Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2025.061455

ARTICLE

# GMS: A Novel Method for Detecting Reentrancy Vulnerabilities in Smart Contracts

**Dawei Xu[1,2], Fan Huang[1], Jiaxin Zhang[1], Yunfang Liang[1], Baokun Zheng[3,*] and Jian Zhao[1]**

[1]School of Computer Science, Changchun University, Changchun, 130012, China
[2]School of Computer Science and Technology, Beijing Institute of Technology, Beijing, 100081, China
[3]School of Information Management for Law, China University of Political Science and Law, Beijing, 102249, China

*Corresponding Author: Baokun Zheng. Email: zhengbk@cupl.edu.cn

Received: 25 November 2024; Accepted: 13 January 2025; Published: 16 April 2025

**ABSTRACT:** With the rapid proliferation of Internet of Things (IoT) devices, ensuring their communication security has become increasingly important. Blockchain and smart contract technologies, with their decentralized nature, provide strong security guarantees for IoT. However, at the same time, smart contracts themselves face numerous security challenges, among which reentrancy vulnerabilities are particularly prominent. Existing detection tools for reentrancy vulnerabilities often suffer from high false positive and false negative rates due to their reliance on identifying patterns related to specific transfer functions. To address these limitations, this paper proposes a novel detection method that combines pattern matching with deep learning. Specifically, we carefully identify and define three common patterns of reentrancy vulnerabilities in smart contracts. Then, we extract key vulnerability features based on these patterns. Furthermore, we employ a Graph Attention Neural Network to extract graph embedding features from the contract graph, capturing the complex relationships between different components of the contract. Finally, we use an attention mechanism to fuse these two sets of feature information, enhancing the weights of effective information and suppressing irrelevant information, thereby significantly improving the accuracy and robustness of vulnerability detection. Experimental results demonstrate that our proposed method outperforms existing state-of-the-art techniques, achieving a 3.88% improvement in accuracy compared to the latest vulnerability detection model AME (Attentive Multi-Encoder Network). This indicates that our method effectively reduces false positives and false negatives, significantly enhancing the security and reliability of smart contracts in the evolving IoT ecosystem.

**KEYWORDS:** Smart contract; Internet of Things; reentrancy vulnerabilities; graph neural network

## 1 Introduction

With the rapid development of the Internet of Things (IoT), billions of devices are now interconnected via the Internet, providing unprecedented convenience for a wide range of applications. However, the widespread adoption of IoT also brings with it significant security risks, including data breaches, device tampering, and unauthorized access. To address these challenges, the decentralized nature of blockchain technology is increasingly viewed as an ideal solution for IoT security. Through smart contracts, blockchain can automatically execute and verify transactions between devices, ensuring data authenticity and immutability. Nevertheless, smart contracts themselves are not immune to security vulnerabilities, such as reentrancy attacks and overflow issues. If left unchecked, these vulnerabilities can compromise the security of the entire system. Therefore, ensuring the security of smart contracts is a critical aspect that cannot be

overlooked in the integration of blockchain and IoT. One of the key reasons Ethereum is often referred to as "Blockchain 2.0" is its introduction of smart contracts, which bring programmability to the blockchain. Smart contracts are Turing-complete programs that operate on the Ethereum Virtual Machine (EVM). Initially proposed by Szabo [1], the concept of smart contracts was envisioned as a way to embed contracts into physical entities, creating flexible and controllable digital assets. Szabo believed that smart contracts could be constructed using algorithms and computer networks, eliminating the need for human intervention. Once deployed on the network, these contracts automatically execute according to predefined rules.

Today, blockchain, exemplified by Ethereum, has brought its vision into reality. On one hand, smart contracts serve as a technological tool that empowers many industries (such as the Internet of Things [2], agriculture [3], privacy protection [4,5], etc.); on the other hand, with the growth of decentralized finance (DeFi), which attracts an increasing amount of capital, smart contracts now control and manage a large volume of digital assets. As of March 2024, the market value of the decentralized finance sector has exceeded $1 trillion. With such widespread application and the attraction of substantial funds, smart contracts have become the target of numerous hacker attacks, resulting in significant thefts every few days, with total losses from attacks reaching $8.56 billion [6]. Since 2016, the DAO (Decentralized Autonomous Organization) incident [7,8], which marked the beginning of smart contract security issues and was severe enough to lead to an Ethereum hard fork. In December 2024, GemPad—a no-code smart contract deployment platform—was the victim of a hack targeting its smart contracts across the Ethereum, BNB Chain, and Base networks. The attacker stole an estimated $1.9 million of locked assets from the protocol by exploiting a reentrancy vulnerability in the project's smart contracts.

