Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2024.050281

ARTICLE

Smart Contract Vulnerability Detection Method Based on Feature Graph and Multiple Attention Mechanisms

Zhenxiang He*, Zhenyu Zhao, Ke Chen and Yanlin Liu

School of Cyber Security, Gansu University of Political Science and Law, Lanzhou, 730070, China

*Corresponding Author: Zhenxiang He. Email: hzx6198@gsupl.edu.cn

Received: 01 February 2024 Accepted: 08 April 2024 Published: 15 May 2024

ABSTRACT

The fast-paced development of blockchain technology is evident. Yet, the security concerns of smart contracts represent a significant challenge to the stability and dependability of the entire blockchain ecosystem. Conventional smart contract vulnerability detection primarily relies on static analysis tools, which are less efficient and accurate. Although deep learning methods have improved detection efficiency, they are unable to fully utilize the static relationships within contracts. Therefore, we have adopted the advantages of the above two methods, combining feature extraction mode of tools with deep learning techniques. Firstly, we have constructed corresponding feature extraction mode for different vulnerabilities, which are used to extract feature graphs from the source code of smart contracts. Then, the node features in feature graphs are fed into a graph convolutional neural network for training, and the edge features are processed using a method that combines attention mechanism with gated units. Ultimately, the revised node features and edge features are concatenated through a multi-head attention mechanism. The result of the splicing is a global representation of the entire feature graph. Our method was tested on three types of data: Timestamp vulnerabilities, reentrancy vulnerabilities, and access control vulnerabilities, where the F1 score of our method reaches 84.63%, 92.55%, and 61.36%. The results indicate that our method surpasses most others in detecting smart contract vulnerabilities.

KEYWORDS

Blockchain; smart contracts; deep learning; graph neural networks

1 Introduction

Zuckerberg, the founder of Facebook, had forecasted that humanity would progress into a new era of Web 3.0, indicating blockchain technology's central position in the modern Internet field. Essentially, blockchain technology is a distributed ledger technology. Miners on the blockchain use computational power to verify and record transactions, which ensures data integrity and security. Furthermore, blockchain technology is widely used in digital transactions and cryptocurrency due to its decentralized and tamper-proof nature. As the value of cryptocurrencies and digital assets continues to rise, blockchain technology has become the cornerstone of a vast commercial bridge. According to statistics, the market value of cryptocurrencies is about $835 billion as of 2023 [1]. Thus, protecting the security of blockchain technology has become a focal point of attention for many experts and scholars. Exploiting security vulnerabilities in smart contracts is the main threat to blockchain technology. Smart contracts are segments of code that can execute automatically on the blockchain. Since multiple users jointly develop these codes, many contracts have potential security vulnerabilities during their composition. Additionally, due to the tamper-proof nature of blockchain, contracts that have already been deployed are challenging to modify. Moreover, smart contracts usually require interaction with external contracts; attackers commonly exploit it to attack smart contracts. Nowadays, many smart contracts are deployed on various blockchain platforms, such as virtual currencies, the Internet of Things, healthcare, logistics, etc. Nearly 100 million smart contracts have been successfully deployed on Ethereum, participating in over 2 billion transactions. However, frequent contract vulnerabilities have posed a massive challenge to the blockchain, and recurrent security incidents also reduce people's trust in Internet finance. The existing detection methods primarily focus on specific keywords in vulnerable contracts, which is insufficient to address the ever-changing tactics of attacks. The method proposed in this paper further tracks vulnerabilities based on control flow, considers tracking the "edges" in the feature graph, and employs graph neural networks to detect latent vulnerabilities within contracts effectively.

Conventional methods of detecting vulnerabilities in smart contracts significantly depend on experts devising feature-matching modes for static analysis tools, drawing from their previous experiences. Many representative smart contract analysis tools have emerged, which performed exceptionally well in the early stages of smart contracts. However, as the scale of blockchain continues to expand, attackers can skillfully exploit the vulnerabilities of these tools to commit crimes against smart contracts. Consequently, researchers have applied deep learning to smart contract vulnerability detection. Reference [2] converts the contract's bytecode into a grayscale image and then uses a convolutional neural network to extract features from these images. Yet, the convolution and pooling processes in convolutional neural networks can blur the inherent logical relationships in contracts, which will ignore crucial information. Aiming to concentrate on the internal logical relations within the code, researchers refer to the method in which the compiler processes the code into an Abstract Syntax Tree (AST); they have suggested abstracting the contract's source code into a non-Euclidean graph before processing [3]. On this foundation, Zhuang et al. [4] proposed a smart contract vulnerability detection method based on graph neural networks. This method employs Degree-free Graph Convolutional Neural Networks (DR-GCN) and Temporal Message Propagation Networks (TMP) to learn the features of the graph. Reference [5] proposed a fully automatic method for smart contract vulnerability detection at the function level, which focuses on functions and variables that directly contribute to the vulnerability of a contract. A limitation of these approaches is their excessive emphasis on the features of nodes within the graph while neglecting the significance of edge features. However, edge features often play a crucial role in expressing complex relationships and interaction behaviors; effectively aggregating edge features can reveal potential vulnerabilities and abnormal patterns in smart contracts.

To further respond to current attack methods. This paper improves the feature extraction mode proposed by Qian [5] and proposes a feature extraction mode specifically for access control vulnerabilities. To more effectively extract the information contained in the feature graph. This paper proposes the Multi-Head Node and Edge Combine Model (MH-NEC), which fully utilizes both node and edge features. It employs a degree-free graph neural network to extract node features and combines attention mechanisms with gated units for updating edge features. Attention mechanisms and gated units can finely regulate the flow of information, which ensures the effective utilization of edge information during the aggregation process. In addition, the multi-head attention layer of the MH-NEC model enhances the model's ability to capture interactions between nodes and edges. During this process, the attention weights produced by each head dictate the method of combining node features and edge features, which achieves a complete representation of the information within the graph. According to experiments on timestamp vulnerabilities, reentrancy vulnerabilities, and access control vulnerabilities, this method has good results. The main contributions of this paper are as follows:

1. Regarding timestamp vulnerabilities, based on the original research, we further focused on operations involving relational operators. Miners comparing time. stamp with unreasonable time values is a method of manipulating timestamps.

2. For reentrancy vulnerabilities, this paper further checks whether the contract contains *spender.call*. *spender.call* allows for calling any function of the target contract (as long as it matches its signature) and has no gas restrictions, which leads to reentrancy vulnerabilities in the contract.

3. This paper establishes a complete feature extraction mode for access control vulnerabilities. This method checks whether the contract contains *tx.origin* and permission change operations, and we examine whether essential functions have stringent access restrictions. The above methods compensate for the limited means of access control vulnerability detection.

4. This paper adopts a method combining attention mechanism and gate control unit in terms of processing edge features. This method precisely controls information flow and ensures the effective use of side information in the aggregation process.

5. This paper comprehensively integrates updated node and edge features through a multi-head attention mechanism, effectively enhancing the depth and breadth of feature processing.

In the paper's Section 2, we introduce the related work of smart contract vulnerability detection; Section 3 mentions some preparations required for the experiment; Section 4 introduces the complete experimental method; and Section 5 compares the experimental results. The results are analyzed in detail, and Section 6 concludes the paper.

2 Related Work

Effectively preventing smart contract vulnerabilities is an important part of ensuring the security of blockchain technology. This section introduces six representative static analysis tools for early smart contract vulnerability detection tasks. Next, we discussed three methods and their main advantages when applying deep learning to smart contract detection.

