectors, suggesting structural and semantic resemblance. Conversely, lower similarity scores signify differences between code samples. Cosine similarity values range from −1 to 1, where 1 indicates that vectors point in the same direction, 0 represents orthogonal vectors, and −1 signifies vectors pointing in opposite directions. However, these raw similarity scores lack intuitive interpretation. Therefore, softmax normalization is applied to transform the values into a range between 0 and 1. The softmax function, used in classification problems, normalizes vector values by converting them to exponential probabilities such that the sum of all values equals 1. It is defined as:

$$\text{softmax}\,(x_i) = \frac{e^{x_i}}{\sum_{i=1}^{N} e^{x_i}} \tag{2}$$

In this framework, the cosine similarity cos_sim(A, B) is normalized using the softmax function.

3.2 Graph Extraction

The graph structure extraction module, illustrated in Fig. 1B, combines data from ASTs and CFGs to interpret the smart contract source code. The extracted graph structures undergo centrality analyses to evaluate the importance of each node.

3.2.1 AST from Source Code

As depicted in process ① of Fig. 1, the source code is parsed into a tree structure to generate an AST by analyzing its syntax. This representation encapsulates the structural information of the code and the relationships between its components. Each element of the code (e.g., function definitions, variable declarations, control statements) is expressed as a node within the tree, providing a visual representation of the source code's structure [38]. To parse Solidity-based smart contracts, the open-source parser tool tree-sitter-solidity is utilized. This tool takes the Solidity source code as input and generates a hierarchical tree structure, where each node represents a syntactic element. Tree-sitter-solidity effectively illustrates relationships among various elements, such as functions, variables, conditional statements, and loops, within the code. This structural analysis enables the detection of vulnerabilities by examining how different parts of the contract interact and revealing potential issues related to control flow, data flow, and function call relationships. In addition, by analyzing these interactions, the AST can identify patterns indicative of security risks, such as variables with inappropriate scopes or functions lacking access control. In traditional approaches, vulnerabilities are often detected through static analysis or pattern matching techniques that focus solely on the code's surface-level structure. However, these methods may overlook intricate dependencies between different components of the contract, leading to false negatives or missed vulnerabilities. Our approach, by integrating the analysis of both AST and control flow, provides a more comprehensive view of the contract's architecture. For example, vulnerabilities like reentrancy attacks can be detected by analyzing specific patterns, such as "msg.sender" interactions, which are often related to external calls that can trigger unexpected behavior. Additionally, AST analysis can reveal potential timestamp dependencies by identifying functions that rely on block timestamps, highlighting areas where unpredictable values might influence critical operations. This allows for the identification of dependencies between external and internal functions, helping to uncover unintended vulnerabilities, such as reentrancy or incorrect state updates, that arise from the structure of the code. Such an approach complements CFG analysis by focusing on the logical relationships within the code's static structure.

3.2.2 CFG from Source Code

As outlined in process ② in Fig. 1, the CFG plays a crucial role in evaluating the operational behavior of smart contracts. The CFG represents the control flow within a contract, enabling the detection of potential vulnerabilities by mapping all possible execution paths. The graph extraction process begins by preprocessing the smart contract source code into bytecode, which is then parsed into opcode. During bytecode parsing, each pair of characters in the bytecode corresponds to a single opcode. Since the opcodes form a sequence of program commands, this sequence is treated as a "sentence" for graph node embedding. To embed these opcode sequences, Sent2Vec [39] is employed, converting them into fixed-length vector representations. The key advantage of using Sent2Vec is that it captures the semantic meaning of the opcode sequences, allowing the model to learn not just the syntax but the operational significance of each basic block in the smart contract. Th extracted information is represented as a graph G(V, E), where V denotes the set of vertices (nodes), corresponding to the basic blocks of the contract, and E represents the edges that define the relationships between vertices. In this paper, we measure the size of the input in terms of the number for vertices |V| and the number of edges |E| of the graph. Each vertex, composed of an opcode sequence, is embedded into a vector using Sent2Vec, capturing the semantic meaning of the command sequence. The graph G(V, E) undergoes further analysis through the graph structure extraction module, which computes centrality measures to evaluate the importance of each node. Centrality is a key indicator used to determine the significance of individual nodes within the graph. This study considers three centrality measures: Degree Centrality, Katz Centrality, and Closeness Centrality. These measures enhance the learning process of the vulnerability detection model, improving its ability to identify code vulnerabilities [40,41]:

