Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.070930

ARTICLE

Smart Contract Vulnerability Detection Based on Symbolic Execution and Graph Neural Networks

Haoxin Sun[1], Xiao Yu[1,*], Jiale Li[1], Yitong Xu[1], Jie Yu[1], Huanhuan Li[1], Yuanzhang Li[2] and Yu-An Tan[2]

[1]School of Computer Science and Technology, Shandong University of Technology, Zibo, 250000, China
[2]School of Computer Science, Beijing Institute of Technology, Beijing, 100081, China

*Corresponding Author: Xiao Yu. Email: yuxiao8907118@163.com

Received: 28 July 2025; Accepted: 28 September 2025; Published: 09 December 2025

ABSTRACT: Since the advent of smart contracts, security vulnerabilities have remained a persistent challenge, compromsing both the reliability of contract execution and the overall stability of the virtual currency market. Consequently, the academic community has devoted increasing attention to these security risks. However, conventional approaches to vulnerability detection frequently exhibit limited accuracy. To address this limitation, the present study introduces a novel vulnerability detection framework called GNNSE that integrates symbolic execution with graph neural networks (GNNs). The proposed method first constructs semantic graphs to comprehensively capture the control flow and data flow dependencies within smart contracts. These graphs are subsequently processed using GNNs to efficiently identify contracts with a high likelihood of vulnerabilities. For these high-risk contracts, symbolic execution is employed to perform fine-grained, path-level analysis, thereby improving overall detection precision. Experimental results on a dataset comprising 10,079 contracts demonstrate that the proposed method achieves detection precisions of 93.58% for reentrancy vulnerabilities and 92.73% for timestamp-dependent vulnerabilities.

KEYWORDS: Smart contracts; graph neural networks; symbolic execution; vulnerability detection

1 Introduction

Blockchain technology has experienced significant advancement, particularly in the development and deployment of smart contracts [1]. However, the widespread adoption of smart contracts has also introduced substantial cybersecurity risks. A prominent example is the 2016 DAO incident, in which malicious actors exploited reentrancy vulnerability in contracts to a considerable amount of Ethereum funds [2]. Consequently, the detection of security vulnerabilities in smart contracts has become as a critical area of focus in blockchain security research.

Recent advances in deep neural network techniques have greatly facilitated their application to the detection of vulnerabilities within smart contracts. Existing detection approaches can generally be categorized into two types. The first type is sequence-based models [3], which represent contract code as a one-dimensional sequence and employ recurrent neural networks (RNNs) for vulnerability identification [4]. While these approaches effectively capture local syntactic features, they fail to account for the hierarchical syntactic hierarchy, control flow, and data dependency present in contract code. Consequently, they struggle to model global semantic information, leading to lower detection accuracy. The second type consists of graph-based models [5], which transform contract code into graph structure and apply graph convolutional networks (GCNs) or similar techniques to perform vulnerability detection [6]. These methods

are advantageous in that they preserve the structural information of the code and can effectively capture global semantic relationships. However, graph-based models also have limitations. First, they are inadequate to identify complex logical vulnerabilities that are triggered only under specific inputs or path conditions, which can result in missed detection. Second, they often produce high falseopositive rates when analyzing structurally similar contracts with significant semantic divergency. Therefore, although graph-based models outperform sequence-based models in capturing hierarchical structure and global semantic features, their precision remains insufficient when used in isolation.

To address these challenges and improve the precision of vulnerability identification, this paper introduces GNNSE, a novel hybrid framework that combines GNNs with symbolic execution. The proposed method leverages GNNs to model both structural and global semantic features of smart contracts, thereby overcoming the feature representation limitations of sequence models. Symbolic execution, paths and constraints can be explored in greater depth, compensating for the inability of graph-based models in complex vulnerability detection. Specifically, the approach involves extracting function information, critical variables, and data flow from contract source code to build semantic graphs. It then applies GNNs to analyze node-edge relationships, identify vulnerabilities, and filter high-risk contracts. Finally, precise path analysis of these high-risk contracts is performed via symbolic execution to achieve accurate vulnerability detection. Experimental results demonstrate that GNNSE effectively integrates the hierarchical and semantics modeling strengths of GNNs with the path exploration and constraint solving capabilities of symbolic execution. This integration significantly enhances the detection of complex logic vulnerabilities, reduces false-positive rates, and improves the overall reliability of smart contract vulnerability detection.

The contributions of this paper are summarized as follows:

(1) A vulnerability detection framework is proposed that integrates GNNs with symbolic execution. GNNs capture semantic graph dependencies and learn vulnerability patterns from large-scale datasets, enabling efficient identification of high-risk contracts.
(2) Path-sensitive vulnerability analysis is incorporated through symbolic execution, which simulates contract execution paths, generates path constraints, and performs in-depth logical analysis to accurately detect potential vulnerabilities in high-risk contracts, thereby improving detection precision.

2 Related Work

Techniques for smart contract security analysis techniques are generally categorized into two primary methodologies: traditional static analysis and deep learning-based approaches. Among static analysis techniques, symbolic execution plays a pivotal role. By treating variables as symbolic values and tracking their propagation, symbolic execution enables comprehensive exploration of execution path and facilitates the identification of potential vulnerabilities. Representative tools include Oyente [7], which performs dynamic exploration of contract execution paths while employing constraint-solving techniques and auxiliary methods for analysis and verification, and Mythril [8], which combines semantic abstraction, data taint inspection, and execution path evaluation to uncover security flaws within smart contracts.

In parallel, the rapid development of deep learning approaches has created new opportunities for vulnerability detection. Tann et al. [9] pioneered the use of an LSTM-based deep learning framework to detect security flaws in contracts. Huang [10] converted contract bytecode into RGB-formatted images to facilitate feature extraction and vulnerability identification via CNNs. In recent years, progress has been made across several dimensions of research. In symbolic execution hybrid methods, He et al. [11] proposed a “parallel and simplified symbol execution” (ParSE) mechanism, which reduces the solving load by leveraging multi-core parallelization, simplifying symbol states and constraints, and integrating with Oyente and Mythril as plugins, significantly enhancing detection throughput and path coverage. In

the domain of multimodal and multi-view learning, TMF-Net [12] achieves cross-modal alignment by integrating multimodal features from source code, bytecode, and graph structures. This method captures local syntax details while strengthening global semantics and dependency modeling, outperforming single-modal methods in vulnerability detection. For edge AI-based auditing, EVuLLM [13] adopts lightweight parameter efficient fine-tuning and quantization techniques such as QLoRA, enabling large models to operately efficiently in resource-constrained edge environments and improving the practicality of auditing. CSCO [14] mines contextual semantic features of contracts from opcode sequences to characterize execution logic and behavior patterns, combining expert knowledge to extract security-oriented pattern features for improved detection. Furthermore, approaches based on Large Language Models (LLMs) have recently emerged. SmartGuard [15] proposes an LLM-enhanced vulnerability detection framework, while LLM4Fuzz [16] and PromFuzz [17] adopt LLMs to guide fuzz testing, identify critical paths, and locate functional vulnerabilities. These advancements indicate a shift from single-model deep learning methods toward multimodal fusion and more intelligent analysis techniques.