2.1 Static Analysis Tool

Static analysis tools are an important approach in the early stage of smart contract vulnerability detection. Experts develop these tools based on past contract detection experience. These tools detect contract vulnerabilities by analyzing the source code of contracts without executing the program. They mainly rely on understanding and insight into potential execution paths to identify known contract vulnerabilities. The most representative static analysis tools include Oyente [6], one of the early static analysis tools for smart contract vulnerabilities. It is based on the control flow graph of the contract and uses symbolic execution technology to detect potential problems in smart contracts. Oyente can identify multiple vulnerability types, such as reentrancy attacks, exception handling issues, and dependency flaws based on transaction order. Security [7] is a tool with scalability, fully automated operation, and high accuracy. Security examines a contract's dependency graph and extracts precise semantic data from the code, which enables an assessment of a contract's compliance and potential security vulnerabilities. Mythril [8] is a static analyzer for smart contracts. It combines conceptual analysis, taint tracking, and control flow verification to identify typical vulnerabilities in Ethereum smart contracts. Mythril can detect various problems, including reentrancy attacks, integer overflow problems, and exception management flaws. Slither [9] is a static analysis framework for Ethereum smart contracts. Slither simplifies the contract analysis process using a static single assignment form and a reduced instruction set while retaining the semantic information lost when Solidity source code is converted to Ethereum Virtual Machine (EVM) bytecode. Manticore [10] is a dynamic analysis tool dedicated to binary analysis, especially suitable for smart contracts and complex software systems. It offers high flexibility, automation capabilities, and in-depth analytical precision. SmartCheck [11] is a static analysis tool that deeply supports the Solidity language. It automatically identifies common programming errors and security vulnerabilities through in-depth scanning and analysis of smart contract codes. A primary problem of these conventional tools is the increased computational complexity as the execution paths deepen. Additionally, due to their heavy reliance on predefined rules, these tools have limited semantic analysis capabilities, which may lead to false positives and negatives.

2.2 The Methods of Deep Learning

As deep learning models evolve, researchers are applying these techniques to smart contract vulnerability detection [12–14]. Compared to traditional static analysis tools, deep learning approaches are not dependent on predefined detection rules. Instead, they learn the features of vulnerabilities through the training process. This approach demonstrates greater flexibility and adaptability when addressing new vulnerabilities. Smart contract vulnerability detection methods based on deep learning can be divided into three categories. The first type refers to the natural language processing method used to analyze the contract content. These approaches transform the vulnerability detection task into a multi-classification or multi-label problem. Specifically, this involves converting the source code or opcodes into AST [15] or Javascript Object Notation (JSON) files [16]. Subsequently, text processing methods such as N-gram [17] are used for feature extraction, followed by classification using models like Support Vector Machine (SVM) and Logistic Regression (LR). The second method focuses on the binary bytecode within contracts [18]. According to the RGB principle, this method transforms binary files into coded images of equal size. It uses a convolutional neural network to extract and classify the features of the image. This type of method solves the problem of poor scalability in traditional tools. The limitations of these two approaches stem from their inadequate consideration of the inherent logical relationships in the code. When analyzing contracts, it is important to deeply understand the program's internal logical relationships. In-depth dependency analysis can accurately assess the functionality of a contract and its potential risks. The work of Allamanis et al. has proven that graphics can be used to represent source code; this method can preserve the syntactic and semantic information of the code [19]. Based on this, researchers have proposed a third type of smart contract detection method, which abstracts the smart contract source code into a non-Euclidean graph, such as a Program Dependency Graph (PDG) [20] or Function Call Graph (FCG) [21]. Then, features are extracted using graph neural networks for classification. Researchers use DR-GCN or Bidirectional Graph Neural Network (BGNN) [22] for training.

2.3 Motivation

Methods of using graph neural networks to analyze contracts still have limitations. Firstly, these studies [20–22] mainly focus on the node features of contract graphs; they pay less attention to edge features. This suggests that the internal dependency relationships in the code still offer limited assistance in vulnerability detection tasks. Secondly, the original feature extraction mode can no longer deal with newly emerging contract vulnerabilities. To address these issues, we updated the existing feature extraction mode. More importantly, we innovate an edge feature aggregation model. This model can effectively utilize the dependencies in the contract. We also incorporate a multi-head attention mechanism, which can focus on multiple features.

3 Prepared Work

3.1 Ethereum

Understanding the related technologies of Ethereum holds considerable significance for our subsequent research into vulnerability detection in smart contracts on Ethereum. Deploying an Ethereum node is a fundamental step in accessing and participating in the Ethereum. A node not only stores the data of the entire blockchain but also participates in the verification and broadcast of transactions. Based on the node's configuration, it can serve as a full node, which preserves the full blockchain data and history, or as a lightweight node that only retrieves block header information to lessen resource usage. Participants can directly access Ethereum, send transactions, deploy smart contracts, and interact with other nodes by running a node. Regarding consensus mechanisms, Ethereum initially used the Proof of Work (PoW) algorithm, which secures the network and maintains decentralization by resolving complex mathematical puzzles. Nevertheless, due to the significant energy demands of PoW, Ethereum intends to shift towards the more efficient and eco-friendly Proof of Stake (PoS) algorithm. PoS algorithm selects nodes to create new blocks based on factors such as the amount of coins held and the duration of holding. This aims to reduce energy consumption and increase transaction processing speed.

3.2 Smart Contract Security Vulnerability

Among the hundreds of millions of smart contracts successfully deployed, there are approximately a dozen known types of smart contract vulnerabilities. Timestamp and reentrancy vulnerabilities are the most common and threaten blockchain security [23]. Given the diversity of blockchain platforms, smart contracts increasingly interact with external contracts in the current environment. Access control vulnerabilities in contracts will directly threaten the security of interactive contracts and trigger chain effects. Digging deeper into access control vulnerabilities is critical to understanding and preventing security risks across contracts, which helps ensure the robustness of the entire system. Current methods for detecting access control vulnerabilities still rely on traditional tools, and few studies have adopted deep learning technology for access control vulnerability detection. Therefore, this paper will focus on timestamp vulnerabilities, reentrancy vulnerabilities, and access control vulnerabilities. This section will briefly introduce these vulnerabilities.

3.2.1 Timestamp Vulnerability

The timestamp of each block usually represents the time miners mined the block. However, these timestamps are unreliable, as blockchain protocols allow miners to adjust the timestamps within a certain range (about 15 s in Ethereum). This means that if the logic of a smart contract overly relies on timestamps, miners might exploit this to manipulate the outcomes of the contract. A typical case of timestamp vulnerabilities is the "Rubixi" smart contract in 2016. This contract is a Ponzi scheme based on Ethereum. It contains a major timestamp dependency vulnerability, and this contract's revenue distribution mechanism relies on timestamps to determine when payments are made. Miners achieve unfair revenue distribution by manipulating timestamps, allowing some participants to gain undue benefits.

3.2.2 Reentrancy Vulnerability

Reentrancy vulnerability is among smart contracts' most common security vulnerabilities, especially on EVM-based platforms such as Ethereum. This vulnerability is because the smart contract executes calls to external contracts after performing critical operations (such as updating balances, changing ownership, etc.) and before state changes. Attackers use this mechanism to interrupt the execution of the contract and repeatedly call the contract's functions to steal Ethereum. Reentrancy vulnerability is the most famous vulnerability among smart contract vulnerabilities because it directly led to the most significant security incident since the establishment of Ethereum-"The DAO." The attacker exploited the reentrancy vulnerability in the DAO contract to repeatedly withdraw funds, resulting in the illegal transfer of approximately $50 million worth of Ethereum. This incident caused huge economic losses and led to the split between Ethereum and Ethereum Classic, which seriously affected the development of the Ethereum community.

3.2.3 Access Control Vulnerability

Access control vulnerabilities are a common security issue in smart contracts, primarily occurring when the contract's permission management is improperly set up. This type of vulnerability permits users to carry out actions they are not authorized, like withdrawing funds, modifying crucial settings, or conducting important transactions. A famous case is the multi-signature contract vulnerability of Parity Wallet. In this case, the contract allows an ordinary user to become the contract owner and perform destructive operations on the contract. This user inadvertently triggered a self-destruct function, resulting in numerous wallets being frozen and unable to access their funds. The damage caused by this breach is estimated to be as high as 150 million US dollars.