- Degree Centrality measures the number of connections a node has to other nodes, providing insight into the node's importance within the graph. The degree centrality value is normalized by dividing the degree of a node by the maximum possible number of connections in the graph, $N-1$, where N is the total number of nodes. The degree centrality x_i of node i is calculated as:

$$x_i = \frac{\deg(i)}{N-1} \tag{3}$$

  where $\deg(i)$ represents the degree of node i, and the degree centrality indicates the fraction of nodes to which i is connected, normalized relative to the graph size.
- Katz centrality measures the importance of a node by considering the importance of its neighbors. The Katz centrality x_i of node i is calculated as a weighted sum of the Katz centralities of its neighbors j:

$$x_i = \alpha \sum_j A_{ij} x_j + \beta \tag{4}$$

  where A_{ij} represents the adjacency matrix of the graph, indicating whether an edge exists between nodes i and j, α is a weight controlling the influence of neighboring nodes, β is a constant representing the initial importance of a node. To ensure convergence, α must satisfy the condition:

$$\alpha < \frac{1}{\lambda_{max}} \tag{5}$$

  where λ_{max} denotes the maximum eigenvalue of the adjacency matrix.
- Closeness centrality quantifies how close a node is to all other nodes in the network. It is calculated as the reciprocal of the average shortest path length from node i to all other nodes. The closeness centrality x_i of node i is given by:

$$x_i = \frac{N-1}{\sum_{i \neq j} d(i,j)} \tag{6}$$

where $d(i,j)$ denotes the shortest path distance between nodes i and j, and N is the total number of nodes in the graph. Nodes with smaller average shortest path distances have higher closeness centrality, reflecting their accessibility within the network.

Centrality analysis provides an interpretable framework for assessing each node's contribution to the overall execution and identifying potential vulnerabilities. To better capture nuanced relationships, techniques such as dynamic analysis, deeper feature extraction, or enhanced context-aware modeling can be incorporated. Applying softmax regularization to the three centrality measures normalizes their values, allowing the model to appropriately weigh node significance during learning. This integrated approach improves the effectiveness of smart contract vulnerability detection.

- Degree centrality is computationally efficient, making it suitable for large-scale graphs where quick evaluations of connectivity are needed. It highlights the immediate influence of a node within its local neighborhood, emphasizing critical program commands or frequently accessed variables, which are often pivotal for detecting vulnerabilities. However, while Degree Centrality captures direct relationships, it may not account for more complex interdependencies that involve indirect paths. The space complexity of Degree Centrality is O (|V|), as it requires storing the neighbor list for each node. This means that the memory usage is proportional to the number of nodes, making it relatively space efficient compared to other centrality measures. The time complexity of calculating Degree Centrality is O (|V| + |E|). This is because the algorithm needs to traverse all the edges to count the degree for each node.
- Katz centrality extends the analysis by incorporating both immediate neighbors and the broader network influence, weighted by a decay factor (α). This measure is particularly effective for detecting vulnerabilities that propagate through multiple execution layers, such as reentrancy attacks. For example, Katz centrality captures the cumulative influence of functions indirectly connected through the call hierarchy. This centrality measure provides a more comprehensive view of node importance, but it may still miss subtle interdependencies if they are highly context-dependent or non-linear in nature. The space complexity for Katz Centrality is O ($|V|^2$). This is due to the need to store the adjacency matrix A of the graph, and matrix multiplication is required, which additionally demands space for storing intermediate results. This method involves high-dimensional matrix operations, making it relatively space-intensive. However, Katz centrality can provide valuable insights in more complex contract structures. The time complexity of calculating Katz Centrality is O ($|V|^3$) in the worst case due to the matrix multiplication required to compute the centralities for all nodes. If an adjacency matrix A of size |V| × |V| is used, matrix multiplication takes O ($|V|^3$) time, making this method relatively expensive for large graphs.
- Closeness centrality focuses on the proximity of a node to all other nodes in the graph, offering insights into the accessibility of critical paths, such as conditional branches or loops. These elements often act as bottlenecks or decision points in the code execution flow [42]. This measure is especially useful for identifying vulnerabilities that can impact the entire program's behavior due to their strategic location within the graph. Closeness centrality is especially useful for identifying vulnerabilities that can impact the entire program's behavior due to their strategic location within the graph. However, it may not fully capture vulnerabilities that depend on specific, non-obvious paths or complex interactions between nodes. The space complexity of Closeness Centrality is O ($|V|^2$). This is because it requires storing a distance matrix for calculating the shortest paths between all pairs of nodes. For instance, using the Floyd-Warshall algorithm, the matrix would need O ($|V|^2$) space. The time complexity of Closeness

Centrality is O ($|V|^2$) in the general case, especially when using algorithms like the Floyd-Warshall algorithm to compute the shortest paths between all pairs of nodes in the graph.