As awareness of smart contract security has grown, several studies have focused on designing security defenses, such as secure programming libraries and smart contract upgrade mechanisms. However, these solutions either offer limited protection against reentrancy attacks or may introduce new security vulnerabilities if not implemented correctly. Other studies have addressed the issue from the perspective of vulnerability detection, which can be broadly categorized into three categories: dynamic execution, static analysis, and deep learning-based methods. The former excessively relies on developer-defined rules and, as the focus on smart contract security defenses and the attack surface for reentrancy by malicious attackers expand, exhibits high false positive and false negative rates. Zheng et al. [9] have studied common reasons for false positives in traditional vulnerability detection tools regarding reentrancy vulnerabilities: 1) Access Control: When a reentrant function can only be called by other internal functions of the contract or is protected by function modifiers like onlyOwner() or reentrancy locks, it is difficult for malicious attackers to exploit the reentrant function even if it is vulnerable. 2) No State Change After External Calls: If a function can be called externally multiple times in a single transaction, previous reentrancy detection tools might flag the contract as reentrant. However, in some cases, the external call might not involve asset-related actions, and the contract's state may not change after the external call. 3)Transfer/Send Mechanisms: Unlike the call.value() function, transfer() and send() modify the maximum gas limit to 2300 gas, which is insufficient to invoke a contract or write to any storage variables. Deep learning-based approaches, on the other hand, typically treat smart contracts as text sequences and employ natural language processing algorithms for classification. However, these methods often fail to fully exploit the rich control flow and data flow information inherent in smart contracts. This paper proposes a pattern-matching approach for detecting reentrancy vulnerabilities in smart contracts. The method is based on the root causes of vulnerabilities that have led to reentrancy attacks in recent years, as well as the research by Zheng et al. [9]. It aims to extract features directly related to these vulnerabilities. A Graph Attention Neural Network is employed to extract smart contract graph embeddings from the contract's structure. Finally, the SE-Net (Squeeze-and-Excitation Networks) network is used to fuse the two sets of extracted features. This approach not only effectively leverages the rich control flow and

data flow information within smart contracts but also improves the detection accuracy for the reentrancy vulnerability pattern defined in this paper. It's worth noting that an increasing number of researchers prefer to train smart contract vulnerability detection models on the cloud, which makes data security and privacy issues [10,11] particularly important.

Based on the investigation of the types of vulnerabilities that have caused reentrancy attacks in recent years, and in combination with the research by Zheng et al. [9], particularly the causes of tool false positives, vulnerabilities, and the new type of reentrancy vulnerability (read-only reentrancy), some rule patterns for reentrancy vulnerabilities have been established.

## 2 Background

### *2.1 Ethereum*

The technical foundation of Ethereum lies in its unique blockchain architecture, which supports the execution of smart contracts. Smart contracts are programs that automatically execute contract code on the blockchain without the control or verification of a centralized authority. This capability makes Ethereum not just a digital currency platform, but a comprehensive decentralized application (DApp) platform. Compared to Bitcoin, which primarily focuses on being a digital cash system, Ethereum offers a wider range of possibilities, including but not limited to financial transactions, supply chain management, Metaverse [12], Decentralized Identity System [13], and various other applications [14]. A key feature of Ethereum is its scalability. By introducing technologies such as sharding and state channels, Ethereum can handle more transactions and improve network efficiency [15]. The implementation of these technologies allows Ethereum to significantly enhance its processing capacity and transaction speed without compromising security and decentralization principles. Additionally, Ethereum has adopted the Proof of Stake (PoS) mechanism as a replacement for Bitcoin's energy-intensive Proof of Work (PoW) mechanism. This change not only reduces energy consumption but also improves the network's security and accessibility [15]. The PoS mechanism incentivizes honest participants to maintain the network's security and health, thereby lowering the economic cost of attacking the network.

Despite significant advancements in technology and applications, Ethereum faces several challenges and limitations. One of the most pressing issues is the security of smart contracts, which remain vulnerable to exploitation by malicious code or inherent flaws. Moreover, as the number of decentralized applications (DApps) grows, network congestion and high transaction fees have become ongoing challenges for users. Nonetheless, through its innovative technologies and flexible application framework, Ethereum has secured an important position in the fields of cryptocurrency and blockchain. It not only provides users with a powerful decentralized platform but also drives the further development and application of blockchain technology. However, to achieve long-term sustainable growth, Ethereum needs to continuously address existing challenges and continue exploring new possibilities.

### *2.2 EVM*

The Ethereum Virtual Machine (EVM) is a key component built on the Ethereum blockchain, providing a decentralized platform for the execution of smart contracts and allowing for the execution and state updates of these contracts. The EVM is a Turing-complete virtual machine. It operates based on a globally accessible single-state model, which is altered by applying machine-level instructions known as opcodes. These opcodes form the foundation of programs executed on Ethereu [16]. Every smart contract running on Ethereum is a collection of these opcodes, which are validated by the nodes in the network and executed in a specific

order. This design enables the EVM to handle complex computations and logic, thereby supporting the development of various decentralized applications (DApps).

To improve the performance and security of the EVM, researchers have proposed various enhancement methods. For example, EVMTracer is an offline tool that can track and analyze the execution characteristics of smart contracts, such as the degree of parallelization and computational redundancy, helping to optimize EVM performance [17]. Additionally, there is research focused on enhancing the EVM to prevent dangerous transactions with vulnerabilities in real-time, involving steps such as monitoring strategy definition, opcode structure maintenance, and EVM instrumentation. Although the EVM offers powerful functionality, it also faces challenges such as scalability and transaction cost issues. To address these problems, the Ethereum community has developed various Layer 2 (L2) solutions, such as the 1DLT system, which allows for the rapid deployment of EVM-based blockchains while overcoming the drawbacks of traditional L2 systems [18].

Moreover, the economic model of the EVM is one of the key factors in its success. Ethereum uses a mechanism called "gas" to measure and charge for the execution cost of smart contracts. Gas is a virtual, finite resource used to pay for the service fees required to execute smart contracts. This mechanism not only incentivizes users to use network resources efficiently but also promotes the decentralization and security of the network [16].

### *2.3 Ethereum Smart Contract*

Ethereum smart contracts are a significant application of blockchain technology, enabling the automatic execution and management of contract terms on a decentralized network. The concept of smart contracts was first proposed by computer scientist Szabo [19] and later realized through the Ethereum platform. Ethereum not only supports smart contracts but also provides a comprehensive programming environment, allowing developers to create complex logic and functionality using the Solidity language.