3.3 Graph Convolutional Neural Network

In recent years, neural networks have experienced rapid development and have been widely applied to graph data structures, which achieves significant accomplishments in image classification problems [24]. In traditional convolutional neural networks, convolution operations require sliding windows on structured grid data (such as images). This approach is inapplicable to non-Euclidean structures since the nodes in a graph lack a fixed count of neighbors, and their arrangement lacks a predetermined sequence. Handling non-Euclidean structures in real life has become an important subject of study. In 2013, Bruna introduced two classification methods for graph convolutional neural networks, one based on the spectral domain and the other on the spatial domain. Kipf et al. [25] proposed a message propagation network:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) \tag{1}$$

In the formula, $H^{(l+1)}$ represents the node at the lth layer, $\tilde{A} = A + I_N$ is the adjacency matrix plus the identity matrix to enhance self-loops, $\tilde{D}$ in the equation is a diagonal node degree matrix, which $\widetilde{D_{ii}} = \sum_j \widetilde{A_{ij}}$ is used for normalization, $W^{(l)}$ represents the weight matrix of the lth layer, and σ represents a nonlinear activation function.

Qian uses A^2 to enhance normalization and remove the degree matrix D from the original equation. It is represented as:

$$H^{(l+1)} = \sigma\left(\left(A^2 + I\right)H^{(l)}W^{(l)}\right) \tag{2}$$

3.4 Gated Recurrent Unit

Gated Recurrent Unit (GRU) [26] is a variant of Recurrent Neural Network (RNN) used to model and process sequence data. It has a gating mechanism that can effectively capture long-term dependencies and has fewer parameters. The main components of GRU include the Update Gate, Reset Gate, and Candidate Hidden State.

The update gate controls the extent to which the previous moment's hidden state is retained. It calculates a vector with values between 0 and 1 based on the input sequence data and the hidden state from the previous moment.

$$z_t = \sigma\left(W_z \cdot [x_t, h_{t-1}] + b_z\right) \tag{3}$$

Among them, x_t is the input of the current moment, h_{t-1} is the hidden state of the previous moment, W_z and b_z is the learnable weight matrix and bias term, σ is the sigmoid function.

The reset gate is used to decide whether to include the hidden state of the previous moment in the calculation of the current moment.

$$r_t = \sigma\left(W_r \cdot [x_t, h_{t-1}] + b_r\right) \tag{4}$$

Through the control of update gates and reset gates, GRU can decide the degree of retention and forgetting of information, thereby better handling long-term dependencies.

3.5 *Attention Mechanism*

Attention mechanisms allow neural network models to concentrate on the most significant segments of input data. This mechanism enables the model to prioritize processing the most critical information for the current task, thereby enhancing efficiency and effectiveness.

The multi-head attention mechanism extends the self-attention mechanism [27]. Researchers aspire for models to learn diverse behaviors through the same attention mechanism, subsequently amalgamating these behaviors as knowledge to capture a wide range of dependencies within a sequence. Specifically, this model distributes attention across multiple "heads," enabling the model to focus on the input data from several different perspectives or subspaces simultaneously. This approach enhances the model's flexibility and complexity, which allows it to better capture the diversity and complex relationships within the data. Its working principle is shown in Fig. 1.

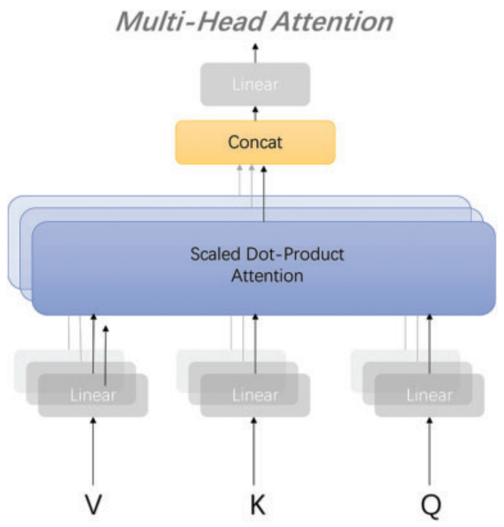


Figure 1: Multi-head attention mechanism

4 Our Method

This paper designs a feature extraction mode for timestamp, reentrancy, and access control vulnerabilities, aiming to extract the corresponding feature maps from the contract source code. During the construction of the feature graph, key functions and variables that may lead to vulnerabilities are considered nodes, and edges are constructed based on control dependencies. One-hot encoding converts The feature graph into the model's input data. The overall workflow of our experiment is shown in Fig. 2.

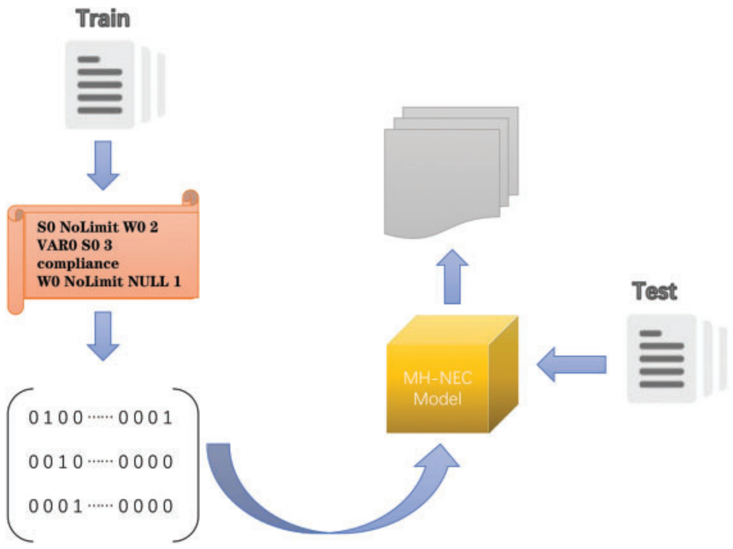


Figure 2: Smart contract vulnerability detection process

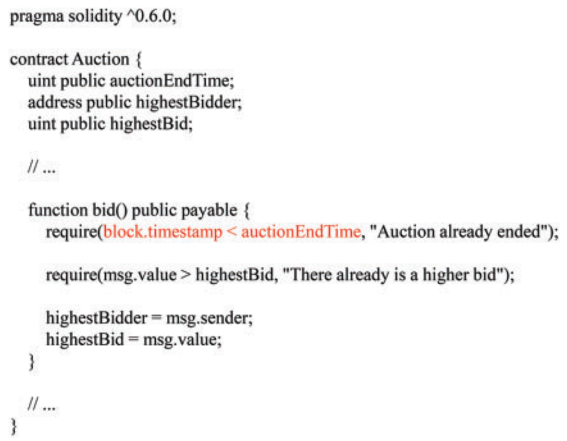
4.1 Feature Extraction Mode

In response to the following three types of vulnerabilities, we have innovated and improved upon the existing feature extraction mode [28,29].

Timestamp: Transactions on Ethereum often use timestamps as conditions for performing critical operations, which often leads to timestamp vulnerabilities. Static analysis tools typically examine the presence of *block.timestamp* to assess whether a contract contains a timestamp vulnerability. This paper adopts the following three methods to check for contract timestamp vulnerabilities.

- We check whether there exists *block.timestamp* in the contract. This is the basic step in determining timestamp vulnerabilities.
- From the perspectives of data flow and control flow, we check the value of *block.timestamp* is assigned to other variables or passed as a parameter to other functions. Such a situation indicates that the contract is at risk of being manipulated by miners.
- If *block.timestamp* is compared with time units (either directly with specific time values or with variables representing time), the contract may be at risk of being manipulated by miners' time.

