The Journal of Systems and Software 216 (2024) 112118

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

A vulnerability detection framework with enhanced graph feature learning[☆]

Jianxin Cheng[a], Yizhou Chen[a], Yongzhi Cao[a,b], Hanpin Wang[c,a,*]

[a] *Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China*
[b] *Zhongguancun Laboratory, Beijing, China*
[c] *School of Computer Science and Cyber Engineering, Guangzhou University, Guangzhou, China*

ARTICLE INFO

Keywords:
Vulnerability detection
Software security
Code representation
Control flow graph
Graph neural network

ABSTRACT

Vulnerability detection in smart contracts is critical to secure blockchain systems. Existing methods represent the bytecode as a graph structure and leverage graph neural networks to learn graph features for vulnerability detection. However, these methods are limited to handling the long-range dependencies between nodes. This means that they might focus on learning local node feature while ignoring global node information. In this paper, we propose a novel vulnerability detection framework with **E**nhanced **G**raph **F**eature **L**earning (EGFL), which aims to extract the global node information and utilize it to improve vulnerability detection in smart contracts. Specifically, we first represent the bytecode as a Control Flow Graph (CFG). To extract global node information, EGFL constructs a linear node feature matrix from CFG, and uses the feature-aware and relationship-aware modules to handle long-range dependencies between nodes. Meanwhile, a graph neural network is adopted to extract the local node feature from CFG. Subsequently, we fuse the global node information and local node feature to generate an enhanced graph feature for capturing more vulnerability features. We evaluate EGFL on the benchmark dataset with six types of smart contract vulnerabilities. Results show that EGFL outperforms fourteen state-of-the-art vulnerability detection methods by 10.83%–60.28% in F1 score.

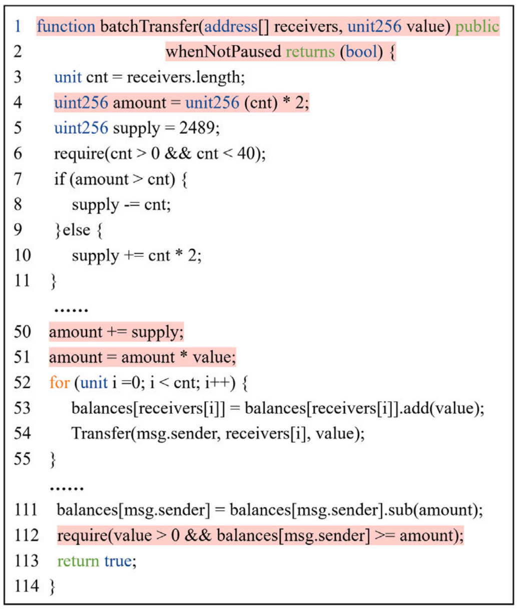
1. Introduction

Detecting software vulnerabilities is essential to ensure the security of software (Li et al., 2021; Tang et al., 2023; Zhang et al., 2023). This work pays attention to vulnerability detection in smart contracts. Smart contracts are program codes that run on blockchain systems and obtain growing attention due to the rapid development of blockchain Vacca et al. (2021), Viglianisi et al. (2020). Specifically, smart contracts are primarily written in Solidity programming language and enable various software applications (Cai et al., 2023; Yuan et al., 2023), such as decentralized finance, supply chain, and online banking. Like other program codes, smart contracts might contain various types of vulnerabilities, which lead to huge financial losses (Feist et al., 2019; Luu et al., 2016). A typical example is that the attackers leveraged a reentrancy vulnerability to launch the DAO attack and stole over 60 million USD (Luu et al., 2016). The security concerns raised by these vulnerabilities can seriously hinder the development of blockchain systems, which makes it essential to detect potential vulnerabilities in smart contracts (Chen et al., 2023; Liao et al., 2022).