The core advantages of smart contracts lie in their automation and immutability. Once the conditions are met, smart contracts can automatically execute predetermined terms without the need for any intermediaries or third-party intervention. This feature gives smart contracts broad application prospects in fields such as finance, supply chain management, and voting systems [20,21].

However, the security of smart contracts has always been a challenge in both research and practice. Since the code of a smart contract cannot be altered once deployed on the blockchain, any vulnerabilities could lead to significant economic losses. Additionally, as the use of smart contracts increases, improving their scalability and privacy protection has become a research hotspot. For example, the Arbitrum system significantly enhances processing capacity and privacy protection by shifting the verification of the virtual machine's behavior off-chain [22].

### *2.4 Reentry Vulnerability*

Reentrancy vulnerabilities can be categorized into three types based on the manner of reentrancy: single-function reentrancy, cross-function reentrancy, and read-only reentrancy.

Single-function Reentrancy: Occurs when the vulnerable function and the function exploited by the attacker are the same. For example, as shown in Fig. 1, Contract A provides a withdrawal function that lacks access control and has public visibility. A malicious attacker can design an evil contract to repeatedly call the withdrawal function using the fallback mechanism, draining all the funds from Contract A.

Cross-function Reentrancy: Occurs when the vulnerable function and the function exploited by the attacker are different. In this case, the vulnerable function has access control or visibility restrictions. The attacker finds other functions within the contract that nest calls to the vulnerable function, exploiting this to perform a cross-function reentrancy attack.

```
contract A{
        mapping(address =>  uint )private Balance;
        ...
        function withdraw( uint  amount) public {
        require(amount < Balance[ msg.sender ]);
        msg.sender.call.value (amount)();
        Balance[msg.sender ]  -= amount;
        }
}
```

```
contract evil{
        address A_add = 0202ff..22;
        ...
        function() payable{
                if(count++ <10)
                        A_add.withdraw ();
        }
}
```

**Figure 1:** A simple example of single function reentrant

Read-only Reentrancy: Occurs when the contract state remains unchanged during reentrancy, allowing the attacker to profit from this situation. While single-function reentrancy and cross-function reentrancy aim to repeatedly call transfer methods to withdraw funds from the contract, read-only reentrancy exploits the fact that the contract state has not yet been updated during reentrancy, enabling the attacker to make a profit. For example, as shown in Fig. 2, Contract A allows for token staking and withdrawal, and provides a query function to check prices based on the total amount of tokens and staked native tokens. Contract B offers staking and withdrawal functions, which depend on Contract A's get_price() function for price queries. This creates a risk of read-only reentrancy between the two contracts. An attacker could use a flash loan to acquire a large amount of native tokens and call Contract A's deposit() function, followed immediately by calling the withdrawal function withdraw(). During the withdrawal process, Contract A would call the attacker's fallback function. At this point, the attacker could again call the staking function in Contract B. Because Contract A's state has not yet fully updated, the token price may be lower than normal, allowing the attacker to exploit this vulnerability to obtain more tokens.

```
contract ContractA {
  uint256 private _totalSupply;
  uint256 private _allstake;
  mapping (address => uint256) public _balances;
  bool check=true;

  modifier noreentrancy(){
    require(check);
    check=false;
    _;
    check=true;
  }

  function get_price() public view virtual returns (uint256) {
    if(_totalSupply==0||_allstake==0) return 10e8;
    return _totalSupply*10e8/_allstake;
  }

  function deposit() public payable noreentrancy(){
    uint256 mintamount=msg.value*get_price()/10e8;
    _allstake+=msg.value;
    _balances[msg.sender]+=mintamount;
    _totalSupply+=mintamount;
  }

  function withdraw(uint256 burnamount) public noreentrancy(){
    uint256 sendamount=burnamount*10e8/get_price();
    _allstake-=sendamount;
    payable(msg.sender).call{value:sendamount}("");
    _balances[msg.sender]-=burnamount;
    _totalSupply-=burnamount;
  }
}
```

```
contract ContractB {
  ContractA contract_a;
  mapping (address => uint256) private _balances;
  bool check=true;

  modifier noreentrancy(){
    require(check);
    check=false;
    _;
    check=true;
  }

  function setcontracta(address addr) public {
    contract_a = ContractA(addr);
  }

  function depositFunds() public payable noreentrancy(){
    uint256 mintamount=msg.value*contract_a.get_price()/10e8;
    _balances[msg.sender]+=mintamount;
  }

  function withdrawFunds(uint256 burnamount) public payable noreentrancy(){
    _balances[msg.sender]-=burnamount;
    uint256 amount=burnamount*10e8/contract_a.get_price();
    msg.sender.call{value:amount}("");
  }
  function balanceof(address acount)public view returns (uint256){
    return _balances[acount];
  }
}
```

**Figure 2:** An example of read-only re-entry

## 3 Method of This Article

The overall overview of the method presented in this paper is illustrated in Fig. 3 and consists of three main parts: Reentrancy Pattern Recognition Phase: Identifies potential reentrancy vulnerability patterns from the source code based on predefined rules. Contract Semantic Graph Construction and Normalization: Extracts control flow and data flow semantics from the source code and highlights key nodes. Vulnerability Detection Phase: Utilizes a Graph Attention Neural Network to convert the normalized contract graph into graph embeddings, combining rule features and graph features to output detection results. The following sections will detail each of these three parts.