The last one is the new mode proposed in this paper. Fig. 3 is the source code of an auction contract. The core logic of the auction contract is that if the timestamp of the current block exceeds the set auction end time, the contract will no longer accept new bids. However, this design is susceptible to timestamp manipulation attacks due to the manipulability of block timestamps. For example, a miner wants to bid after the auction ends. They could mine a block and intentionally set the timestamp slightly below the end of the auction, even though, in reality, that point has already passed. Checking whether *block.timestamp* in the contract is compared with variables, which is an effective means to check timestamp vulnerabilities.



```
pragma solidity ^0.6.0;

contract Auction {
    uint public auctionEndTime;
    address public highestBidder;
    uint public highestBid;

    // ...

    function bid() public payable {
        require(block.timestamp < auctionEndTime, "Auction already ended");

        require(msg.value > highestBid, "There already is a higher bid");

        highestBidder = msg.sender;
        highestBid = msg.value;
    }

    // ...
}
```

Figure 3: Example of auction contract

Reentrancy: Static analysis tools determine the existence of reentrancy vulnerabilities in a contract by checking for *calls.value*. Because *call.value* is a key operation in the transaction, but no gas limit is set. Based on this, we used the following three methods to check whether the contract had reentrancy vulnerabilities.

• In addition to inspecting *call.value* within functions, we also verify whether the functions invoke *call.value* enforces stringent access controls, like onlyOwner, private, and others.

• We examine the position of *call.value*. Appropriately reducing the user's balance before transactions is crucial in averting reentrancy vulnerabilities.

• In this paper, we examine whether there is a *spender.call* within the contract. We found many contracts use *spender.call* to execute transactions, which is unsafe. Similar to *call.value*, invoking external contracts through *spender.call* poses a risk of reentrancy vulnerabilities since it allows calling any function of the target contract (as long as it matches the function's signature) and is not limited by gas.

The last is the reentrancy vulnerability feature extraction mode proposed in this paper. In the CVE-2018-12703 event, the *approveAndCallcode* function in the smart contract is called an unverified *spender.call* method, which allowed attackers to transfer the entire balance of the contract to their accounts.

Access Control: Access control vulnerabilities occur due to inadequate management of permissions and access levels. Since access control vulnerabilities do not have key statements like *call.value* and *block.timestamp*, previous studies rarely developed feature extraction modes for access control vulnerabilities. This paper examines whether the contract has access control vulnerabilities from multiple angles.

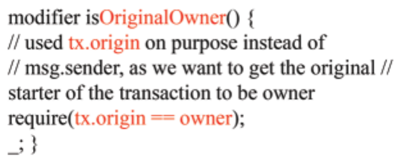
• Check the origin function and *Tx.origin* method in the contract. *Tx.origin* means that the authority belongs to the original initiator of the transaction call stack. Using *msg.sender* instead of *Tx.origin* can effectively avoid this attack.

• We check whether functions with key operations have strictly applied access restriction modifiers.

• We need to consider whether the function allows changing the contract owner. Many functions can add or remove administrators, change user access levels, etc.

• Fallback function lacks access control, and external contracts easily influence operations. We need to examine the interactions between the fallback function and external contracts.

The above four points are all the access control vulnerability feature extraction modes proposed in this paper. Fig. 4 is a case of a contract containing an access control vulnerability. In the *isOriginalOwner* modifier, the contract uses *tx.origin* for access control checking. *Tx.origin* identifies the originator of the entire transaction call stack rather than the immediate predecessor caller. This approach makes the contract vulnerable to "phishing" attacks.



```
modifier isOriginalOwner() {
// used tx.origin on purpose instead of
// msg.sender, as we want to get the original //
starter of the transaction to be owner
require(tx.origin == owner);
_; }
```

Figure 4: Part of the contract containing access control vulnerabilities

4.2 Feature Map

Based on the feature extraction mode proposed in this paper, we construct corresponding feature maps for the three vulnerabilities. The first step is to traverse all functions within the contract and retrieve key variables and key calls that may lead to vulnerabilities. We create nodes "S" and "W" for the filtered function. The second step is to check the access restrictions of the function and add corresponding attributes to it according to its access restrictions. When critical variables and calls involve dangerous operations, we create a node "VAR". For timestamp vulnerabilities, the key variable is considered to be "*block.timestamp*". Assigning "*block.timestamp*" to other variables or comparing it with time units is dangerous. For reentrancy vulnerabilities, "*call.value*" and "*spender.call*" are considered critical calls. We determine the vulnerability's risk level by judging the sequence of these operations relative to transaction actions. For access control vulnerabilities, functions involving origin, translation, and fallback are considered dangerous and likely to cause contract vulnerabilities. The code statements and symbols that appear above and in the above figure are explained in Table 1.

Table 1: Notation table

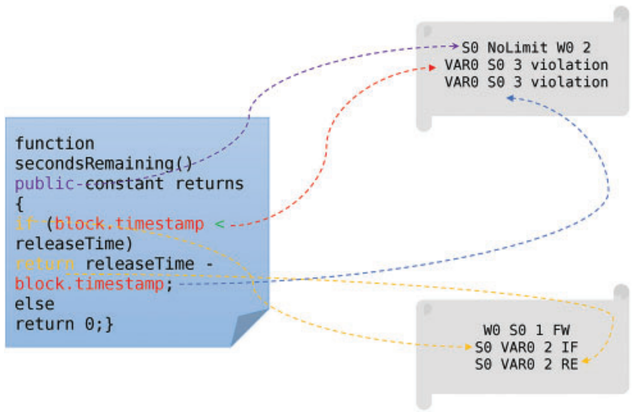
Variable or symbol	Definition
call.value	Function calling operation
spender.call	Function calling operation
block.timestamp	Representing the timestamp of the current block
Tx.origin	Returns the address where the current transaction was initiated
msg.sender	The address of the sender of the current function call
S	Create this node for functions involving key operations
W	Create this node for functions involving key operations
VAR	Create this node for functions involving dangerous operations

To reflect the degree of dependence between nodes, we use control flow as the starting point to abstract various conditional control statements into different types of edges, which can reflect the impact of control dependence on key variables in the contract. The corresponding representation of each control flow in the feature map is presented in Table 2.

Table 2: Edge representation of different control dependencies in the feature graph

Type	Control sentence
FW	normal
IF	if
GB	if else
GN	if then
FOR	while do
RE	return
AH	assert
RG	require
RH	if revert
IT	if throw

The feature tuples of node "S" and node "W" are (the starting node, access restrictions, ending node, execution sequence, and calling address). Since the node "VAR" involves dangerous operations, the feature tuple is represented as (start node, end node, execution order, degree of danger). The tuple of edge features is represented as (start node, end node, execution order, edge type). Fig. 5 shows a smart contract source code and its corresponding feature map.



```
function
secondsRemaining()
public-constant returns
{
if (block.timestamp <
releaseTime)
return releaseTime -
block.timestamp;
else
return 0;}
```

```
S0 NoLimit W0 2
VAR0 S0 3 violation
VAR0 S0 3 violation
```

```
W0 S0 1 FW
S0 VAR0 2 IF
S0 VAR0 2 RE
```

Figure 5: Smart contract source code and its corresponding feature graph

4.3 Model

4.3.1 Degree-Free Graph Convolutional Neural Network

In processing node features, this paper uses the improved graph convolutional neural network model. Based on the standard graph convolutional network, this model introduces several improvements to enhance its performance and adaptability. Its core structure includes multiple graph convolution layers. In the convolutional layer, we enhance the graph's connectivity by squaring the adjacency matrix (adj_sq parameter). It allows the model to consider neighbor information within two steps, thereby capturing a wider range of node relationships. This paper uses a self-connection (scale_identity parameter) operation to strengthen the influence of each node and improve the robustness of node feature representation. In addition, this paper uses a masking mechanism to distinguish valid and invalid nodes, which enables the model to process multiple graphs of different sizes in batch mode. The Rectified Linear Unit (ReLU) activation function after the convolutional layer introduces nonlinearity to the model, which can improve the model's ability to capture the characteristics of complex graph structures. A series of fully connected layers in the model process and refine the features outputted by the graph convolutional layers. Furthermore, the fully connected layers employ a Dropout regularization strategy to mitigate overfitting and augment the model's generalization capacity with varied data. Overall, the model is suitable for dealing with complex relationships between nodes.

DR-GCN does not rely on the degree of nodes when aggregating features, which gives it stronger generalization capabilities. DR-GCN is more effective for graphs with irregular degree distribution, and it does not rely on the degree matrix, which reduces the computational complexity. Fig. 6 shows the processing of node features in the experiment.