Existing efforts and limitations. As countermeasures, many efforts have been developed to identify vulnerabilities based on the smart contract bytecode. Among them, static analysis (Tikhomirov et al., 2018; Torres et al., 2018) and dynamic analysis (Breidenbach, 2017; Nguyen et al., 2020) tools, primarily use predefined patterns or specifications to analyze smart contract vulnerabilities, which may overlook many potential vulnerabilities and designing these patterns is labor-intensive (Chen et al., 2023; Liu et al., 2023a). Recently, the deep learning-based vulnerability detection methods (Qian et al., 2023; Sendner et al., 2023; Tann et al., 2018; Zeng et al., 2022; Zhang et al., 2022a) have gained considerable attention, given that they have achieved impressive performance by automatically learning the vulnerability features. For example, some researchers (Sendner et al., 2023; Tann et al., 2018; Zhang et al., 2022a) convert the bytecode into the sequence structure, i.e., the opcode sequence, and then introduce the techniques of natural language processing to learn the vulnerability features hidden in the bytecode. Unfortunately, these sequence-based methods ignore the rich program semantics of the bytecode (Huang et al., 2021).

☆ Editor: Dario Di Nucci.
* Corresponding author at: Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China.
E-mail addresses: jianxin@stu.pku.edu.cn (J. Cheng), yizhouchen@stu.pku.edu.cn (Y. Chen), caoyz@pku.edu.cn (Y. Cao), whpxhy@pku.edu.cn (H. Wang).

https://doi.org/10.1016/j.jss.2024.112118
Received 14 December 2023; Received in revised form 3 April 2024; Accepted 31 May 2024
Available online 1 June 2024



```
1   function batchTransfer(address[] receivers, unit256 value) public
2                  whenNotPaused returns (bool) {
3     unit cnt = receivers.length;
4     uint256 amount = unit256 (cnt) * 2;
5     uint256 supply = 2489;
6     require(cnt > 0 && cnt < 40);
7     if (amount > cnt) {
8       supply -= cnt;
9     }else {
10      supply += cnt * 2;
11    }
      ......
50    amount += supply;
51    amount = amount * value;
52    for (unit i =0; i < cnt; i++) {
53      balances[receivers[i]] = balances[receivers[i]].add(value);
54      Transfer(msg.sender, receivers[i], value);
55    }
      ......
111   balances[msg.sender] = balances[msg.sender].sub(amount);
112   require(value > 0 && balances[msg.sender] >= amount);
113   return true;
114 }
```

Fig. 1. A motivating example of the integer overflow vulnerability. The code statements marked pink are vulnerable. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

To this end, existing methods (Qian et al., 2023; Zeng et al., 2022) represent the bytecode as various graph structures, such as Control Flow Graph (CFG), where the nodes and edges denote the bytecode statements and program dependencies among statements, respectively. Subsequently, these graph-based methods use various graph neural networks (Kipf and Welling, 2017; Veličković et al., 2018) to learn the graph feature from all connected node features for vulnerability detection. However, some studies (Hellendoorn et al., 2019; Wen et al., 2023) show that the graph neural networks update the node features through neighborhood aggregation, which makes them limited to dealing with long-range dependencies between nodes that are not directly connected. This suggests that existing graph-based methods might focus on learning the local node feature while failing to capture the global node information (i.e., long-range dependencies between nodes). Thus, the performance of existing graph-based methods in smart contract vulnerability detection may be limited, since they ignore the global node information of the bytecode.

Our solution. To alleviate this limitation, we present a novel vulnerability detection framework for smart contracts with **E**nhanced **G**raph **F**eature **L**earning (EGFL). The objective of EGFL is to learn the global node information and combine it with the local node feature for improving vulnerability detection in smart contracts.