Despite these advancements, challenges remain. Deep learning methods, beginning with LSTM and CNN frameworks, have improved vulnerability detection accuracy but still struggle with modeling complex semantic relationships. While symbolic execution improvement approaches have increased efficiency, their ability to capture complex logical scenarios remains limited. Similarly, multimodal and edge AI methods enhance semantics and improve practicality but remains insufficient when analyzing deep logical dependencies and complex vulnerability patterns. Large Language Models show promise in enabling intelligent exploration but suffer from high computational costs and limited controllability. Consequently, accurately detecting vulnerability in complex logical scenarios continues to be a major research challenge, as false positives or false negatives [18,19] may occur.

To address these limitations, this study introduces GNNSE, a hybrid approach that combines graph neural networks (GNNs) with symbolic execution. GNNSE inherits the strengths of symbolic execution in precise path exploration and constraint solving while leveraging GNN's ability to learn graph structural and semantic representation of smart contracts. By modeling contract execution logic and dependency relationships more comprehensively, GNNSE provides a more robust and accurate solution for smart contract vulnerability detection.

3 System Design

Compared with traditional static analysis methods, GNNs are capable of modeling the hierarchical structure and semantic relationships within smart contracts. By leveraging large-scale training, GNNs can learn deep semantic patterns of vulnerabilities, thereby significantly improving the detection of structural and logical vulnerabilities. However, GNNs also inherent limitations: they lack accurate simulation of execution paths, which hinders their ability to accurately identify vulnerabilities that are triggered only under specific input conditions. This limitation also makes them prone to misjudging vulnerabilities involving complex logical independency. In contrast, symbolic execution offers precise vulnerability verification through constraint solving but suffers from scalability challenges, such as low execution efficiency, making it impractical for large-scale contract analysis. Although prior studies have attempted to integrate multiple detection techniques, these efforts often adopt heterogeneous strategies, leading to inconsistencies that undermine both detection accuracy and efficiency. To address these challenges, this study proposes a smart contract vulnerability detection method called GNNSE that combines GNNs and symbolic execution to enhance detection accuracy while maintaining efficiency.

GNNSE consists of three core modules:

(1) Semantic Graph Construction: Parses smart contract source code to extract key functions and logical structures, subsequently constructing a semantic graph representation.
(2) GNNs Vulnerability Screening: Processes the semantic graph through a GNN model to identify potential vulnerability patterns and efficiently filter out high-risk contracts.
(3) Symbolic Execution Testing: Symbolizes input parameters, performs path exploration, and utilizes constraint solving to verify potential vulnerabilities with high precision while reducing false positive rates. The structural layout of GNNSE vulnerability analysis framework is depicted in Fig. 1.

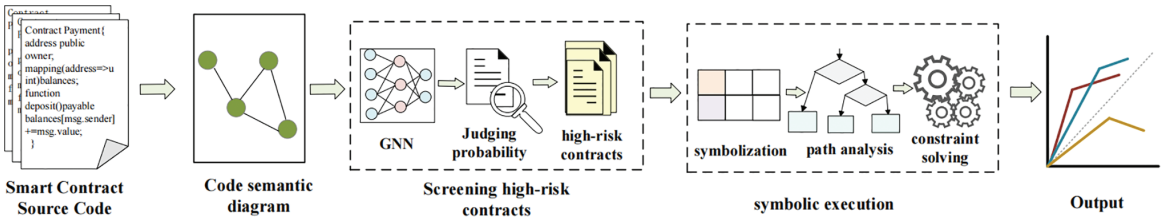

Figure 1: Vulnerability detection framework diagram of GNNSE

3.1 Semantic Graph Construction

Smart contract vulnerabilities are often closely related to specific function calls and variables. For instance, functions such as call.value and block.timestamp play a pivotal role in determining the presence of vulnerabilities. However, traditional methods for vulnerability detection, including text sequences or control flow graphs, are often inadequate in capturing the full syntax and semantics of smart contracts. Building upon the work of Liu et al. [20], this paper proposes the adoption of semantic graphs to effectively model both semantic information and control dependencies. By explicitly modeling key function calls as primary nodes and aggregating important variables into these nodes through feature transfer and aggregation mechanisms during normalization, the proposed method emphasizes the core semantics relevant to vulnerability detection while preserving contextual information and reducing graph complexity. This approach not only preserves contextual information and captures the complex logic of the contract, but also reduces graph complexity, thereby facilitating a more efficient representation of contract's logic. Furthermore, the normalization process ensures the retention of dependencies between variables and critical function calls, while simplifying the overall graph structure, mitigating training challenges arising from excessive density, and ultimately improving the accuracy of vulnerability detection.

3.1.1 Construction of Nodes and Edges

In vulnerability detection, various components of a function exert differing degrees of impact on vulnerability detection. To achieve more refined modeling of vulnerability patterns, this paper divides functions into two types of nodes: (1) Main nodes: These nodes represent function calls that are critically associated with vulnerability detection, typically encompassing both built-in functions and custom functions that could potentially trigger vulnerabilities. These nodes are direct sources of potential vulnerability triggers. For instance, in the case of reentrancy vulnerabilities, function calls involving call.value should be monitored, wheras for timestamp vulnerabilities, the usage of block.timestamp may serve as the root cause of the vulnerability. (2) Secondary nodes: These nodes are variables that indirectly influence vulnerability detection. They represent state information or critical variables within the contract, such as balance or bonusFlag. While these variables do not directly activate vulnerabilities, they provide essential contextual information that aids in the explanation and identification of vulnerability behavior.