CFG and AST are two classical representations of source code. CFG primarily captures the flow changes between code blocks, often neglecting the syntactic information within each block. In contrast, AST emphasizes the syntax of the source code.

3.3 Vulnerability Detection Module

This section discusses the vulnerability detection module, which combines features obtained from the CFG and AST representations. After normalizing the features using softmax, they are concatenated to create prompts for LLM.

1. Feature dimension unification: When concatenating different features, at least one dimension must match. The two modules produce fixed-size outputs to mitigate the risk of overfitting while preserving critical information.
2. Feature fusion: After unifying the dimensions, features are fused by concatenating tensors along a specific dimension. The concatenated dimensions must be of the same size to ensure compatibility. The combined feature vector for analysis is represented as:

$$V_{concat} = Concat\left(V_{codeBERT}, V_{CFG}\right) \tag{7}$$

This fusion integrates both the textual context (from CodeBERT) and the structural information (from CFG embeddings) into a unified representation. By combining both textual and structural information, this approach creates a more holistic model that captures both the high-level semantic understanding and low-level operational behavior of the contract. Textual information, such as function names or comments, helps identify developer intentions, while structural analysis uncovers how different components interact and behave within the contract. For example, the integration of both textual and structural information enhances the detection of vulnerabilities like reentrancy attacks and timestamp dependence. Textual features can identify suspicious external calls, while structural analysis can detect recursive function calls that could lead to reentrancy vulnerabilities. Similarly, textual information highlights functions that depend on block timestamps, while structural analysis tracks the execution flow, uncovering dependencies that may lead to vulnerabilities. By combining these two domains, the model provides richer feature representations, enabling the detection of a wider range of vulnerabilities while ensuring a more comprehensive understanding of the code.

A set of customized prompts is designed for effective vulnerability detection. These prompts standardize outputs when LLMs are used as detectors. A basic prompt includes the following elements:

- Task: Determine whether the provided Solidity code contains vulnerabilities.
- Contract Code
- Answer: Vulnerable/Non-Vulnerable or specify the vulnerability type or error function.

The complete algorithm for vulnerability detection is illustrated in Algorithm 1.

Algorithm 1: Overall steps of the algorithm for calculating the vulnerability detection. Vulnerability detection with centrality and LLM

Input: Solidity code snippet
Output: Vulnerability status (Vulnerable or Non-Vulnerable)
1: // Step 1: Load and preprocess Solidity code

(Continued)

Algorithm 1 (continued)

2: Parse the Solidity code to generate the Abstract Syntax Tree (AST)
3: Construct the Control Flow Graph (CFG)
4: // Step 2: Generate feature vectors
5: Generate CodeBERT feature vector:
6:codebert_features <- generate_codebert_features(code)
7: Generate AST feature vector:
8: ast_features <- generate_ast_features(ast)
9: Compute CFG centrality features:
10: degree_centrality <-compute_degree_centrality(cfg)
11: katz_centrality <-compute_degree_centrality(cfg)
12: closeness_centrality <-compute_degree_centrality(cfg)
13: Combine CFG centrality features:
14: cfg_centrality_features <- concatenate(degree, katz, closeness)
15: // Step 3: Combine all features
16: combined_features <- concatenate(degree, katz, closeness)
17: // Step 4: Generate prompt for LLM
18: prompt <- "Task: output Vulnerable or Non-Vulnerable."
19: Append Solidity code and combined features to the prompt
20: // Step 5: Analyze vulnerability using LLM
21: llm_output <- analyze_with_llm(prompt)
22: if llm_output contains "Vulnerable" then
23: Output: Vulnerable
24: Identify vulnerable function and error type
25: else
26: Output: Non-Vulnerable
27: end if

4 Experiments

This section presents the empirical evaluation of the proposed method using publicly available datasets. The performance of the approach is assessed based on the following research questions:

RQ1: How does graph structural information contribute to the performance of the proposed approach?

RQ2: How effective is the proposed approach in detecting six well-known vulnerability types in smart contracts?