More specifically, in the first phase, we transform the bytecode as a CFG and vectorize all nodes into corresponding feature vectors. Besides, we traverse all node vectors in a breadth-first search way to generate the linear node feature matrix for establishing long-range dependencies between the CFG nodes. In the second phase, based on the node feature matrix, we leverage the feature-aware and relationship-aware modules to extract the global node information. Among them, the feature-aware module utilizes convolutional operations to capture critical nodes that may be vulnerable. The relationship-aware module establishes and handles long-range dependencies between critical nodes using the multi-head attention mechanism. Meanwhile, a graph neural network is applied to extract the local node feature based on the CFG input. Finally, in the third phase, we use a self-attention mechanism to fuse the global node information and local node feature and generate an enhanced graph feature. It is then fed into a classifier to determine whether the smart contract is vulnerable or not.

Evaluation. To evaluate EGFL, we perform extensive experiments on a widely studied dataset which includes six types of vulnerabilities in smart contracts. Fourteen state-of-the-art vulnerability detection efforts, including six static/dynamic analysis tools and eight deep learning-based detection methods, are chosen as the baseline methods. Results indicate that EGFL outperforms all vulnerability detection methods, with an improvement of 9.44%–35.19%, 8.67%–60.13%, 10.49%–57.98%, and 10.83%–60.28% in the accuracy, recall, precision, and F1 score, respectively. The ablation study analyzes the reasons why EGFL is effective in identifying smart contract vulnerabilities.

In summary, the main contributions of this work are presented below:

- We propose EGFL, a novel vulnerability detection framework for smart contracts. EGFL can extract the global node information, and combine it with the local node feature to generate an enhanced graph feature of bytecode for capturing more vulnerability features.
- Experimental results demonstrate the effectiveness of EGFL in identifying vulnerabilities in smart contracts. To facilitate future research, we release the source code at https://github.com/Jamesken23/EGFL.

The remainder of this paper is organized as follows. Section 2 gives the motivating example and preliminary. In Section 3, we introduce the proposed EGFL framework in detail. Sections 4 and 5 provide the experimental setup and experimental results, respectively. Sections 6 and 7 present the additional discussion of our approach and some related studies. Finally, we summarize our work and offer some perspectives in Section 8.

2. Motivating example and preliminary

In this section, a vulnerable smart contract example is first given to explain our motivation. Then, we provide some preliminaries about smart contracts.

2.1. Motivating example

We give a typical example of the integer overflow vulnerability, as shown in Fig. 1. The "batchTransfer" function allows users to deposit some tokens into the "amount" variable (line 4). However, no overflow judgment is performed for the "amount" variable. The integer overflow vulnerability will occur when the "amount" variable reaches its maximum integer value (i.e., 2^{256}-1). To be more specific, the attackers can bypass the code responsible for verifying the "balance" (line 112) when the "amount" variable is manipulated to overflow (lines 1, 50, and 51). Thus, the attacker can transfer plenty of tokens at a minimal expense by exploiting this vulnerability. That is, we can accurately identify this vulnerability by capturing the relationship between these critical code statements. In Fig. 2, this motivating example is compiled into the CFG where each node corresponds to a line of code statement. There are five critical nodes relevant to the vulnerability in pink, and the long-range dependency can be seen as the relationship between these critical nodes that are distant from each other in the graph (Wen et al., 2023). Based on the above analysis, only by building the long-range dependencies between these critical nodes and extracting global node information, we can detect the integer overflow vulnerability in this example.

Existing graph-based methods (Qian et al., 2023; Zeng et al., 2022) primarily use graph neural networks, such as Graph Convolutional Network (GCN) (Kipf and Welling, 2017) and Graph Attention Network (GAT) (Veličković et al., 2018), to learn the graph features for vulnerability detection. These graph neural networks are hard to learn long-range dependencies among the nodes of CFG, such as the critical nodes in Fig. 2, due to the limitation of the one-hop neighborhood