Based on this categorization of intra-function nodes and drawing inspiration from the concepts of Control Flow Graphs (CFGs) and Data Flow Graphs (DFGs), we propose the construction of three types of edges—control flow, data flow, and backflow—to model the relationships and dependencies between nodes. These edges collectively represent potential execution paths within a smart contract function, with each edge annotated by a timestamp to denote its relative execution order. Specifically, control flow edges represent transitions between basic blocks within control structures such as conditionals, branches, and loops. Data flow edges capture dependencies between variable definitions and their subsequent usages, where an edge is established from node A to node B if A defines a variable that B later utilizes. Backflow edges, introduced to capture feedback or cyclic dependencies, connect successor nodes back to their dominators or loop entry points, thereby highlighting control loops and strongly connected structures. By formalizing the semantics of these edge types, we achieve a more precise representation of the execution and dependency semantics within contract functions.

3.1.2 Normalising the Semantic Graph

The initial semantic graph comprises a significant number of both main nodes and secondary nodes. Most GNNs have typically flat information propagation, which may lead to the underemphasis of certain nodes that are more critical to vulnerability detection. In other words, the importance of different nodes in vulnerability detection process can vary substantially. If all nodes are treated equally during the training of GNNs, the model may fail to sufficiently prioritize the most influential nodes, potentially undermining detection accuracy. Therefore, to enhance the focus of feature representation and the effectiveness of the graph structure, this paper introduces a semantic graph normalisation strategy.

Specifically, only the main nodes, which carry significant semantic weight, are retained, while the semantic information from the secondary nodes is transmitted to their corresponding main nodes through feature aggregation before the secondary nodes are removed. After normalization, each main node retains its intrinsic features while simultaneously incorporating information from neighboring secondary nodes. As illustrated in Fig. 2, during the graph normalization process: the state of N1 directly influences the transfer logic of K2, thereby propagating its features to K2; the value of N2 value is derived from the logic of K1, requiring feature transmission to K1; and the close coupling of N3 with the transfer operation of K3 necessitates feature passing to K3.

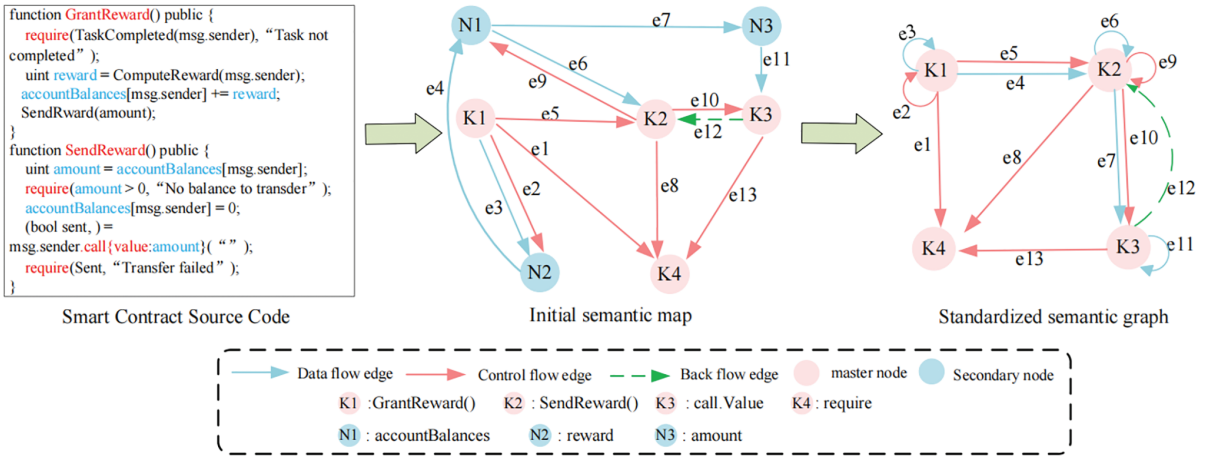

Figure 2: Source code data construct semantic graph data structure

3.2 GNNs Vulnerability Screening

To address the limitations of sequence models in capturing structured and global semantic information, this paper employs GNNs with more flexible architectures to analyze semantic graphs and identify high-risk contracts with potential vulnerabilities. GNNs leverage the structural information of contracts, such as function calls, variable dependencies, and control flow, and are capable of capturing semantic associations between distant nodes through learnable aggregation strategies. In contrast to GCNs, which rely on fixed-neighbourhood aggregation strategies, GNNs offer enhanced expressive power and feature modelling capabilities. This enables them to automatically learn potential vulnerability patterns from normalized semantic graphs, thereby enhancing vulnerability detection performance. By feeding the constructed semantic graph into GNNs, the model can efficiently and accurately screen out high-risk contracts with potential vulnerabilities, thus providing a refined target set for the subsequent symbolic execution.

At the core of GNNs is the graph convolution operation, which weights the feature vectors of each node based on the information from its neighboring nodes. The information is then passed to the next layer, where the node features are updated gradually, progressively enhancing the model's understanding of graph structure. For each node v in the graph, the update process of its k-th layer feature representation is shown in Formula (1):

$$h_v^{(k)} = \sigma\left(w^{(k)} \cdot \left(h_v^{(k-1)} + \sum_{v \in N(v)} h_v^{(k-1)}\right)\right) \tag{1}$$

where $h_v^{(k)}$ denotes the representation of the characteristic of node v in the k-th, while $h_v^{(k-1)}$ is the representation of the characteristic of node v in the previous layer. $N(v)$ represents the set of neighbors of v, while $h_v^{(k-1)}$ is the feature representation of a neighboring node v in the previous layer. $w^{(k)}$ is the matrix of learning weights for the k-th, and σ denotes the activation function.

After multiple graph convolution operations, the features of nodes are progressively updated, allowing the model to gain a more accurate representation of the contract. This process improves the model's ability to identify potential vulnerabilities within smart contract. At the final stage, the updated features of each node are processed by a fully connected layer, which maps the features to vulnerability risk values. This allows the model to effectively access the risk associated with potential vulnerabilities in the contract. Specifically, for each node v, its output feature vector $\widehat{h_v}$ is computed via Eq. (2):

$$\widehat{h_v} = FC(h_v^{(l)}) \tag{2}$$

where $h_v^{(l)}$ is the convolutional representation of the features of the last graph of a node v in the last convolutional graph, and FC is a fully connected surface operation to map the features of a node to a high-dimensional space. The output feature vector $\widehat{h_v^{(l)}}$ is used to estimate the vulnerability probability using Eq. (3):

$$P(v) = \sigma\left(\widehat{h_v} W_{out} + b_{out}\right) \tag{3}$$

where $P(v)$ is the probability value of vulnerability corresponding to node v. W_{out} is the weight matrix of the output layer. b_{out} is the discriminant of the output layer, and σ is a sigmoid activation function, which maps the output to the range $[0, 1]$, providing the probability of vulnerability.