4.1 Data and Experimental Configurations

To evaluate the model's performance, a dataset of Ethereum smart contracts with verified source code was curated from Etherscan.io [43,44]. The dataset includes open-source smart contracts that were compiled and deployed on the Ethereum network. For the experiments, 300 code samples comprising approximately 30,000 contract codes were collected. To address the issue of code duplication, the bytecode and related metadata (such as address and compiler version) of the source were processed to avoid redundancies. In practice, many smart contracts are duplicated, reused, and distributed multiple times. This duplication can cause models to learn specific vulnerabilities linked to repeated code patterns. While this may enhance detection rates for known vulnerabilities, it risks overfitting, causing the model to memorize patterns instead

of generalizing, thereby reducing its ability to identify novel vulnerabilities. The dataset was randomly divided into three subsets: 70% for training, 20% for validation, and 10% for testing. The training set was used to teach the neural network to identify various types of vulnerabilities in smart contracts. The validation set was employed during training to fine-tune the model's parameters and prevent overfitting. The test set served to evaluate the model's generalizability in detecting vulnerabilities. Fig. 2 illustrates an example of a smart contract code containing a reentrancy vulnerability, one of the most critical and common issues in Ethereum smart contracts. This example provides a representative sample from the dataset, demonstrating the structure of the smart contract and highlighting the vulnerability. Such examples are essential for understanding the model's input and its analysis process.

```
function withdrawBalance() public {
        uint amount = balances[msg.sender];
        require(msg.sender.call.value(amount)());
        balances[msg.sender] = 0;
}
```

Figure 2: Example of smart contract code with a reentrancy vulnerability

The experimental environment consisted of an Ubuntu 20.04 operating system running on an Intel Xeon Gold 6230 CPU (2.1 GHz) with 256 GB RAM, a 1 TB SSD, and a GeForce RTX 3080 graphics card with video memory. The CUDA 11.6 computing library was utilized to accelerate model learning. The experimental code was implemented in Python 3.9. Additionally, GPT-3.5-Turbo was used as the API (Application Programming Interface) prompt generator in this study.

4.2 Metrics and Evaluation

The performance of four methods namely, the proposed method, CodeBERT [24], GPT, and CodeBERT + AST was compared. All models were trained and evaluated on the same dataset, using a contract sequence length of 512 and an embedding dimension of 768. The evaluation metrics included accuracy, precision, recall, and F1-score.

As shown in Table 1, the experiment assessed the impact of incorporating graph structural information into vulnerability detection. Additional vulnerabilities that were potentially overlooked by existing methods were identified using CFG and centrality analysis. The GPT model was excluded from this experiment as the primary objective was to evaluate the combined effect of graph structure information and CodeBERT. The results in Table 1 confirm that centrality analysis effectively identifies major flows in the code and areas prone to high vulnerability. Moreover, the complementary effect of integrating textual information with graph structural information was evident. These findings demonstrate that the inclusion of graph structure information enhances the model's ability to detect vulnerabilities, outperforming methods that rely solely on text-based analysis. A higher F1-score indicates that our model is more reliable in identifying vulnerabilities, boosting trust and adoption of blockchain technology. This improvement can help prioritize high-risk contracts, enabling more effective vulnerability mitigation. Furthermore, better precision and recall can automate auditing, reducing the workload for human auditors and improving the efficiency of security assessments. This scalability is crucial as the number of smart contracts grows. Ultimately, achieving a higher F1-score advances smart contract security by providing more robust, scalable, and reliable tools for vulnerability detection and risk management.

Table 1: Comparison of graph structure information

Method	Accuracy (%)	Recall (%)	Precision (%)	F1-score (%)
CodeBERT	81.26	78.16	81.60	79.84
CodeBERT + AST	83.48	82.35	77.13	79.65
Our approach	**86.70**	**84.87**	**85.24**	**84.46**

Note: Each metric's best result is highlighted in bold.

Table 2 presents the evaluation results for accuracy, precision, recall, and F1-score. When graph structure information was not utilized, these metrics were observed to be lower. Specifically, when GPT was used independently, it failed to accurately identify vulnerabilities and often produced inconsistent responses. Fig. 3 shows the results of the prompt execution, with one response from GPT 3.5-Turbo and another from our approach. From this figure, when executing with GPT, only 4 vulnerabilities were detected in the source code, which contains a total of 8 vulnerabilities. In contrast, when executing with our approach, all 8 vulnerabilities were identified. Additionally, our approach was able to provide a structural explanation of which functions were responsible for the vulnerabilities.