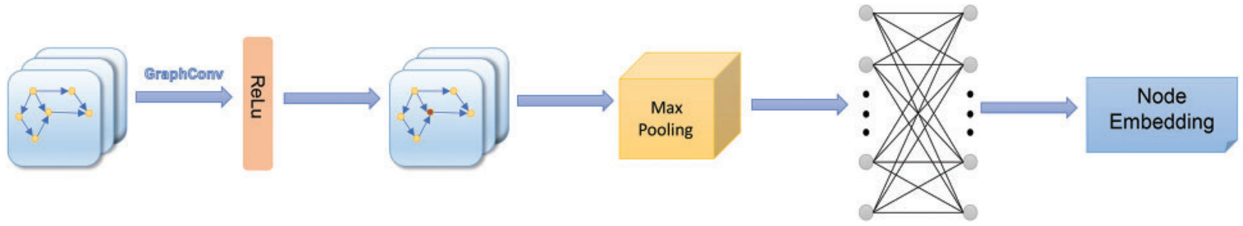


Figure 6: Node feature processing

4.3.2 Edge Aggregation Net

In previous experiments, the importance of edge features was often ignored when processing contract graph features based on deep learning. This paper develops an Edge Aggregation Net (EAN) to process edge features in contract graphs. This model refers to the logic of the graph attention neural network to more effectively aggregate neighbor information. When aggregating adjacent edge features, the attention mechanism assigns weights to edges of different importance. We have introduced gated units to strengthen the model's capability to handle edges of differing significance. Reset gates and update gates allow the model to dynamically adjust the information flow, which can update edge features more efficiently. During edge feature processing, if the reset gate is close to 0, it means that the model focuses more on the current edge feature. On the contrary, if the reset gate is close to 1, the model retains more previous information. The value of the update gate dictates the proportion of original features and current features that make up the new features. A higher value of the update gate means that more of the original features are retained, while a lower value of the update gate means that more current features are allowed to be added.

The processed edge features are sent to MultiLayer Perceptron (MLP) for further feature conversion. This step not only adds nonlinearity, but also further refines the edge features, which makes edge features more suitable for subsequent tasks. To prevent overfitting and enhance the model's generalization capabilities across various graph data types, we added a Dropout layer to the MLP. This strategy can improve the stability and reliability of the model. Fig. 7 shows the processing of edge features in the experiment.

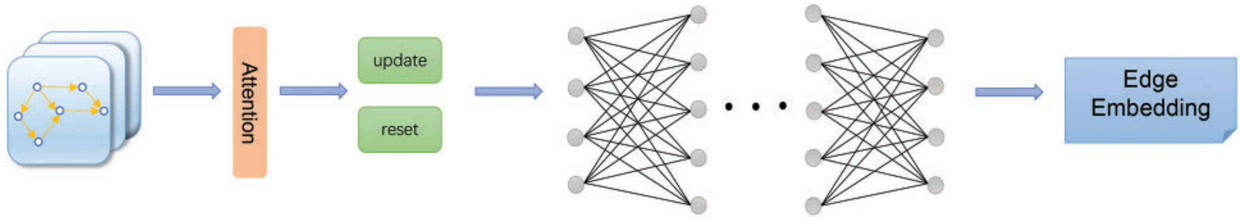


Figure 7: Edge feature processing

4.3.3 MH-NEC

This paper uses a multi-head attention mechanism to integrate the processed node and edge features better. First, we directly splice the node embedding processed by DR-GCN and the edge embedding processed by EAN. This splicing method can retain their original features to the greatest extent. The spliced features are input to the multi-head attention layer. The multi-head attention layer constructs four independent attention heads, each consisting of two fully connected layers and a Tanh activation function. The first fully connected layer maps the concatenated embedding to smaller dimensions, and the second fully connected layer further maps the features processed by the activation function for generating attention scores. The Tanh activation function ensures that the network can capture complex and nonlinear patterns in the input features, crucial for processing the complex relationships of nodes and edges in graph structures. The softmax function normalizes the computed attention scores into final attention weights. Attention weights are applied to the four heads to obtain their respective outputs. These outputs are concatenated and mapped back to the dimensions of the original input space through a fully connected layer. By learning various representations of input data in parallel, the model improves its comprehension and handling of input features. Since each head may focus on different aspects, the model can provide richer information for subsequent classification tasks. Fig. 8 shows how the processed node embedding and edge embedding are combined to form a global graph representation.

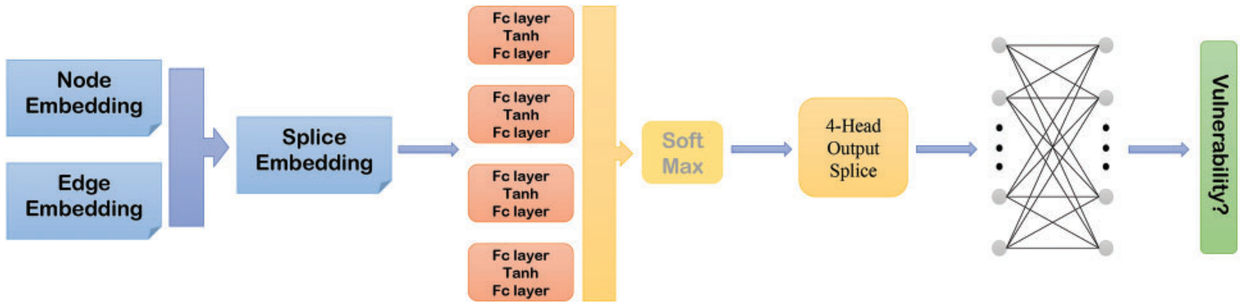


Figure 8: Using multi-attention mechanism to splice node embedding and edge embedding

5 Experiment

5.1 Dataset

Due to the current immaturity of smart contract vulnerability detection research, there are no official datasets or universally recognized reliable third-party datasets available at present. This paper combines the results of several literatures. A total of 5525 smart contracts containing 19363 functions were collected as the experimental data of this paper. To collect smart contracts without vulnerabilities, we utilized the wild dataset. They use nine static analysis tools to detect 47587 contracts on Ethereum. To guarantee the precision and dependability of the vulnerability-free dataset, we consider contracts identified as free of vulnerabilities by all nine tools mentioned in the literature as the preliminary vulnerability-free data for this experiment. However, relying solely on the tools' detection results cannot eliminate the possibility of false positives, so we manually verified a random sample of one-third of the contracts. Detailed analysis of the contract samples confirmed that these contracts did not contain any known security vulnerabilities, thereby enhancing the credibility of the dataset. The vulnerability-free dataset includes a total of 2741 smart contracts; it covers 7548 unique functions. To prove that the method proposed in this paper has good generalization ability, we combined the wild dataset and the Ethereum Smart Contracts (ESC) dataset to form the vulnerable data in the experiment. This part covers three common vulnerability types: Timestamp, reentrancy, and access control, with 2784 contracts and 11815 functions.

In addition, we deliberately introduced about 1120 pieces of noise data (vulnerable contracts containing critical calls) into the dataset to verify the robustness of our method. These experimental data consist solely of the source code of smart contracts genuinely deployed on Ethereum. Since these data are directly derived from the real world, this guarantees that our research and conclusions have substantial practical applicability and will contribute meaningfully to subsequent studies in smart contract vulnerability detection.

5.2 Experimental Environment

The experimental environment is shown in Table 3.

Table 3: Experimental environment

Hardware/environment	Description
GPU	Nvidia 1050Ti
CPU	Intel (R) Core (TM) i7-8750H CPU @ 2.20 GHz 2.21 GHz
RAM	16.0 GB
Framework	PyTorch 1.10.2

5.3 Evaluation Metrics

Since smart contract vulnerability detection is a two-classification problem, Accuracy, Recall, Precision, and F1 score are used as indicators to evaluate model performance.