Based on the output probability value of GNNs $P(v)$, this model generates an overall vulnerability probability for each smart contract. The results are stored in an array format, which includes the contract name and corresponding probability. During the preprocessing stage, the system first reads the source code

of all contract and stores them in a dictionary, where the key is the file name and the value is the source code. The system then traverses the results output by GNNs, matching the contract names with the file names in the dictionary. Contracts with a vulnerability probability exceeding a specified threshold are identified and marked as high-risk contracts. In this study, the default threshold is set to 0.7, which effectively filters out potential high-risk contracts. This threshold can be modified based on specific requirements. For contracts identified as high-risk, the system extracts their source code from the dictionary and stores it separately for further analysis by the symbolic execution module. Through the risk contract screening of GNNs, the system effectively filters high-risk contracts in large-scale smart contract datasets, allowing symbolic execution resources to be focused on those contracts most likely to contain vulnerabilities.

3.3 Symbolic Execution Testing

This module provides in-depth analysis of high-risk contracts through three stages: symbolization, path analysis, and constraint solving. By thoroughly examining the execution paths of contracts, it uncovers vulnerabilities that may be embedded or hidden in complex paths, reducing false positives and enhancing the precision of security flaws detection.

3.3.1 Symbolization

Initially, the source code of each identified high-risk contract is parsed by the compiler and converted into bytecode. This process begins with syntax analysis to construct an abstract syntax tree (AST), where components such as function definitions, variable declarations, and control flow structures are represented as AST nodes. Based on the node type and the contextual semantics of the AST, each statement is mapped to its corresponding opcode (e.g., a function call node may be mapped to CALL upcode), resulting in bytecode that aligns with the logic of the source code. Next, the bytecode undergoes instruction-level parsing to identify the contract's opcodes and their corresponding operational logic. This step also involves extracting the contract's input parameters (e.g., account addresses, fund amounts, transaction amounts, etc.) and execution flow information. Input parameters are then replaced with symbolic variables in place of actual numerical inputs. This symbolic representation decouples the contract's execution from specific input data, enabling an abstract analysis of contract's behavior and an accurate simulation of its runtime results under varying input conditions.

As illustrated in Fig. 3, for a deposit and withdrawal contract, the four variables msg.sender, msg.value, balances [msg.sender], and amount need to be symbolized. These symbolized variables will be utilized during the path exploration and constraint solving stages to construct path conditions, assisting in the identification of potential vulnerabilities that may arise in complex input dependencies.

3.3.2 Path Analysis

The path analysis phase involves an in-depth analysis of the execution flow extracted from the contract, aiming to generate all possible execution paths. Each path represents an execution sequence within the contract, from start to finish, encompassing all conditional branches, loops, function calls, and other relevant constructs. The key to path analysis is the generation of path constraints. Each path traverses several decision points, such as if, for, and while statements. At these decision points, symbolic execution creates symbolic constraints corresponding to the conditional expressions, ensuring that the path's execution adhere to the logical constraints. This approach guarantees that the entire path analysis is both accurate and complete.

A path constraint $PC(pi)$ is a collection of symbolic constraints that describes all the control flow conditions that must be satisfied during the execution of a path pi. Each path pi consists of multiple

decision points, which determine the execution branches of the program. These branches are determined not by fixed values variables. To capture all potential execution flows along the path, symbolic execution generates the corresponding symbolic constraints for each decision point. Specifically, let the input to the contract be represented by the set of symbolic variables $X = \{x_1, x_2, \ldots, x_n\}$, and let each control condition $C_j(x_1, x_2, \ldots, x_n)$ on path pi describe the constraints that the symbolic input must satisfy under that condition. By evaluating all control flow decisions along the path, the path constraint $PC(pi)$ connects these conditions with logical AND to form a complete constraint expression for the execution of the path, as shown in Eq. (4):

$$PC(pi) = \Lambda_{j=1}^{m} C_j(x_1, x_2, \ldots, x_n) \tag{4}$$

where m is the number of control conditions involved on the path pi, and each $C_j(x_1, x_2, \ldots, x_n)$ is a symbolic constraint corresponding to the j-th condition on the path. The result of the path analysis of the contract depicted in Fig. 3 is shown in Fig. 4. In the case of the deposit function, there are no decision points, meaning that the path constraint can be reduced to a single unconstrained execution path. The withdraw function, however, contains a 'require' decision point, resulting in the generation of two execution paths.