### *3.1 Reentry Vulnerability Feature Extraction*

Drawing on the work of Zheng et al. [9] and research into attack incidents caused by reentrancy vulnerabilities [16], this paper summarizes and implements the following three types of reentrancy vulnerability rules using keyword matching and syntax analysis. Table 1 lists some of the relevant keywords collected in this paper. The first pattern is CallValueInvocation, which checks whether the function contains a call to call.value. The second pattern is CallAfterVary, which examines whether there are state changes with financial risk after an external call is completed. This pattern is primarily designed for read-only reentrancy, as it checks for risks if the state remains unchanged after the external call. The third pattern is CrossFunction, which checks for additional attack surfaces when a function has access control. The related reentrancy vulnerability pattern keywords are mainly collected from open-source Ethereum smart contracts, GitHub, and blogs.

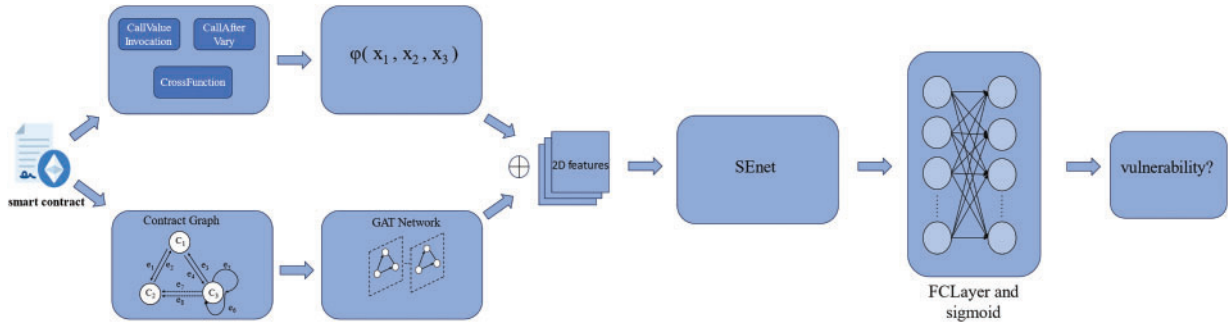


**Figure 3:** The overall architecture of the method described in this article

**Table 1:** Keywords related to re-entry vulnerability characteristics

| Type | Keywords |
|---|---|
| Call | "call.value" |
| Funcion access | "private", "onlyOwner", "internal", "onlyGovernor", "onlyCommittee", "onlyAdmin", "onlyOwner", "onlyCongressMembers", "preventReentrancy", "noReentrancy"... |
| Relevate account | "balances[msg.sender]", "Accounts[msg.sender]", "[msg.sender]", "[from]", "[to]", "[_to]", "Bids[msg.sender]", "tokenManage[token_]"... |

### *3.2 Construction and Standardization of Contract Semantic Graph*

Inspired by the work of [23], the contract graph defines nodes and edges derived from smart contract functions. Nodes include core nodes, ordinary nodes, and fallback nodes, while edges comprise control flow, data flow, and fallback edges. Core nodes represent key functions and important variables used to detect reentrancy vulnerabilities in smart contracts, ordinary nodes represent function calls and variable assignments that assist in vulnerability detection, and fallback nodes are special nodes representing the fallback functions in the smart contract.

To illustrate how to represent the source code of a smart contract as a contract semantic graph, this paper provides a simplified example in Fig. 4. For instance, consider Contract A, where the goal is to assess whether its withdrawal function has a reentrancy vulnerability.

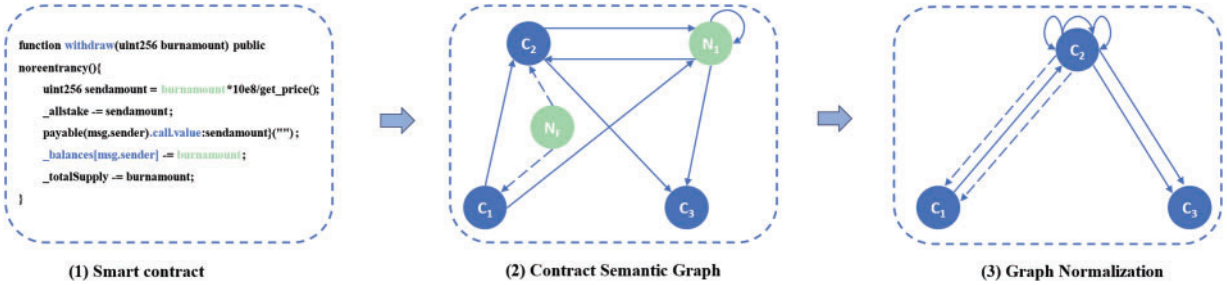


**Figure 4:** Construction and standardization of contract semantic graph

Given that different functions produce graphs with varying structures, and to standardize the contract semantic graph, this paper is inspired by [23]. The normalization process involves removing all ordinary nodes and aggregating their features into the nearest core nodes. For example, in Fig. 4, the ordinary node N1 is removed, and its features are aggregated into the nearest core nodes C2 and C3. For ordinary nodes with multiple neighboring core nodes, their features are propagated to all these core nodes. Edges connected to the removed ordinary nodes are retained, but their start or end nodes are shifted to the corresponding core nodes.