Accuracy: The percentage of correct predictions in the total sample:

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}} \tag{5}$$

Precision refers to the predictive result, which means the probability that a sample is positive among all those predicted to be positive:

$$P = \frac{TP}{TP + FP} \tag{6}$$

The recall is for the original sample, and its meaning is the probability of being predicted as a positive sample among the actual positive samples:

$$R = \frac{TP}{TP + FN} \tag{7}$$

To be able to consider both precision and recall, the F1 score is usually used as an overall measure of model performance:

$$F1 = 2 * \frac{P * R}{P + R} \tag{8}$$

5.4 Results and Analysis

In this section, we compare MH-NEC with ten other detection methods on three vulnerabilities: Timestamp, reentrancy, and access control, and analyze the experimental results. These include four static analysis tools: Manticore, Smartcheck, Oyente, and Mythril; two analysis methods, Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN) based on traditional neural networks; and four analysis methods using graph neural networks: DR-GCN, TMP, Smart Contract Vulnerability Detection Mechanism (DM) [30], and Combining Graph feature and Expert patterns (CGE). DR-GCN and TMP are the earliest models that use graph neural networks for smart contract vulnerability detection tasks. On this basis, DM and CGE combined expert knowledge to further analyze the key functions that lead to vulnerabilities in the contract.

In addition, we performed ablation experiments on the MH-NEC model using the existing feature extraction mode and the improved feature extraction mode in this paper and analyzed the experimental results. Since the vulnerability-free data in this experiment all come from the results of tool detection, we focus on the recall rate and F1 score when compared with traditional tools [31]. The results from Table 4 indicate that traditional tools perform poorly in detecting timestamp vulnerabilities. As the tool with the best performance, Mythril's recall rate only reached 51.07%. The recall rates of LSTM and CNN reached 65.44% and 78.25%, which are significantly improved compared to traditional tools. The detection method that converts source code into a contract graph has an excellent performance in recall rate and F1 score, which fully confirms the huge potential of graph neural networks in the field of smart contract vulnerability detection. Thanks to its unique message propagation mechanism, the TMP model can effectively capture long dependencies, and its F1 score reached 82.04%. DM and CGE perform better due to expert mode and improved neural networks. This paper's method more effectively aggregates edge features and enhances the ability to analyze and track abnormal timestamps. The recall rate of this method in the timestamp vulnerability detection task reached 98.35%, far exceeding that of other smart contract vulnerability detection methods. The F1 score also reached 84.63%, which is 1.24% higher than the best existing method.

Table 4: Detection capabilities of different methods for timestamp vulnerabilities

Method	Accuracy	Recall	Precision	F1
Manticore	89.74	35.15	100.0	52.01
Smartcheck	90.63	40.78	100.0	57.93
Oyente	90.85	42.14	100.0	59.29
Mythril	92.26	51.07	100.0	67.61
LSTM	89.77	65.44	68.50	66.93
CNN	92.51	78.25	75.33	76.76
DR-GCN	93.55	88.74	75.04	81.32
TMP	93.73	90.49	75.04	82.04
DM	93.67	94.37	73.30	82.51
CGE	94.39	91.65	76.50	83.39
OURS	95.00	98.35	76.21	84.63

Table 5 shows the performance comparison of ten representative detection methods and our method on the reentrancy vulnerability detection task. Unlike timestamp vulnerability detection, individual traditional tools perform better than conventional neural networks in detecting reentrancy vulnerabilities. Among them, the recall rate of Mythril reached 59.73%, which is higher than 52.64% and 54.89% of LSTM and CNN. In reentrancy vulnerability detection, relying solely on checking the presence of *call.value* greatly increases the likelihood of false positives. Analyzing the sequence of reentrancy operations and transaction operations is the basis for determining whether a contract has a reentrancy vulnerability. During their processing, LSTM and CNN neglected this issue, whereas Mythril employed a control flow verification method. This method focuses more on the semantic relations between programs and has yielded impressive outcomes. DR-GCN only focuses on node characteristics in the contract graph, so its performance is biased. TMP, DM, and CGE retain the semantic relationship between programs, so they perform better on the reentrancy vulnerability detection task. Compared with previous methods, our method pays more attention to the semantic and logical relationships within the function and pays special attention to the potential threats of *spender.call*. Therefore, this paper's method has significantly improved reentrancy vulnerability detection. Compared with existing methods, our method improves the recall rate by 9.29% and the F1 score by 6.03%.

Table 5: Detection capabilities of different methods for reentrancy vulnerabilities

Method	Accuracy	Recall	Precision	F1
Manticore	68.09	18.90	100.0	31.79
Smartcheck	73.42	32.45	100.0	49.00
Oyente	78.73	45.95	100.0	62.97
Mythril	84.16	59.73	100.0	74.79
LSTM	75.86	52.64	78.99	63.18
CNN	79.75	54.89	89.62	68.09

(Continued)

Table 5 (continued)

Method	Accuracy	Recall	Precision	F1
DR-GCN	83.97	59.81	95.93	71.73
TMP	83.56	62.04	94.19	74.81
DM	90.35	78.68	96.09	86.52
CGE	90.88	79.92	96.27	87.33
OURS	94.84	87.97	98.23	92.55

The method of static analysis tools to detect this vulnerability by detecting function access restrictions is not rigorous, which can also be reflected in the results in Table 6. Since the methods using graph neural networks did not address the detection of access control vulnerabilities, we applied the feature extraction mode proposed in this paper to the DR-GCN, TMP, DM, and CGE neural networks. Compared with previous detection methods, our method reached 59.65%, 70.52%, and 61.36% in recall, precision, and F1 score, which can prove that our work has made up for the shortcomings in the field of access control vulnerability detection.

Table 6: Detection capabilities of different methods for access control vulnerabilities

Method	Accuracy	Recall	Precision	F1
Manticore	88.68	25.46	100.0	40.58
Smartcheck	85.88	34.83	100.0	51.66
Oyente	88.50	24.24	100.0	39.02
Mythril	91.21	42.16	100.0	59.31
LSTM	76.42	44.20	30.78	36.29
CNN	87.39	46.59	59.95	52.43
DR-GCN	86.88	51.93	57.56	54.60
TMP	87.62	52.74	58.60	55.52
DM	87.95	55.80	61.30	58.42
CGE	88.12	55.60	62.19	58.71
OURS	89.78	59.65	70.52	61.36

The experimental results of the above methods on the three vulnerabilities of timestamp, reentrancy, and access control show us that the detection capabilities of most static analysis tools are very poor. In vulnerability detection tasks, the performance of traditional neural networks is still mediocre, even inferior to that of static analysis tools. In contrast, graph neural networks achieve better results on all three types of vulnerabilities. It can be proved that graph neural network is very effective in smart contract vulnerability detection. Our method combines the advantages of previous methods and makes up for the shortcomings of previous methods. Compared with other graph neural network methods, our method has improved accuracy, recall, precision, and F1 score. Compared with previous methods, our method emphasizes the utilization of edge features, which can better capture the logical structure of the code and accurately identify potential vulnerabilities. Effective edge feature aggregation can capture the complex relationships between codes; it includes indirect relationships and longer dependency chains, which is highly effective in detecting complex vulnerabilities involving interactions among multiple components. The three figures (a), (b), and (c) in Fig. 9 further visually present the results of seven neural network methods on timestamp, reentrancy, and access control vulnerability detection tasks. Methods 1 to 7, respectively, represent LSTM, CNN, DR-GCN, TMP, DM, CGE, and methods of this paper. It can be seen that our method is better than other methods.