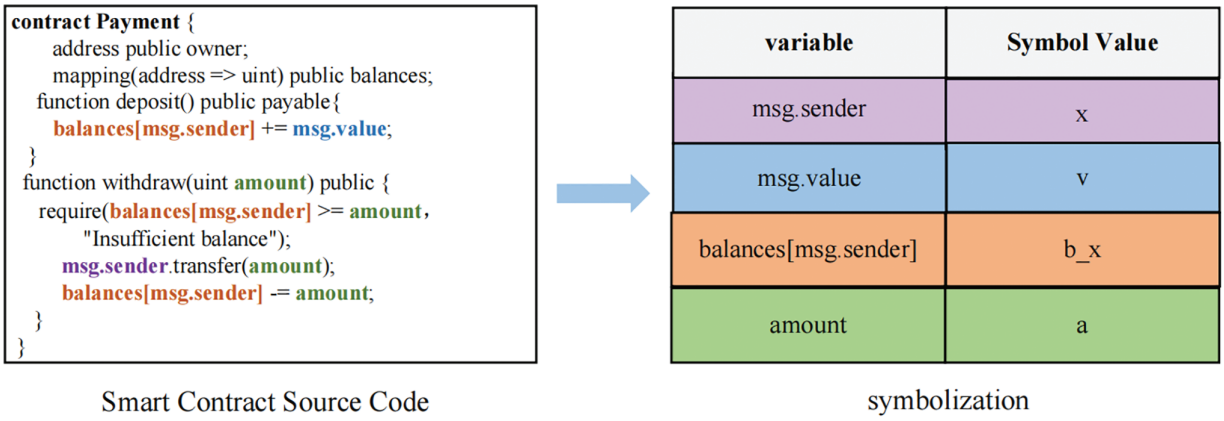

Figure 3: Symbolic presentation of variables

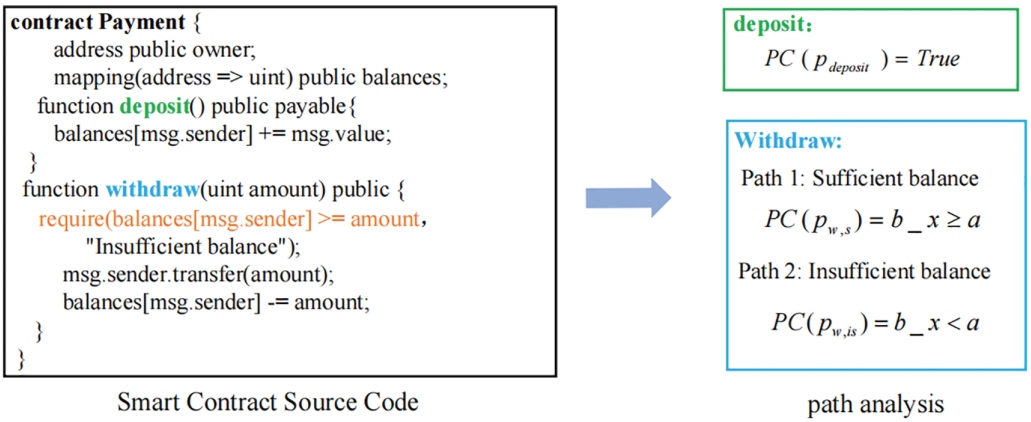

Figure 4: Path analysis result chart of the contract

3.3.3 Constraint Solving

The constraint solving stage, based on path constraints, is crucial for verifying the feasibility of each execution path and identifying whether there exist inputs values that satisfy the constraint conditions. The key to the constraint solving stage is to translate the path conditions into formal logical expressions and submit them to an underlying Satisfiability Modulo Theories (SMT) solver for processing. The solver will attempt to find a set of input values that satisfy all path constraints simultaneously. Upon finding a solution, the resulting model provides a set of specific input values that direct the program to execute along a specific path, thereby verifying its reachability. If the path is reachable, potential vulnerabilities along the path may be activated. In practice, this process involves obtaining the path constraint formulas through symbolization and path analysis, which are then converted into SMT format, such as the expression supported by Z3. These formulas are then passed to the solver. Once the solver returns a model, it can be mapped to actual transaction inputs to verify whether potential vulnerabilities exist along the path. By solving these constraints, the approach not only confirms the existence of vulnerabilities but also identifies security risks in complex execution paths, thereby improving the accuracy and reliability of vulnerability detection.

Taking the contract in Fig. 4 as an example. In the case of the deposit() function, there is only one unconstrained path, meaning that no constraint solving is necessary. However, withdraw() function contains two execution paths. In path 1, they system checks whether there exist an input that satisfies $b\_x \geq a$. If the constraint is satisfied, the contract continues executing. In contrast, in path 2, if there is an input that satisfies $b\_x < a$, the require statement fails, and the contract rolls back the transaction without transferring funds or updating the balance. Constraint solving reveals a reentrancy vulnerability in the withdraw() function. In path 1, transfer() is executed before the balance is updated. Since transfer() triggers the receiver's callback function (e.g., fallback() or receive()), an attacker can exploit this callback to recall the withdraw() function via a malicious contract during this process. This allows the attacker to bypass the balance check, enabling repeated withdrawals and potentially draining the contract's funds.

To address the issue of path explosion, this paper introduces path pruning strategies in the constraint solving stage. Specifically, during the symbol execution process, the system dynamically assesses the satisfiability of the current path conditions, using the solver to verify constraints in real-time. When it is determined that the constraint set for a particular path has no solution, that is, the path is unreachable under any input, the subsequent exploration of the path is immediately terminated and discarded. This strategy effectively conserves solving resources by avoiding invalid paths, thereby focusing the solver's efforts on real and feasible paths. While maintaining the path coverage ability, it improves the efficiency of analysis and reduces the performance pressure related to path explosion.

It is important to note, however, that while symbolic execution offers high accuracy in vulnerability detection, its computational cost can be significant, especially when analyzing large-scale smart contracts or contracts with complex control flow structures. To mitigate this issue, our approach incorporates GNNs-based risk contract screening mechanism. This strategy ensures that symbolic execution is only applied to contracts identified as high-risk, significantly reducing the computational burden of symbolic execution and concentrating limited computational resources on contracts that are most likely to contain vulnerabilities.

4 Experimentation

Reentrant vulnerabilities and timestamp dependency vulnerabilities are among the most prevalent and critical security issues in smart contracts. These two types of vulnerabilities have not only triggered numerous typical attack events in history, causing serious impacts on the blockchain ecosystem, but they also remain among the most common vulnerabilities in current smart contracts [21]. Therefore, they are of considerable importance both in academic research and practical applications. While various detection

methods have been proposed in recent years to identify these vulnerabilities, existing approaches still face challenges such as high false positive rates and insufficient detection accuracy [22,23]. This underscores that detecting these vulnerabilities remains a significant challenge and a focal point in the field of smart contract security. Therefore, this paper focuses on reentrant vulnerabilities and timestamp dependency vulnerabilities as the primary research subjects. This choice is motivated not only by their high-risk nature and widespread relevance in practical applications, but also by their potential to effectively validate the performance of the proposed methods.

It is important to point out that the experimental data presented in this study specifically addresses the two aforementioned types of vulnerabilities. This focus ensures that the advantages of the model are clearly demonstrated in high-risk vulnerability scenarios. In future work, the scope of experiments will be expanded to include other common vulnerabilities, such as integer overflows and access control defects, to further evaluate the generality and applicability of the proposed GNNSE model across a broader spectrum of security concerns.