### *3.3 Vulnerability Detection*

#### *3.3.1 GAT*

After obtaining the normalized contract semantic graph, this paper uses a Graph Attention Network [24] (GAT) architecture to learn graph embeddings. The process of extracting graph embeddings consists of two phases: the message propagation phase and the aggregation phase. In the message propagation phase, the neural network passes information according to the sequence of edges. Subsequently, GAT aggregates the neighboring nodes of each node to update its hidden state.

$$\vec{h}_i' = \sigma\left(\sum_{j\in N_i} \alpha_{ij} W\vec{h}_j\right) \tag{1}$$

$\sigma$ is the nonlinear activation function, $N_i$ represents the neighboring nodes of node i in the graph, and W is the weight matrix. The attention coefficient $\alpha_{ij}$ is given by the following formula:

$$\alpha_{ij} = \frac{\exp\left(\mathcal{T}\left(\vec{a}^T\left[W\vec{h}_i \oplus W\vec{h}_j\right]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\mathcal{T}\left(\vec{a}^T\left[W\vec{h}_i \oplus W\vec{h}_k\right]\right)\right)} \tag{2}$$

$\oplus$ denotes concatenation, $\vec{a}^{T}$ represents the weight vector of a single-layer MLP (Multilayer Perceptron), and $\mathcal{T}$ is the Leaky ReLU (Rectified Linear Unit) function. After iterating through all the edges, GAT generates the final high-level graph semantic embedding by aggregating all parameters and the hidden states of the nodes in the graph.

$$\mathcal{G} = \sum_{i=1}^{V} \sigma\left(\mathrm{P}_{\mathrm{gate}}\left(\mathrm{M}_1\vec{h}_i' + \mathrm{b}_1\right)\right) \odot \mathrm{P}\left(\mathrm{M}_2\vec{h}_i' + \mathrm{b}_2\right) \tag{3}$$

$\odot$ denotes the element-wise product, and $\sigma$ is the activation function. The matrix $M_1$ and bias vector $b_1$ are trainable network parameters. $V$ represents the number of nodes, and P refers to the multilayer perceptron.

*3.3.2 SENet*

The graph embedding features obtained through GAT are converted into an RGB (Red, Green, Blue) three-channel image. Specifically, the obtained graph embeddings are mapped from the range [−1, 1] to [0, 255] to serve as two channels of the image feature, while the rule-based features are assigned as the third channel, resulting in a three-dimensional image feature. This image is then used as input to SENet [25]. For an input feature map X with dimensions H × W × C, where H and W represent the height and width of the feature map, and C represents the number of channels.

SENet first compresses the spatial dimensions of each feature map into a feature vector $Z$ with dimensions C × 1 × 1 using global average pooling. This is represented by the formula:

$$Z = GlobalAvgPool\,(X) \tag{4}$$

Then, SENet introduces an excitation module that learns the importance weights for each channel. Let Wexcite be the parameters of the excitation module. The excitation s for each channel can be calculated using the formula:

$$S = \sigma\left(W_{excite} \cdot f_{ReLU}\left(W_{squeeze} \cdot z\right)\right) \tag{5}$$

where $W_{squeeze}$ and $W_{excite}$ are the weight parameters in the compression and excitation modules, $f_{ReLU}$ denotes the ReLU activation function, and $\sigma$ denotes the sigmoid function.

Recalibration: The excitation vector $S$ is multiplied with the original feature vector X to perform a weighted recalibration of the feature responses for each channel. Through this process, SENet dynamically weights each channel, allowing the network to focus more on features that are important for the current task, thereby improving the model's performance and generalization capability.

## 4 Experiment

In this section, we evaluate the proposed method on the Ethereum Smart Contract dataset (ESC). We aim to address the following research questions:

RQ1: How does the proposed method compare to traditional tools in detecting reentrancy vulnerabilities? Can it effectively avoid some of the reasons for false negatives and false positives in traditional detection tools?

RQ2: Can the proposed method effectively detect reentrancy vulnerabilities? How does its accuracy, precision, recall, and F1 score compare to state-of-the-art graph neural network-based vulnerability detection methods?

RQ3: How does the proposed reentrancy vulnerability pattern matching method, SENet, and GAT affect the performance of the proposed approach?

### *4.1 Experimental Setup*

Experimental Environment: All experiments were conducted in the following hardware and software environment Operating System: Windows 10, Memory: 16 GB RAM, CPU: Intel i7-7700HQ, GPU: NVIDIA GTX 1050Ti 4 G, Python 3, TensorFlow 2.

Parameter Settings: Based on the experience of previous researchers [23,26,27], we set the parameters as follows: the Adam optimizer was used for the network with a learning rate of 0.001, Dropout set to 0.5, and a batch size of 32. In the dataset, 20% was selected as the test set and 80% as the training set. The average test results were taken as the final results.

Dataset: We use the ESC (Ethereum smart contract.) dataset [26] for experimentation, which consists of 307,396 smart contract functions from 40,932 smart contracts collected from real-world Ethereum. Among these functions, approximately 5013 are involved in calls that carry the risk of reentrancy vulnerability attacks.

Evaluation Metrics: Accuracy, recall, precision, and F1 score were used as evaluation metrics to measure the model's vulnerability detection performance. Accuracy is the ratio of the number of correctly predicted data points to the total number of data points. Recall represents the percentage of positive prediction samples among all samples with correct predictions. Precision is the percentage of positive prediction samples among those predicted as positive. The F1 score is used to measure the balance between precision and recall.

TP (True Positive) represents the number of correctly predicted positive samples, TN (True Negative) represents the number of correctly predicted negative samples, FP (False Positive) represents the number of incorrectly predicted positive samples, and FN (False Negative) represents the number of incorrectly predicted negative samples.