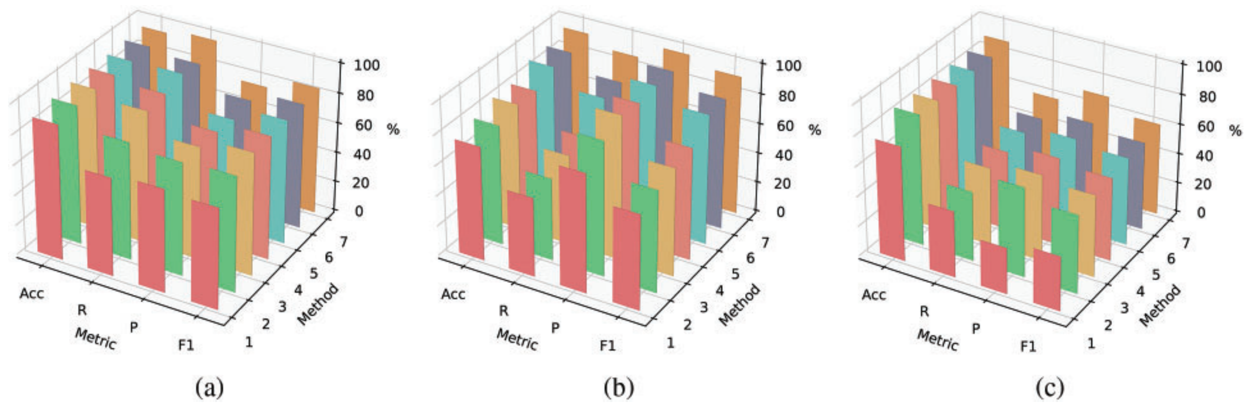


Figure 9: Performance comparison of seven neural networks on (a) timestamp vulnerability, (b) reentrancy vulnerability, and (c) access control vulnerability detection tasks

To better evaluate the model's overall performance under the condition of limited data volume, we used the 5-fold cross-validation method to conduct experiments in this paper. We divided the datasets into five subsets of equal size. During the experiment, each subset is used as the test set once, with the remaining k-1 subsets combined to form the training set. The average of 5 experimental results is regarded as the final result. The 5-fold cross-validation method can reduce the risk of overfitting and improve the accuracy of evaluating the model's generalization ability [32]. Table 7 shows the experimental results of 5-fold cross-validation on the timestamp vulnerability using this method. It can be concluded from the data in the table that the model's overall performance in this article is strong on the timestamp vulnerability dataset. The standard deviation is 3.66, indicating fluctuations in model performance.

Table 7: Experimental results using 5-fold cross-validation method in timestamp vulnerability

Accuracy	Recall	Precision	F1
93.59	94.83	68.16	78.80
95.47	100.0	78.91	86.90
94.06	97.92	73.59	82.56
95.78	100.0	77.82	85.48
96.09	99.00	82.58	89.42

Table 8 shows the results of using the 5-fold cross-validation method for the reentrancy vulnerability detection task. The table results show the model's high consistency and strong overall performance in the reentrancy vulnerability detection task. The performance variability of the model on different data subsets is relatively small, and the performance is more stable and consistent.

Table 8: Experimental results using 5-fold cross-validation method in reentrancy vulnerability

Accuracy	Recall	Precision	F1
95.54	88.56	97.32	92.48
94.53	86.93	98.91	92.27
93.75	86.25	98.06	91.46
95.20	88.92	98.70	93.29
95.20	89.20	98.20	93.23

Table 9 shows the results of using the 5-fold cross-validation method on the access control vulnerability dataset. The data shows that the performance of our method in access control vulnerability detection is relatively average and fluctuates greatly on different subsets. The detection methods for access control vulnerabilities and the balance of their datasets still require improvement.

Table 9: Experimental results using a 5-fold cross-validation method in access control vulnerability

Accuracy	Recall	Precision	F1
90.47	58.83	71.82	62.44
87.19	47.37	72.49	54.56
92.19	80.76	71.69	74.24
89.69	61.19	68.03	60.61
89.38	50.12	68.56	57.94

Timestamp and reentrancy vulnerabilities are two common security risks in smart contracts. To detect these vulnerabilities more accurately, the paper presents enhancements to the original feature extraction mode [33]. Specifically, we further check whether *block.timestamp* is used for comparison operations in timestamp vulnerability detection. We found that many contracts use spender in the reentrancy vulnerability detection task. *call* instead of *call.value* is a new potential risk point because it also leads to reentrancy vulnerabilities. Subsequently, we conducted ablation experiments to verify the effectiveness of the improvement methods proposed in this paper. Table 10 shows the performance comparison between the original feature extraction mode and the improved feature extraction mode in this paper on the timestamp vulnerability detection task and the reentrancy vulnerability detection task. The experimental results show that the performance of the improved feature extraction mode in the timestamp vulnerability detection task has slightly increased, with an F1 score improvement of 0.24% compared to before. The method presented in this paper has shown significant improvement across all metrics in detecting reentrancy vulnerabilities. Compared to the feature extraction mode, which focuses solely on calls and *value*, the method described in this paper has achieved an increase of 5.50% in the F1 score.

These results show that the method proposed in this paper can more effectively complete the task of detecting timestamp vulnerabilities and reentrancy vulnerabilities. Fig. 10 visually shows the role of the improved feature extraction mode in the vulnerability detection task.

Table 10: Improvement effect of feature extraction mode in this paper

Metrics	Timestamp		Reentrancy	
	Old mode	New mode	Old mode	New mode
Accuracy	94.94	95.00	90.66	94.84
Recall	98.35	98.35	79.53	87.97
Precision	75.93	76.21	96.15	98.23
F1	84.39	84.63	87.05	92.55

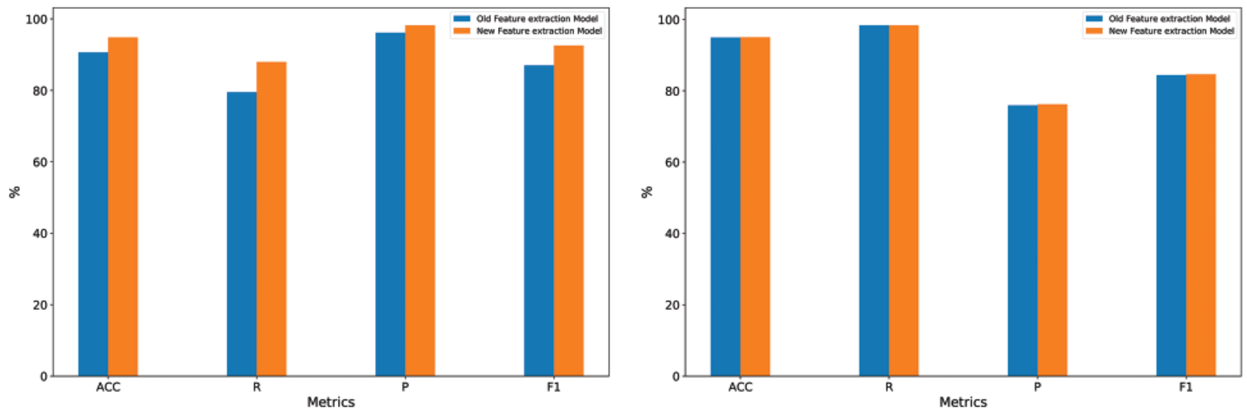


Figure 10: Performance impact of the improved feature extraction mode on timestamp vulnerability detection tasks and reentrancy vulnerability detection tasks

6 Conclusion

This paper proposes a smart contract vulnerability detection method based on feature graphs and multi-head attention mechanisms, improving complex contract structures' processing capabilities. Our method optimizes existing feature extraction modes and proposes innovative feature extraction modes for access control vulnerabilities. Additionally, the introduction of the multi-head attention mechanism not only ensures the precise capture of node features but also adequately accounts for the significance of edge features. It allows for a more detailed capture of the interactions among them, thereby improving the accuracy of vulnerability detection. The methods proposed in this paper have been validated for their effectiveness in detecting smart contract vulnerabilities through experiments conducted on three primary types of vulnerabilities. These results provide a new perspective for smart contract security research and a valuable methodological reference for future research and practice.

During the experiment, although we successfully demonstrated the effectiveness of our method in dealing with timestamp vulnerabilities, reentrancy vulnerabilities, and access control vulnerabilities. A key issue in our experiments is that we lack a diverse public dataset. Although the current dataset supports our conclusions, its limited scope and diversity might affect the universality of our results. In the future, we will mainly focus on collecting and constructing smart contract vulnerability datasets. Considering the achievements of this study in identifying timestamp vulnerabilities, reentrancy vulnerabilities, and access control vulnerabilities, we plan to develop a comprehensive smart contract vulnerability detection system in the future. The system covers existing vulnerability types and can adapt to emerging security threats, thereby providing more comprehensive and dynamic security protection.