Table 2: Performance comparison between experimental results from six well-known vulnerability types

Vulnerability type	Method	Accuracy (%)	Recall (%)	Precision (%)	F1-score (%)
Integer overflow	GPT	72.72	–	–	–
	CodeBERT	84.21	82.99	85.71	84.32
	CodeBERT + AST	88.43	87.27	88.89	88.07
	Our approach	**90.68**	**89.26**	**91.05**	**90.14**
Integer underflow	GPT	50.33	–	–	–
	CodeBERT	66.31	62.57	73.78	67.71
	CodeBERT + AST	75.27	74.00	72.12	73.05
	Our approach	**83.03**	**83.09**	**86.43**	**84.73**
Reentrancy	GPT	66.00	–	–	–
	CodeBERT	74.82	74.36	83.12	78.49
	CodeBERT + AST	74.55	77.39	66.48	71.52
	Our approach	**84.30**	**84.22**	**85.36**	**84.78**
Timestamp	GPT	60.30	–	–	–
	CodeBERT	81.78	76.32	78.81	77.54
	CodeBERT + AST	78.54	77.90	79.46	78.67
	Our approach	**84.72**	**80.74**	**82.60**	**81.65**
TOD	GPT	74.66	–	–	–
	CodeBERT	81.18	81.56	78.38	79.94
	CodeBERT + AST	81.16	**85.49**	76.19	80.57
	Our approach	**82.96**	82.64	**80.41**	**81.51**
CallDepth	GPT	61.11	–	–	–
	CodeBERT	**82.80**	**73.96**	**85.12**	**79.14**
	CodeBERT + AST	73.23	61.24	72.05	66.20
	Our approach	77.46	70.02	78.79	74.14

Note: Each metric's best result is highlighted in bold.

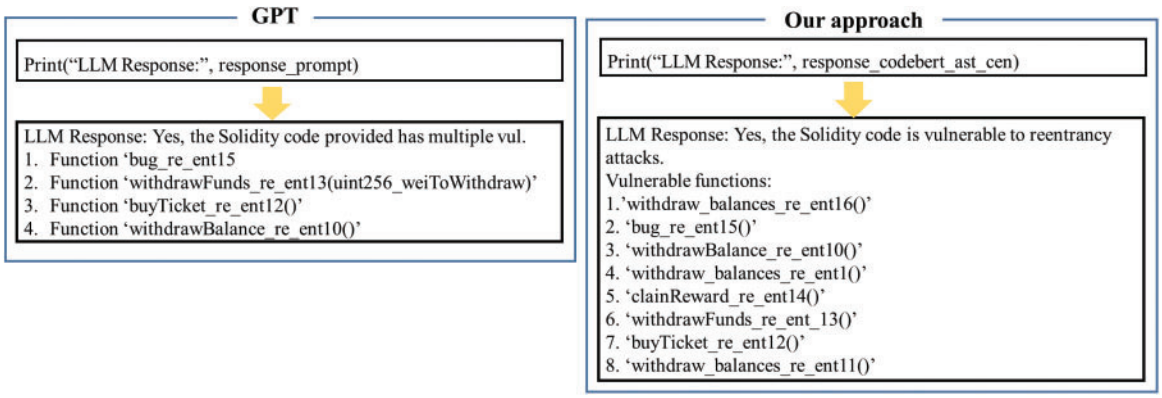

Figure 3: GPT & our approach prompt answers

Consequently, while GPT may serve as a supportive tool for performing graph structural analysis, it cannot reliably function as a standalone solution. The proposed model consistently outperformed the comparison methods across all evaluation indices. Notably, its performance was highest when the centrality structure was utilized, highlighting the importance of leveraging graph structure information in smart contract vulnerability detection. The improvement in performance can be attributed to the model's ability to emphasize critical paths and nodes in the code, as identified through graph centrality analysis, and to consider their influence on overall code execution. However, the performance of the graph structure for the CallDepth metric was slightly lower compared to other models. This is likely because CallDepth relies on the logical characteristics associated with the depth of specific function calls, which may not be effectively captured by graph centrality analysis or abstract syntax tree (AST) representations. This indicates that further feature extraction or improved learning strategies may be needed to better address depth-related information and call layer characteristics.

Table 3 presents a performance comparison between the proposed model and baseline models (CodeBERT, CodeBERT + AST, and our approach). From this table, it can be observed that the CodeBERT model required significantly more computation time. In contrast, our approach model demonstrated the shortest computation time, indicating higher efficiency in terms of both accuracy and resources for vulnerability detection tasks.