### *4.2 Comparison with Traditional Methods (RQ1)*

This paper first compares the proposed method, GMS, with existing smart contract vulnerability detection tools, including Smartcheck [28], Oyente [29], Mythril [30], Securify [31], and Slither [32]. As shown in Table 2 and Fig. 5, it is evident that traditional detection tools have not achieved good results in detecting reentrancy vulnerabilities. Even Slither, which is widely used in the industry, has only a 77.12% accuracy rate. As noted in Zheng et al.'s research [9], these tools do not handle issues such as reentrancy locks, caller access control, function access control, or economic risks after a function is reentered very well. In contrast, the proposed method, based on deep learning and incorporating custom pattern matching rules, effectively avoids some of the issues with false negatives and false positives encountered by traditional tools in detecting reentrancy vulnerabilities.

**Table 2:** Compare with traditional methods

| Methods | Acc (%) | Recall (%) | Precision (%) | F1 (%) |
|---|---|---|---|---|
| Smartcheck | 52.97 | 32.08 | 25.00 | 28.10 |
| Oyente | 61.62 | 54.71 | 38.16 | 44.96 |
| Mythril | 60.54 | 71.69 | 39.58 | 51.02 |
| Securify | 71.89 | 56.60 | 50.85 | 53.57 |
| Slither | 77.12 | 74.28 | 68.42 | 71.23 |
| **GMS** | **94.07** | **91.09** | **90.57** | **91.13** |

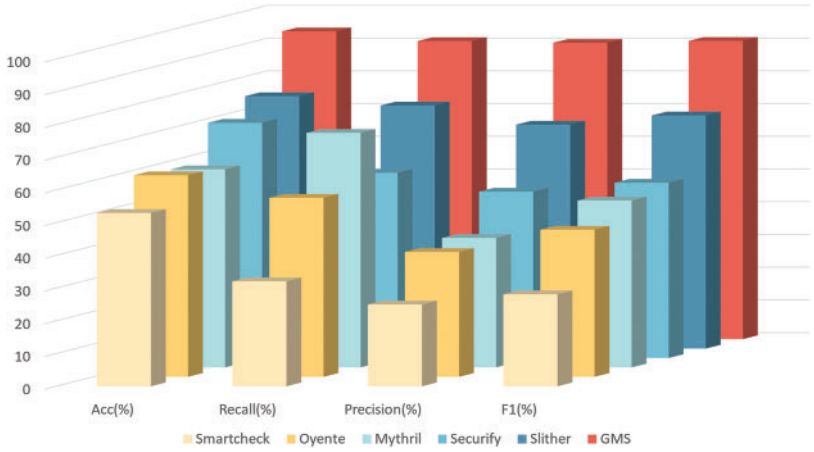


**Figure 5:** Compare with traditional methods

It is worth noting that the method proposed in this paper incurs the least cost in terms of vulnerability detection time. As shown in Fig. 6, Mythril has the longest average detection time, while Oyente and Smartcheck have similar detection times. Securify's average vulnerability detection time is 1.6 s, Slither's is 0.6 s, and the method proposed in this paper achieves the shortest execution time, requiring an average of only 0.1 s to detect vulnerabilities. Comparative experiments demonstrate that the proposed deep learning and pattern matching-based smart contract vulnerability detection method significantly improves detection efficiency compared to traditional methods.

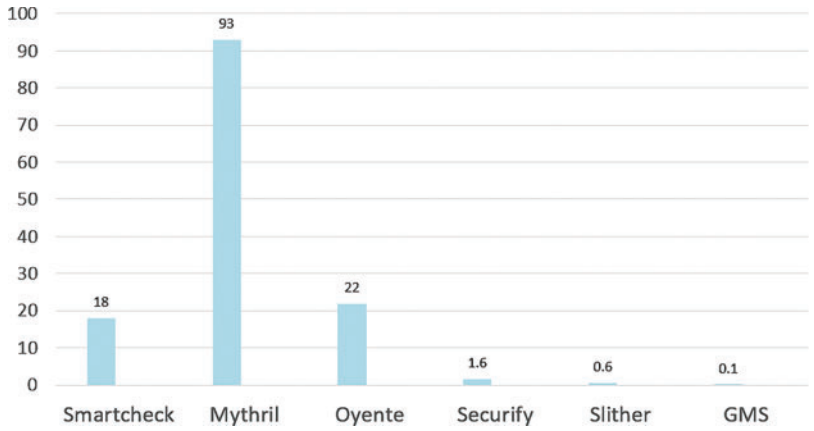


**Figure 6:** Average time spent on vulnerability detection

### *4.3 Comparison with Graph Neural Network Methods (RQ2)*

In this section, we benchmark the proposed method against existing graph neural network-based vulnerability detection methods, including:

DR-GCN [23]: A degree-reduced graph convolutional network that enhances node connectivity and removes the diagonal node degree matrix.

TMP [23]: A temporal message propagation network that learns contract graph features by continuously passing information along edges according to their temporal order.

CGE [26]: A smart contract vulnerability detection method that combines temporal message propagation networks with expert knowledge.

AME [27]: A method that combines graph neural networks with expert knowledge in an interpretable manner using self-attention mechanisms.

This paper compares the GMS method with four existing graph neural network-based approaches. The performance of different methods is shown in Table 3, including metrics such as accuracy, recall, precision, and F1 score. It can be observed that GMS achieves an accuracy 3.88% higher than AME, which is the best-performing method among the four. Experimental results indicate that graph neural network-based methods generally outperform traditional approaches, highlighting the significant potential of modeling source code as graphs and applying graph neural networks. Further analysis shows that the superior performance of GMS compared to other graph neural network-based methods is primarily due to its comprehensive consideration of various patterns that could lead to reentrancy vulnerabilities.