Acknowledgement: Not applicable.

Funding Statement: This work was partly supported by the Gansu Province Higher Education Institutions Industrial Support Program: Security Situational Awareness with Artificial Intelligence and Blockchain Technology. Project Number (2020C-29).

Author Contributions: The authors confirm contribution to the paper as follows: Study conception and design: He Zhenxiang, Zhao Zhenyu; data collection: Zhao Zhenyu, Liu Yanlin; analysis and interpretation of results: Zhao Zhenyu, Chen Ke; draft manuscript preparation: Zhao Zhenyu. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: Data is openly available in a public repository. The data that support the findings of this study are openly available in ESC at https://github.com/Messi-Q/Smart-Contract-Dataset and sbwild at https://github.com/smartbugs/smartbugs-wild. Because all the data and vulnerabilities used in this article come from published papers, and these papers are publicly available. My research will not potentially impact the manufacturer's reputation and legal and ethical responsibilities, and I am willing to cooperate with the manufacturer under any circumstances.

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

References

[1] A. M. Shaker, A. Alsamman, and K. Chebbi, "Feedback trading in the cryptocurrency market," *Stud. Econ. Finance*, vol. 41, no. 1, pp. 46–63, May 2023. doi: 10.1108/SEF-02-2023-0096.

[2] T. H. D. Huang, "Hunting the Ethereum smart contract: Color-inspired inspection of potential attacks," 2018. Accessed: Mar. 21, 2024. [Online]. Available: https://arxiv.org/abs/1807.01868

[3] P. Momeni, Y. Wang, and R. Samavi, "Machine learning model for smart contracts security analysis," in *Proc. Int. Conf. on Privacy, Security and Trust (PST)*, Fredericton, NB, Canada, 2019, pp. 1–6.

[4] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang and Q. He, "Smart contract vulnerability detection using graph neural network," in *Proc. Int. Conf. on Int. Joint Conf. on Artificial Intelligence (IJCAI)*, Yokohama, Japan, 2020, pp. 3283–3290.

[5] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu and X. Wang, "Combining graph neural networks with expert knowledge for smart contract vulnerability detection," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 2, pp. 1296–1310, Jul. 2021.

[6] L. Luu, D. H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. SIGSAC*, New York, NY, USA, 2016, pp. 254–269.

[7] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. SIGSAC*, New York, NY, USA, 2018, pp. 67–82.

[8] B. Mueller, "Mythril—Reversing and bug hunting framework for the ethereum blockchain," 2017. Accessed: Mar. 21, 2024. [Online]. Available: https://github.com/Consensys/mythril

[9] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. Int. Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, Montreal, QC, Canada, 2019, pp. 8–15.

[10] M. Mossberg *et al.*, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proc. Int. Conf. on Automated Software Engineering (ASE)*, San Diego, CA, USA, 2019, pp. 1186–1189.

[11] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts," in *Proc. Int. Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, New York, NY, USA, 2018, pp. 9–16.
[12] M. Shafiq, Z. Tian, A. K. Bashir, X. Du, and M. Guizani, "CorrAUC: A malicious bot-IoT traffic detection method in IoT network using machine-learning techniques," *IEEE Internet Things J.*, vol. 8, no. 5, pp. 3242–3254, Mar. 2021. doi: 10.1109/JIOT.2020.3002255.
[13] M. Shafiq, Z. Tian, A. K. Bashir, X. Du, and M. Guizani, "IoT malicious traffic identification using wrapper-based feature selection mechanisms," *Comput. Secur.*, vol. 94, no. 4, pp. 101863, Jul. 2020. doi: 10.1016/j.cose.2020.101863.
[14] L. Zhang *et al.*, "CBGRU: A detection method of smart contract vulnerability based on a hybrid model," *Sens.*, vol. 22, no. 9, pp. 3577–3600, Mar. 2022. doi: 10.3390/s22093577.
[15] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo and J. Grundy, "SmartEmbed: A tool for clone and bug detection in smart contracts through structural code embedding," in *Proc. IEEE Int. Conf. on Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA, 2019, pp. 394–397.
[16] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2Vec: Learning contract-wide code representations for vulnerability detection on Ethereum smart contracts," in *Proc. ACM Int. Symp. on Blockchain and Secure Critical Infrastructure (BSCI)*, New York, NY, USA, 2021, pp. 47–59.
[17] W. B. Cavnar and J. M. Trenkle, "N-gram-based text categorization," in *Proc. Symp. on Document Analysis and Information Retrieval (SDAIR)*, Las Vegas, NV, USA, 1994, pp. 14.
[18] M. Rossini, M. Zichichi, and S. Ferretti, "On the use of deep neural networks for security vulnerabilities detection in smart contracts," in *Proc. IEEE Int. Conf. on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, Atlanta, GA, USA, 2023, pp. 74–79.
[19] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," 2017. Accessed: Mar. 21, 2024. [Online]. Available: https://arxiv.org/pdf/1711.00740.pdf
[20] Y. Zou, B. Ban, Y. Xue, and Y. Xu, "CCGraph: A PDG-based code clone detector with approximate graph matching," in *Proc. Int. Conf. on Automated Software Engineering (ASE)*, New York, NY, USA, 2020, pp. 931–942.
[21] M. Cai, Y. Jiang, C. Gao, H. Li, and W. Yuan, "Learning features from enhanced function call graphs for Android malware detection," *Neurocomputing*, vol. 423, pp. 301–307, Jan. 2021. doi: 10.1016/j.neucom.2020.10.054.
[22] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "*BGNN4VD:* Constructing bidirectional graph neural-network for vulnerability detection," *Inf. Softw. Tech.*, vol. 136, pp. 106576, Aug. 2021. doi: 10.1016/j.infsof.2021.106576.
[23] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proc. Int. Conf. on Software Engineering (ICSE)*, New York, NY, USA, 2020, pp. 530–541.
[24] J. Wang, B. Wei, J. Zhang, X. Yu, and P. K. Sharma, "An optimized transaction verification method for trustworthy blockchain-enabled IIoT," *Ad Hoc Netw.*, vol. 119, no. 1, pp. 102526, Aug. 2021. doi: 10.1016/j.adhoc.2021.102526.
[25] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016. Accessed: Mar. 21, 2024. [Online]. Available: https://arxiv.org/abs/1609.02907
[26] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014. Accessed: Mar. 21, 2024. [Online]. Available: https://arxiv.org/abs/1406.1078
[27] Z. Niu, G. Zhong, and H. Yu, "A review on the attention mechanism of deep learning," *Neurocomputing*, vol. 452, pp. 48–62, Sep. 2021. doi: 10.1016/j.neucom.2021.03.091.
[28] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. Int. Conf. on Automated Software Engineering (ASE)*, New York, NY, USA, 2018, pp. 259–269.
[29] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," 2018. Accessed: Mar. 21, 2024. [Online]. Available: https://sukritkalra.github.io/data/papers/zeus.pdf

[30] Z. Liu, M. Jiang, S. Zhang, J. Zhang, and Y. Liu, "A smart contract vulnerability detection mechanism based on deep learning and expert rules," *IEEE Access*, vol. 11, pp. 77990–77999, Jul. 2023. doi: 10.1109/ACCESS.2023.3298048.

[31] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He and S. Ji, "Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion," 2021. Accessed: Mar. 21, 2024. [Online]. Available: https://arxiv.org/abs/2106.09282

[32] H. Wu, H. Dong, Y. He, and Q. Duan, "Smart contract vulnerability detection based on hybrid attention mechanism model," *Appl. Sci.*, vol. 13, no. 2, pp. 770–794, Jan. 2023. doi: 10.3390/app13020770.

[33] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, "Towards automated reentrancy detection for smart contracts based on sequential models," *IEEE Access*, vol. 8, pp. 19685–19695, Jan. 2020. doi: 10.1109/ACCESS.2020.2969429.