Table 3: Performance comparison

Method	Time (s)
CodeBERT	230
CodeBERT + AST	140
Our approach	**110**

Note: Each metric's best result is highlighted in bold.

5 Conclusions

With the advancement of blockchain technology, the use of smart contracts across various blockchain platforms has grown significantly. However, vulnerabilities in smart contracts have been exploited in attacks that not only harm users but also undermine the reliability of blockchain platforms. Consequently, detecting vulnerabilities prior to deploying smart contracts is crucial to mitigate potential risks.

This paper proposes a method for detecting vulnerabilities in smart contracts by combining code tokenization with graph-based structural analysis. The approach generates a CFG that captures the call relationships between opcode sequences in the source code, extracts control flow path characteristics, and employs graph centrality analysis to effectively identify the primary flows within the graph. Experimental results demonstrate that incorporating structural information enhances model performance and improves the identification of vulnerabilities.

The proposed model is specifically designed and trained for Solidity-based smart contracts, as Solidity is the predominant language in the Ethereum ecosystem. However, it is important to acknowledge that different smart contract languages vary in syntax, semantics, and bytecode compilation processes. While this approach is tailored to Solidity, it has the potential to be extended to other smart contract languages. By adapting graph-based representations, such as CFG or AST structures, for other languages, the underlying learning mechanisms of the proposed model could remain applicable.

Future research will aim to expand the scope of the method by exploring additional vulnerability types and enhancing performance through the integration of weighted elements in the graph combination process. Furthermore, the size and diversity of the dataset play a pivotal role in determining the model's generalizability. A larger, more diverse dataset would enable the model to learn from a broader range of examples, improving its ability to recognize complex patterns and reducing the risk of overfitting. Consistent graph structures, such as CFGs or ASTs, are critical to maintaining the model's robustness, even when dataset size or diversity changes. Additionally, advanced graph-based techniques could be leveraged to improve vulnerability detection further. For instance, spectral analysis may help identify potential vulnerabilities in interdependent contract functions due to high connectivity, such as those prone to reentrancy. Similarly, community detection algorithms could reveal isolated vulnerabilities by identifying graph substructures corresponding to functions or contracts with limited external connections. Furthermore, techniques like shortest path algorithms (e.g., Dijkstra or Bellman-Ford) could be employed to detect vulnerable entry points by analyzing the shortest paths between exposed functions and vulnerable contract states. Incorporating these methodologies could significantly enhance the identification and analysis of complex vulnerabilities. Future work will evaluate the impact of dataset design and structural representation on improving vulnerability detection performance. Additionally, the feasibility of adapting the model to other blockchain programming languages will be explored, along with an assessment of how variations in language design may influence the effectiveness of vulnerability detection.

Acknowledgement: Not applicable.

Funding Statement: This research was supported by the Seoul Business Agency (SBA), funded by the Seoul Metropolitan Government, through the Seoul R&BD Program (FB240022); and by the Korea Institute for Advancement of Technology (KIAT), funded by the Korea Government (MOTIE) (RS-2024-00406796), through the HRD Program for Industrial Innovation; and by the Excellent Researcher Support Project of Kwangwoon University in 2024.

Author Contributions: The authors confirm their contributions to the paper as follows: study conception and design: Ra-Yeon Choi, Dong-Hyuk Im; data collection: Ra-Yeon Choi, Minsoo Jang; analysis and interpretation of results: Ra-Yeon Choi, Dong-Hyuk Im; writing—original draft preparation: Ra-Yeon Choi, Yeji Song; writing—review and editing: Taekyung Kim, Jinhyun Ahn, Dong-Hyuk Im. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data underlying the results presented in this paper may be obtained from the authors upon reasonable request.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest regarding the present study.