4.1 Experimental Setup

4.1.1 Datasets

To assess the performance of the proposed approach, this paper integrates the publicly available datasets from Liu et al. [24] and Qian et al. [25]. These datasets were initially compiled from real smart contracts collected by Liu and Qian from the Ethereum and VNT chain platforms. Due to the presence of partially duplicated contract data, this dataset underwent a deduplication process to ensure the validity of the experimental results. The final experimental dataset consists of 10,079 deduplicated smart contract files. Among these data, the reentrant vulnerability dataset includes 5189 contracts, of which 2016 are identified as vulnerable, while the timestamp dependency vulnerability dataset comprises 4890 smart contracts, of which 2242 being vulnerable.

4.1.2 Evaluation Metrics

The experiment employs four key performance metrics to evaluate detection performance: accuracy, recall, precision, and F1-score. Accuracy (ACC) serves as the primary indicator for the model's effectiveness. Recall measures the proportion of actual positive instances that the model correctly identifies. Precision refers to the proportion of true positive predictions among all positive predictions made by the model. The F1-score provides a harmonic average of precision and recall, offering a unified measure of the model's classification performance [26,27].

4.1.3 Environment and Parameter

Experiments were conducted on a system equipped with a 2.60 GHz Intel Core i7-9750H CPU and 16 GB RAM. The optimization process employed the Adam algorithm in conjunction with the cross-entropy cost function for model training. The training parameters included a learning rate of 0.002, a packet loss rate of 0.2, a batch size of 64, and 100 training epochs. The dataset was split randomly, with 80% as the training set and 20% as the testing set. Each model was evaluated across five independent experiments, and the average value was used as the model's performance indicator for each vulnerability type.

4.2 Experimental Results

In this section, a comparative assessment is conducted between the proposed GNNSE and the mainstream contract vulnerability detection techniques. The results of this comparison are presented in Table 1.

The methods include three sequence models (RNN, LSTM [28], and GRU [29]), two graph-based models (GCNs and TMP), and a newly proposed attention-based wide and deep neural network AWDNN [30]. Additionally, ablation experiments were performed to access the effect of the symbolic execution module on vulnerability detection. Specially, the GNN model in the table represents the performance of the GNNSE model without the symbolic execution module.

Table 1: Performance comparison of reentrant and timestamp vulnerabilities

Models	Reentrant vulnerability				Timestamp dependency vulnerability			
	ACC (%)	Recall (%)	Precision (%)	F1 (%)	ACC (%)	Recall (%)	Precision (%)	F1 (%)
RNN	69.23	71.79	68.85	70.92	65.81	71.31	62.58	66.67
LSTM	70.51	68.37	71.42	69.86	73.50	75.14	72.54	73.82
GRU	82.47	83.76	81.66	82.70	84.61	83.75	82.35	83.05
GCNs	85.89	80.32	83.05	81.66	86.75	85.47	87.71	86.58
TMP	88.03	86.06	85.36	85.01	89.21	84.42	88.02	86.19
GNN	88.89	82.78	87.82	85.23	89.74	91.80	84.21	87.84
AWDNN	91.80	90.59	86.22	88.51	90.59	93.16	88.61	90.83
GNNSE	93.58	89.34	92.73	90.83	92.73	95.72	90.32	92.94

The data presented in Table 1 indicates that GNNSE outperforms the three sequence models in the vulnerability detection task. Specifically, in the detection of reentrant vulnerability and timestamp vulnerability, the average accuracy of the sequence models is only 74.07% and 74.64%, respectively. When compared to the best-performing sequence model, GRU, GNNSE demonstrates substantial improvements in accuracy–rising from 82.47% to 93.58% for reentrant vulnerabilities, and from 84.61% to 92.73% for timestamp vulnerabilities. In addition to accuracy, GNNSE also outperforms the sequence models in other metrics. For example, in reentrant vulnerability detection task, GNNSE achieves a F1-score of 90.83%, which is 8.13% higher than GRU's F1-score of 82.70%. Additionally, GNNSE's recall rate increases from 83.76% in GRU to 89.34%. These improvements primarily stem from the differences in the models' abilities to capture structural and semantic information inherent in the contracts. Sequence models treat code as a one-dimensional sequence, which limits the ability to capture key structural and semantic information, such as function calls, control flow, data dependencies, resulting in restricted detection capabilities. In contrast, GNNSE, adopting a graph-based model, offers clear advantages in capturing structural information and global semantic information of the contract, enabling more effective vulnerability detection.

Moreover, GNNSE also outperforms both GCNs and TMP in detecting reentrant and timestamp vulnerability. Compared to the optimal TMP model, GNNSE achieves an increase in accuracy of 5.55%, from 88.03% to 93.58%, for reentrant vulnerability detection, and 3.52%, from 89.21% to 92.73%, for timestamp vulnerability detection. The F1-score improvements are similarly significant, with GNNSE increasing the F1-score by 5.82% and 6.75%, respectively. This performance improvement can be attributed to the limitations of using a single graph model, which makes it difficult to detect complex logic vulnerabilities triggered by specific inputs or path conditions. While TMP introduces temporal mechanisms to capture contextual information, it still relies on static graph models, which lack path sensitivity to path reachability during program execution. In contrast, GNNSE integrates a symbolic execution mechanism within its graph neural network, which systematically explores all execution paths of the program, constructing precise path

constraints. This allows GNNSE to compensate for the limitations of single graph-based models, enabling the identification of complex logic vulnerabilities and reducing both false positives and false negatives.

Furthermore, when compared with the newly proposed AWDNN method, GNNSE shows improvement in the accuracy of detecting reentrant vulnerability and timestamp vulnerability, increasing from 91.80% to 93.58% for reentrant vulnerabilities, and from 90.59% to 92.73% for timestamp vulnerabilities. The primary reason for this difference lies in the approach used by AWDNN, which combines wide and deep neural networks with custom attention mechanisms. While this method captures both shallow rules and deeper semantics and emphasizes key segments via attention, it remains primarily focused on learning complex logical structural relationships indirectly through deep perception and attention. As a result, it lacks sensitivity to the structural and path-based aspects of the contract, limiting its ability to detect complex logic vulnerabilities. GNNSE, however, integrates both graph neural networks and symbol execution, making it sensitive to both structure and path, thus improving its ability to identify and mitigate false positives and false negatives, resulting in superior performance in vulnerability detection tasks.