**Table 3:** Comparison with graph neural network based methods

| Methods | Acc (%) | Recall (%) | Precision (%) | F1 (%) |
|---|---|---|---|---|
| DR-GCN | 81.47 | 80.89 | 72.36 | 76.39 |
| TMP | 84.48 | 82.63 | 74.06 | 78.11 |
| CGE | 89.15 | 87.62 | 85.24 | 86.41 |
| AME | 90.19 | 89.69 | 86.25 | 87.94 |
| **GMS** | **94.07** | **91.09** | **90.57** | **91.13** |

### *4.4 Ablation Study (RQ3)*

The proposed method in this paper extracts features of smart contract reentrancy vulnerabilities using pattern matching, applies graph attention networks (GAT) to extract contract graph features, and utilizes SENet to combine these features. A shown in Table 4, the contribution of each component is evaluated by removing or replacing corresponding modules from the GMS network.

**Table 4:** Ablation study

| Methods | Acc (%) | Recall (%) | Precision (%) | F1 (%) |
|---|---|---|---|---|
| **GMS** | **94.07** | **91.09** | **90.57** | **91.13** |
| GMS (CNN) | 93.52 | 89.80 | 89.49 | 89.64 |
| GCN-PM-SENet | 83.76 | 80.31 | 80.94 | 80.62 |
| PM-SENet | 72.40 | 67.80 | 55.41 | 60.98 |

Graph Attention Networks (GAT): To demonstrate the effectiveness of GAT in extracting contract graph features, GAT in the GMS method was replaced with GCN. The accuracy of GCN-PM-SENet was 83.76%, which is 10.31% lower than the accuracy of GMS.

Pattern Matching Module: To assess the effectiveness of the pattern matching module, graph features were removed, and only pattern matching features were used. The accuracy of PM-SENet was 72.40%. Although this is significantly lower than the performance of graph neural network-based methods, it is still 0.51% higher than the accuracy of Securify.

SENet Module: To verify the ability of SENet to integrate graph features with pattern matching-based reentrancy vulnerability features, SENet in the GMS network was replaced with a CNN (Convolutional Neural Network). The accuracy of GMS(CNN) was 93.52%, which is 0.55% lower than the accuracy of GMS.

## 5 Conclusion

In this paper, we propose a novel reentrancy vulnerability detection method for smart contracts, combining deep learning with pattern matching. Compared to traditional methods, our approach effectively mitigates false positives and misidentifications caused by factors such as reentrancy locks, caller and function access control, and the evaluation of economic risks posed by reentered functions. Extensive experiments show that our method achieves a detection accuracy of 94.07%, with accuracy, recall, precision, and F1 score surpassing traditional detection methods and many graph neural network-based approaches. We believe our work marks an important step toward uncovering the potential of deep learning in the task of smart contract vulnerability detection.

**Acknowledgement:** The authors are grateful to the anonymous reviewers and the editor for their valuable comments and suggestions.

**Funding Statement:** This work was supported by the Higher Education Research Project of Jilin Province: JGJX24C118, and the National Defense Basic Scientific Research Program of China (No. JCKY2023602C026).

**Author Contributions:** The authors confirm contribution to the paper as follows: study conception and desgin: Dawei Xu, Fan Huang, Jiaxin Zhang, Yunfang Liang; analysis and interpretation of result: Dawei Xu, Fan Huang; draft manuscript preparation: Fan Huang, Jiaxin Zhang, Yunfang Liang; supervision, writing—review & editing: Baokun Zheng, Jian Zhao. 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 openly available in (ESC datasets) at (https://drive.google.com/file/d/1yFJSCiUuoiSx4uWYNcCESUvsEs5DOGM9/view) (accessed on 26 December 2024).

**Ethics Approval:** Not applicable.

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

## References

1. Szabo N. Formalizing and securing relationships on public networks. First Monday. 1997;2(9):1–21. doi:10.5210/fm.v2i9.548.
2. Zhang C, Shen T, Bai F. Toward secure data sharing for the IoT devices with limited resources: a smart contract-based quality-driven incentive mechanism. IEEE Internet Things J. 2023;10(14):12012–24. doi:10.1109/JIOT.2022.3142786.
3. Jamil F, Ibrahim M, Ullah I, Kim S, Kahng HK, Kim DH. Optimal smart contract for autonomous greenhouse environment based on IoT blockchain network in agriculture. Comput Electron Agric. 2022;192(5):106573. doi:10.1016/j.compag.2021.106573.
4. Feng Q, He D, Zeadally S, Khan MK, Kumar N. A survey on privacy protection in blockchain system. J Netw Comput Appl. 2019;126(2):45–58. doi:10.1016/j.jnca.2018.10.020.
5. Zhang C, Zhao M, Liang J, Fan Q, Zhu L, Guo S. NANO: cryptographic enforcement of readability and editability governance in blockchain databases. IEEE Trans Dependable Secure Comput. 2024;21(4):3439–52. doi:10.1109/TDSC.2023.3330171.
6. Defillama. Total Value Hacked. [cited 2024 Dec 26]. Available from: https://defillama.com/hacks.