References

1. Kalra S, Goel S, Dhawan M, Sharma S. ZEUS: analyzing safety of smart contracts. In: Network and Distributed Systems Security (NDSS) Symposium; 2018 Feb 18–21; San Diego, CA, USA. p. 1–12.
2. Yaga D, Mell P, Roby N, Scarfone K. Blockchain technology overview. arXiv:1906.11078. 2019.
3. Dinh TTA, Liu R, Zhang M, Chen G, Ooi BC, Wang J. Untangling blockchain: a data processing view of blockchain systems. IEEE Trans Knowl Data Eng. 2018;30(7):1366–85. doi:10.1109/TKDE.2017.2781227.
4. Wang JT, Hou HY, Chandra IJ, Chen J. Patient-aware information-hiding mechanism for a blockchain-based electronic health record system. Humcentric Comput Inf Sci. 2024;14(58):41–59. doi:10.22967/HCIS.2024.14.058.
5. Ragab M, Bahaddad AA, Hamed D, Alkhayyat A, Mansour RF, Gupta D. Blockchain-driven privacy preserving electronic health records analysis using sine cosine algorithm with deep learning model. Humcentric Comput Inf Sci. 2024;14(9):37–54. doi:10.22967/HCIS.2024.14.009.
6. Paganini P. Uniswap and Lendf.me hacked, attacker stole $25 million worth of cryptocurrency. [cited 2020 Apr 20]. Available from: https://securityaffairs.com/101895/cyber-crime/uniswap-lendf-me-hacked.html.
7. Hong H, Sun Z. A secure peer to peer multiparty transaction scheme based on blockchain. Peer Peer Netw Appl. 2021;14(3):1106–17. doi:10.1007/s12083-021-01088-4.
8. Hong H, Hu B, Sun Z. Toward secure and accountable data transmission in Narrow Band Internet of Things based on blockchain. Int J Distrib Sens Netw. 2019;15(4):155014771984272. doi:10.1177/1550147719842725.
9. Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, et al. VulDeePecker: a deep learning-based system for vulnerability detection. In: 2018 Network and Distributed System Security Symposium (NDSS'18); 2018 Feb 18–21; San Diego, CA, USA. p. 1–15.
10. Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, et al. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA); 2018 Dec 17–20; Orlando, FL, USA. p. 757–62.
11. Kim S, Choi J, Ahmed ME, Nepal S, Kim H. VulDeBERT: a vulnerability detection system using BERT. In: 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW); 2022 Oct 31–Nov 3; Charlotte, NC, USA. p. 69–74. doi:10.1109/ISSREW55968.2022.00042.
12. Contro F, Crosara M, Ceccato M, Dalla Preda M. EtherSolve: computing an accurate control-flow graph from ethereum bytecode. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC); 2021 May 20–21; Madrid, Spain. p. 127–37.
13. Albert E, Gordillo P, Livshits B, Rubio A, Sergey I. EthIR: A framework for high-level analysis of ethereum bytecode. In: International Symposium on Automated Technology for Verification and Analysis; 2018 Oct 7–12; Los Angeles, CA, USA. Berlin/Heidelberg, Germany: Springer; 2018, p. 513–20.
14. Liu Z, Qian P, Wang X, Zhuang Y, Qiu L, Wang X. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Trans Knowl Data Eng. 2023;35(2):1296–310.
15. Zhou Y, Liu S, Siow J, Du X, Liu Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Advances in Neural Information Processing Systems (NIPS); 2019 Dec 8–14; Vancouver, BC, Canada. p. 10197–207.
16. Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. In: International Conference on Learning Representations (ICLR); 2018 Apr 30–May 3; Vancouver, BC, Canada. p. 1–17.
17. Shen S, Shinde S, Ramesh S, Roychoudhury A, Saxena P. Neuro-symbolic execution: augmenting symbolic execution with neural constraints. In: 2019 Network and Distributed System Security Symposium (NDSS'19); 2019 Feb 24–27; San Diego, CA, USA. p. 1–15.
18. Cheong YY, Choi LY, Shin J, Kim T, Ahn J, Im DH. GNN-based ethereum smart contract multi-label vulnerability detection. In: IEEE International Conference on Information Networking (ICOIN); 2024 Jan 17–19; Ho Chi Minh City, Vietnam. p. 57–61.