We further access the effectiveness of each model in smart contract vulnerability detection by using the ROC curve and the AUC value. The ROC curve, which ranges from 0 to 1, measures classification performance, with higher values indicating greater discriminative power and better performance. As seen in the AUC values presented in Figs. 5 and 6, GNNSE outperforms all other models in both reentrant vulnerability and timestamp vulnerability detection tasks, achieving AUC values of 0.9168 and 0.9264. These results significantly surpass the AUC values of GCN, which are 0.8418 and 0.8675, respectively. This indicates that GNNSE, by leveraging a more flexible aggregation mechanism, dynamically adjusts the information transmission process based on node semantics, allowing for more effective capture of complex dependency relationships and global semantic information within contracts. This enhancement not only improves the accuracy of high-risk contracts screening but also provides more accurate target contracts for the subsequent symbol execution stages, thereby significantly enhancing the overall vulnerability detection capability.

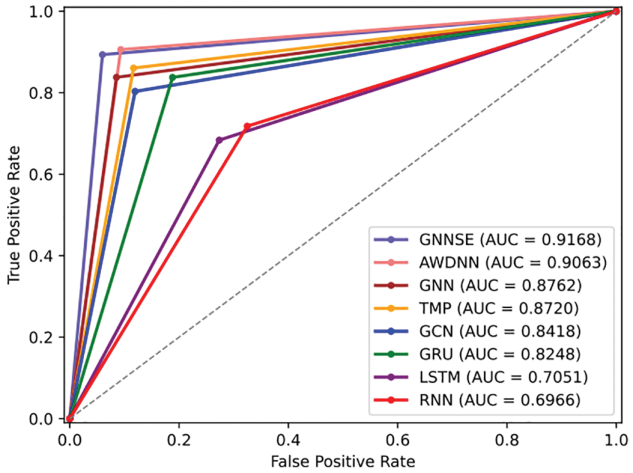

Figure 5: ROC curve and AUC value of reentrant vulnerability

In contrast, TMP exhibits AUC values of 0.8720 and 0.8738, which are superior to the sequence-based model but still fall short due to the model's limited ability to handle path sensitivities. The sequence-based models, on the other hand, struggle with vulnerability detection as they fail to account for the structural aspects of the code. Notably, RNN performs particularly poorly, with an AUC value of only 0.6429 in timestamp vulnerability detection task, substantially lower than that of GNNSE. Furthermore, GNNSE also

surpasses the newly proposed AWDNN method, improving the AUC for both vulnerability detection by 0.0105 and 0.0156, respectively. Experimental findings indicate that GNNSE exhibits a superior detection performance, primarily due to its advanced capability to capture semantic information, and through symbolic execution, perform in-depth path analysis to comprehensively explore potential vulnerabilities in smart contracts.

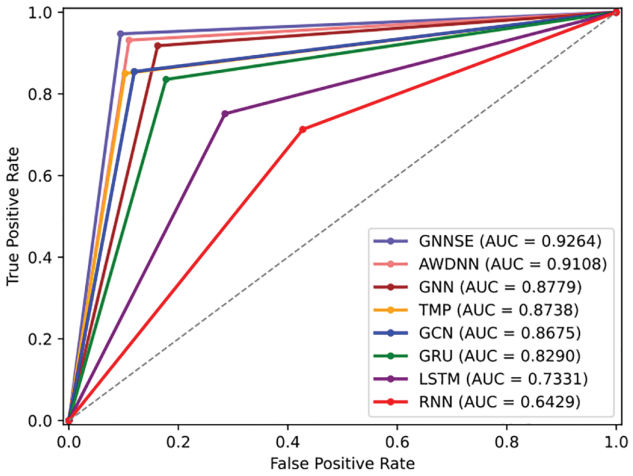

Figure 6: ROC curve and AUC value of timestamp vulnerability

The comparison between GNNSE and GNN (obtained by removing the symbolic execution module from GNNSE) further emphasizes the significant contribution of symbolic execution in enhancing vulnerability detection. As shown in Table 1, the introduction of symbolic execution module leads to a substantial performance boost. In the reentrant vulnerability detection task, GNNSE achieves an accuracy of 93.58%, surpassing GNN by 4.69%, and its recall rate increases from 82.78% to 89.34%, marking an increase of 6.56%. This indicates that symbolic execution systematically explores all the contract execution paths and constructs precise path constraints, which helps detect deeply embedded logical vulnerabilities that GNN alon may miss. Additionally, in terms of precision, GNNSE reaches 92.73%, which is 4.91% higher than GNN's precision of 87.82%, indicating that symbolic execution also contributing significantly to reducing false positives. For the timestamp vulnerability detection task, the overall capability of GNNSE is clearly superior to that of GNN, further confirming that symbolic execution module plays a crucial role in enhancing the accuracy of vulnerability detection by simulating all possible execution paths and uncovering hidden vulnerabilities.

5 Conclusion

This paper proposes a novel method for smart contract vulnerability detection that combines symbolic execution with GNNs. The proposed method first transforms the contract's source code into a semantic graph, extracting key structural information such as control flow and data flow. High-risk contracts are then screened through GNNs, followed by an in-depth analysis of the contract's execution path through symbolic execution, which facilitates precise identification of vulnerabilities. The experimental results demonstrate that this method performs exceptionally well in detecting reentrant vulnerabilities and timestamp vulnerabilities, outperforming traditional sequence models and pure graph models in terms of accuracy and recall. It is important to emphasize that the aim of this study is to improve the accuracy of vulnerability detection through the integration of GNNs and symbolic execution. Accordingly, our experiments have primarily focused on evaluating detection performance, with a more limited focus on runtime and memory overhead.

In future work, we plan to extend the proposed approach to cover a broader range of contract vulnerability detection tasks. We will also optimize the symbolic execution component, develop a comprehensive performance evaluation framework, and explore dynamic configuration mechanisms to further enhance the scalability and efficiency of the proposed method.

Acknowledgement: Not applicable.

Funding Statement: This study was supported by the National Key Research and Development Program of China (2020YFB1005704).

Author Contributions: The authors confirm their contribution to the paper as follows: Haoxin Sun: writing—initial draft, validation, software, resources, methodology, data organization, conceptualization. Xiao Yu, Jiale Li, Jie Yu: organizing background knowledge, reviewing articles. Huanhuan Li, Yitong Xu, Yuanzhang Li, Yu-An Tan: supervision. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The data that support the findings of this study are available from the corresponding author upon reasonable request.

Ethics Approval: Not applicable.