7. Mehar MI, Shier CL, Giambattista A, Gong E, Fletcher G, Sanayhie R, et al. Understanding a revolutionary and flawed grand experiment in blockchain. J Cases Inf Technol. 2019;21(1):19–32. doi:10.4018/JCIT.
8. Dhillon V, Metcalf D, Hooper M. Blockchain enabled applications: understand the blockchain ecosystem and how to make it work for you. Berkeley, CA, USA: Apress; 2021. doi:10.1007/978-1-4842-6534-5
9. Zheng Z, Zhang N, Su J, Zhong Z, Ye M, Chen J. Turn the rudder: a beacon of reentrancy detection for smart contracts on ethereum. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE); 2023; Melbourne, VIC, Australia. p. 295–306. doi:10.1109/ICSE48619.2023.00036
10. Zhang C, Hu C, Wu T, Zhu L, Liu X. Achieving efficient and privacy-preserving neural network training and prediction in cloud environments. IEEE Trans Depend Secure Comput. 2023;20(5):4245–57. doi:10.1109/TDSC.2022.3208706.
11. Hu C, Zhang C, Lei D, Wu T, Liu X, Zhu L. Achieving privacy-preserving and verifiable support vector machine training in the cloud. IEEE Trans Inf Forensics Secur. 2023;18:3476–91. doi:10.1109/TIFS.2023.3283104.
12. Huang Q. Ethereum: introduction, expectation, and implementation. Highlights Sci Eng Technol. 2023;41:175–82. doi:10.54097/hset.v41i.6804.
13. Deng H, Liang J, Zhang C, Liu X, Zhu L, Guo S. FutureDID: a fully decentralized identity system with multi-party verification. IEEE Trans Comput. 2024;73(8):2051–65. doi:10.1109/TC.2024.3398509.
14. Zhang C, Zhao M, Zhang W, Fan Q, Ni J, Zhu L. Privacy-preserving identity-based data rights governance for blockchain-empowered human-centric metaverse communications. IEEE J Sel Areas Commun. 2024;42(4):963–77. doi:10.1109/JSAC.2023.3345392.
15. Ju H, Boutaba R, Kim M, Stiller B. Editorial for special issue on challenges and opportunities of Blockchain and Cryptocurrency. Int J Netw Manag. 2020;30(5):e2133. doi:10.1002/nem.2133.
16. Dameron M. Beigepaper: an ethereum technical specification. In: Ethereum project beige paper; 2018 Feb.
17. Hu X, Burgstaller B, Scholz B. EVMTracer: dynamic analysis of the parallelization and redundancy potential in the ethereum virtual machine. IEEE Access. 2023;11:47159–78. doi:10.1109/ACCESS.2023.3267277.
18. Bottoni S, Datta A, Franzoni F, Ragnoli E, Ripamonti R, Rondanini C, et al. 1DLT: rapid deployment of secure and efficient EVM-based blockchains. In: 4th International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2022); 2022 Dec 12–13; Paris, France. doi:10.4230/OASIcs.Tokenomics.2022.3
19. Szabo N. Smart contracts: building blocks for digital markets; 1996 [cited 2024 Dec 26]. Available from: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.
20. Pinna A, Ibba S, Baralla G, Tonelli R, Marchesi M. A massive analysis of ethereum smart contracts empirical study and code metrics. IEEE Access. 2019;7:78194–213. doi:10.1109/ACCESS.2019.2921936.
21. Christidis K, Devetsikiotis M. Blockchains and smart contracts for the Internet of Things. IEEE Access. 2016;4:2292–303. doi:10.1109/ACCESS.2016.2566339.
22. Kalodner H, Goldfeder S, Chen X, Weinberg SM, Felten EW. Arbitrum: scalable, private smart contracts. In: Proceedings of the 27th USENIX Security Symposium; 2018 Aug 15–17; Baltimore, MD, USA. p. 1353–70.
23. Zhuang Y, Liu Z, Qian P, Liu Q, Wang X, He Q. Smart contract vulnerability detection using graph neural network. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence; 2020 Jul 11–17; Yokohama, Japan: International Joint Conferences on Artificial Intelligence Organization; 2020. p. 3283–90. doi:10.24963/ijcai.2020/454
24. Veličković P, Cucurull G, Casanova A, Romero A, Liò P, Bengio Y. Graph attention networks. arXiv.1710.10903. 2017.
25. Hu J, Shen L, Sun G. Squeeze-and-excitation networks. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition; 2018 Jun 18–23; Salt Lake City, UT, USA: IEEE; 2018. p. 7132–41. doi:10.1109/CVPR.2018.00745
26. Liu Z, Qian P, Wang X, Zhuang Y, Qiu L, Wang X. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Trans Knowl Data Eng. 2023;35(2):1296–310. doi:10.1109/TKDE.2021.3095196.

27. 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. In: International Joint Conference on Artificial Intelligence; 2022. p. 2751–9.
28. Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y. SmartCheck: static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain; 2018; Gothenburg Sweden: ACM. p. 9–16. doi:10.1145/3194113.3194115
29. Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: ICSE'18: 40th International Conference on Software Engineering; 2018; Gothenburg, Sweden; p. 254–69. doi:10.1145/2976749.2978309
30. Mueller B. A framework for bug hunting on the ethereum blockchain. arXiv.2009.02066. 2017.
31. Tsankov P, Dan A, Drachsler-Cohen D, Gervais A, Bünzli F, Vechev M. Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security; 2018; Toronto, ON, Canada: ACM. p. 67–82. doi:10.1145/3243734.3243780
32. Feist J, Grieco G, Groce A. Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB); 2019 May 27; Montreal, QC, Canada: IEEE; 2019. p. 8–15. doi:10.1109/wetseb.2019.00008