19. Salim MM, Deng X, Park JH. A privacy-preserving local differential privacy-based federated learning model to secure LLM from adversarial attacks. Humcentric Comput Inf Sci. 2024;14(57):15–40. doi:10.22967/HCIS.2024.14.057.
20. Brown T, Mann B, Ryder N, Subbiah M, Dhariwal P, Amodei D. Language models are few-shot learners. Adv Neural Inf Process Syst. 2020;33:1877–901.
21. Hu S, Huang T, İlhan F, Tekin SF, Liu L. Large language model-powered smart contract vulnerability detection: new perspectives. In: 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA); 2023 Nov 1–4; Atlanta, GA, USA. p. 297–306.
22. Yu L, Chen S, Yuan H, Wang P, Huang Z, Zhang J, et al. Smart-LLaMA: two-stage post-training of large language models for smart contract vulnerability detection and explanation. arXiv:2411.06221. 2024.
23. He Z, Li Z, Yang S, Qiao A, Zhang X, Luo X, et al. Large language models for blockchain security: a systematic literature review. arXiv:2403.14280. 2024.
24. Jin H, Li Q. Fine-tuning pre-trained CodeBERT for code search in smart contract. Wuhan Univ J Nat Sci. 2023;28(3):237–45. doi:10.1051/wujns/2023283237.
25. Golosova J, Romanovs A. The advantages and disadvantages of the blockchain technology. In: 2018 IEEE 6th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE); 2018 Nov 8–10; Vilnius, Lithuania. p. 1–6.
26. Zheng Z, Xie S, Dai HN, Chen X, Wang H. Blockchain challenges and opportunities: a survey. Int J Web Grid Serv. 2018;14(4):352. doi:10.1504/IJWGS.2018.095647.
27. Zheng Z, Xie S, Dai HN, Chen W, Chen X, Weng J, et al. An overview on smart contracts: challenges, advances and platforms. Future Gener Comput Syst. 2020;105(5):475–91. doi:10.1016/j.future.2019.12.019.
28. Szabo N. Formalizing and securing relationships on public networks. First Monday. 1997;2(9):1–25. doi:10.5210/fm.v2i9.548.
29. Sayeed S, Marco-Gisbert H, Caira T. Smart contract: attacks and protections. IEEE Access. 2020;8:24416–27. doi:10.1109/ACCESS.2020.2970495.
30. Ethereum. Solidity documentation. [cited 2020 Jul 2]. Available from: https://solidity.readthedocs.io/.
31. Wood G. A secure decentralised generalised transaction ledger. Ethereum project yellow paper. [cited 2025 Jan 1]. Available from: https://cryptodeep.ru/doc/paper.pdf.
32. Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS); 2016 Oct 24–28; Vienna, Austria. p. 254–69.
33. Pro S. Smart contract security issues: What are smart contract vulnerabilities how to protect. [cited 2019 Apr 21]. Available from: https://smartym.pro/blog/smart-contract-security-issues-smart-contract-vulnerabilities-and-how-to-protect/.
34. Koschke R, Falke R, Frenzel P. Clone detection using abstract syntax suffix trees. In: 2006 13th Working Conference on Reverse Engineering (WCRE); 2006 Oct 23–27; Benevento, Italy; 2006. p. 253–62.
35. Phan AV, Le Nguyen M, Thu Bui L. Convolutional neural networks over control flow graphs for software defect prediction. In: IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI); 2017 Nov 6–8; Boston, MA, USA: IEEE. p. 45–52.
36. Brent L, Jurisevic A, Kong M, Liu E, Gauthier F, Gramoli V, et al. Vandal: a scalable security analysis framework for smart contracts. arXiv:1809.03981. 2018.
37. Gunawan D, Sembiring CA, Budiman MA. The implementation of cosine similarity to calculate text relevance between two documents. J Phys: Conf Ser. 2018;978:012120. doi:10.1088/1742-6596/978/1/012120.
38. Wu H, Zhang Z, Wang S, Lei Y, Lin B, Qin Y, et al. Peculiar: smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE); 2021 Oct 25–28; Wuhan, China. p. 378–89. doi:10.1109/ISSRE52982.2021.00047.
39. Pagliardini M, Gupta P, Jaggi M. Unsupervised learning of sentence embeddings using compositionaln-gram features. In: Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies; 2018 Jun 1–6; New Orleans, LA, USA. p. 528–40.

40. Wu Y, Zou D, Dou S, Yang W, Xu D, Jin H. VulCNN: an image-inspired scalable vulnerability detection system. In: 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE); 2022 May 25–27; Pittsburgh, PA, USA. p. 2365–76.
41. Choi RY, Cheong YY, Im DH. Image based smart contract flow graphs for vulnerability detection. Seoul, Republic of Korea: Korean Institute of Information Scientists and Engineers; 2024. p. 852–4.
42. Newman MEJ. The structure and function of complex networks. SIAM Rev. 2003;45(2):167–256. doi:10.1137/S003614450342480.
43. Etherscan. [cited 2025 Jan 1]. Available from: https://etherscan.io.
44. Hu T, Li B, Pan Z, Qian C. Detect defects of solidity smart contract based on the knowledge graph. IEEE Trans Reliab. 2024;73(1):186–202. doi:10.1109/TR.2023.3233999.