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

References

1. Wu G, Wang H, Lai X, Wang M, He D, Chan S. A comprehensive survey of smart contract security: state of the art and research directions. J Netw Comput Appl. 2024;226(2):103882. doi:10.1016/j.jnca.2024.103882.
2. Qian P, Liu Z, He Q, Huang B, Tian D, Wang X. Smart contract vulnerability detection technique: a survey. arXiv:2209.05872. 2022.
3. Tang X, Du Y, Lai A, Zhang Z, Shi L. Lightning cat: a deep learning-based solution for smart contracts vulnerability detection. Sci Rep. 2023;13:20106. doi:10.21203/rs.3.rs-3104649/v1.
4. Jain A, Zamir AR, Savarese S, Saxena A. Structural-RNN: deep learning on spatio-temporal graphs. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR); 2016 Jun 26–Jul 1; Las Vegas, NV, USA. p. 5308–17.
5. Xu J, Wang T, Lv M, Chen T, Zhu T, Ji B. MVD-HG: multigranularity smart contract vulnerability detection method based on heterogeneous graphs. Cybersecurity. 2024;7(1):55. doi:10.1186/s42400-024-00245-5.
6. Li Z, Li W, Li X, Zhang Y. Stateguard: detecting state derailment defects in decentralized exchange smart contract. In: Proceedings of the Companion Proceedings of the ACM Web Conference 2024; 2024 May 13–17; Singapore. p. 810–3.
7. 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); 2016 Oct 24–28; Vienna, Austria. p. 254–69.
8. Sharma N, Sharma S. A survey of Mythril, a smart contract security analysis tool for EVM bytecode. Indian J Nat Sci. 2022;13(75):51003–10.
9. Tann WJ, Han XJ, Gupta SS, Ong YS. Towards safer smart contracts: a sequence learning approach to detecting security threats. arXiv:1811.06632. 2018.
10. Huang TH. Hunting the Ethereum smart contract: color-inspired inspection of potential attacks. arXiv:1807.01868. 2018.
11. He L, Zhao X, Wang Y. Parse: efficient detection of smart contract vulnerabilities via parallel and simplified symbolic execution. In: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings; 2024 Apr 14–20; Lisbon, Portugal. p. 272–3.

12. Wang T, Zhao X, Zhang J. TMF-Net: multimodal smart contract vulnerability detection based on multiscale transformer fusion. Inf Fusion. 2025;122:103189. doi:10.1016/j.inffus.2025.103189.
13. Mandana E, Vlahavas G, Vakali A. EVuLLM: ethereum smart contract vulnerability detection using large language models. Electronics. 2025;14(16):3226. doi:10.3390/electronics14163226.
14. Huang H, Guo L, Zhao L, Wang H, Xu C, Jiang S. Effective combining source code and opcode for accurate vulnerability detection of smart contracts in edge AI systems. Appl Soft Comput. 2024;158:111556. doi:10.1016/j.asoc.2024.111556.
15. Ding H, Liu Y, Piao X, Song H, Ji Z. SmartGuard: an LLM-enhanced framework for smart contract vulnerability detection. Expert Syst Appl. 2025;269:126479. doi:10.1016/j.eswa.2025.126479.
16. Shou C, Liu J, Lu D, Sen K. LLM4Fuzz: guided fuzzing of smart contracts with large language models. arXiv:2401.11108. 2024.
17. Yu J, Shao Y, Miao H, Shi J. PROMPTFUZZ: harnessing fuzzing techniques for robust testing of prompt injection in LLMs. arXiv:2409.14729. 2024.
18. Nguyen HH, Nguyen NM, Xie C, Ahmadi Z, Kudendo D, Doan TN, et al. MANDO: multi-level heterogeneous graph embeddings for fine-grained detection of smart contract vulnerabilities. In: Proceedings of the 2022 IEEE 9th International Conference on Data Science and Advanced Analytics (DSAA); 2022 Oct 13–16; Shenzhen, China. p. 1–10.
19. Bresil M, Prasad P, Sayeed MS, Bukar UA. Deep learning-based vulnerability detection solutions in smart contracts: a comparative and meta-analysis of existing approaches. IEEE Access. 2025;13:28894–919. doi:10.1109/access.2025.3532326.
20. Liu Z, Qian P, Wang X, Zhu L, He Q, Ji S. Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. arXiv:2106.09282. 2021.
21. Iuliano G, Di Nucci D. Smart contract vulnerabilities, tools, and benchmarks: an updated systematic literature review. arXiv:2412.01719. 2024.
22. Oss T, Budde CE. Vulnerability anti-patterns in Solidity: increasing smart contracts security by reducing false alarms. arXiv:2410.17204. 2024.
23. Wang Z, Chen J, Zheng P, Zhang Y, Zhang W, Zheng Z. Unity is strength: enhancing precision in reentrancy vulnerability detection of smart contract analysis tools. IEEE Trans Softw Eng. 2024;51(1):1–13. doi:10.1109/tse.2024.3427321.
24. Liu Z, Qian P, Yang J, Liu L, Xu X, He Q, et al. Rethinking smart contract fuzzing: fuzzing with invocation ordering and important branch revisiting. IEEE Trans Inf Forensics Secur. 2023;18:1237–51. doi:10.1109/tifs.2023.3237370.
25. Qian P, Liu Z, Yin Y, He Q. Cross-modality mutual learning for enhancing smart contract vulnerability detection on bytecode. In: Proceedings of the ACM Web Conference 2023; 2023 Apr 30–May 4; Austin, TX, USA. p. 2220–9.
26. Opitz J. A closer look at classification evaluation metrics and a critical reflection of common evaluation practice. Trans Assoc Comput Linguist. 2024;12(1):820–36. doi:10.1162/tacl_a_00675.
27. Powers DM. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. arXiv:2010.16061. 2020.
28. Yu Y, Si X, Hu C, Zhang J. A review of recurrent neural networks: lSTM cells and network architectures. Neural Comput. 2019;31(7):1235–70. doi:10.1162/neco_a_01199.
29. Shiri FM, Perumal T, Mustapha N, Mohamed R. A comprehensive overview and comparative analysis on deep learning models: CNN, RNN, LSTM, GRU. arXiv:2305.17473. 2023.
30. Osei SB, Huang R, Ma Z. An attention-based wide and deep neural network for reentrancy vulnerability detection in smart contracts. J Syst Softw. 2025;223:112361. doi:10.1016/j.jss.2025.112361.