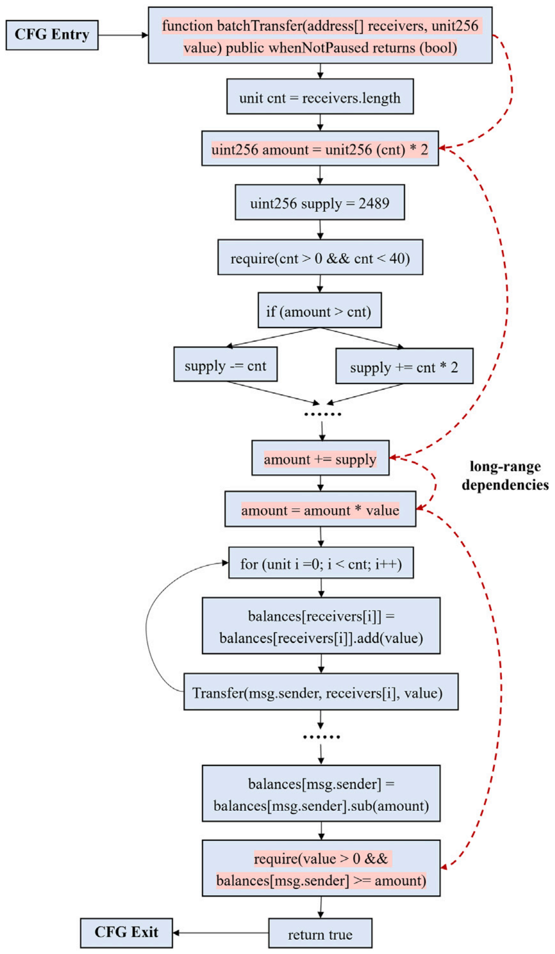

CFG Entry
function batchTransfer(address[] receivers, unit256 value) public whenNotPaused returns (bool)
unit cnt = receivers.length
uint256 amount = unit256 (cnt) * 2
uint256 supply = 2489
require(cnt > 0 && cnt < 40)
if (amount > cnt)
supply -= cnt
supply += cnt * 2
amount += supply
amount = amount * value
long-range dependencies
for (unit i =0; i < cnt; i++)
balances[receivers[i]] = balances[receivers[i]].add(value)
Transfer(msg.sender, receivers[i], value)
balances[msg.sender] = balances[msg.sender].sub(amount)
require(value > 0 && balances[msg.sender] >= amount)
return true
CFG Exit

Fig. 2. The CFG of motivating example. Red dashed lines mean the long-range dependencies between the nodes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

aggregation (Hellendoorn et al., 2019). This means that existing methods ignore the global node information, which might hinder the graph feature learning and further affect their detection performance. To verify this, we analyze the relationship between the model detection performance and the node number of the bytecode CFG, as presented in Section 6.1. Results indicate that the SMS (Qian et al., 2023), a state-of-the-art detection method, does not obtain satisfactory results in the CFG with more nodes, especially exceeding 100 nodes. This proves the limited effectiveness of existing graph-based methods in capturing and utilizing global node information.

To facilitate a clearer understanding of the long-range dependencies between graph nodes, we give the motivating example written in easily readable Solidity source code, rather than bytecode. It is worth noting that the long-range dependencies in the CFG of source code may not exist in the same form in the CFG of bytecode. This discrepancy arises from the aggressive optimizations performed during the compilation process from Solidity source code to bytecode (Cai et al., 2023; He et al., 2023). However, as discussed in Section 6.1, it is still possible for long-range dependencies to be present in the CFG of bytecode, which can ultimately contribute to enhancing the performance of vulnerability detection.

2.2. Smart contract bytecode and vulnerabilities

Smart contracts are program code that executes automatically on the blockchain system. Ethereum, as the most popular blockchain system, has hosted over 46.78 million smart contracts (Viglianisi et al., 2020; Yang et al., 2021). These smart contracts are primarily written in Solidity and then compiled into the bytecode that runs on the Ethereum virtual machine (He et al., 2023; Huang et al., 2021). The bytecode consists of a series of operation instructions that can reflect the program logic embedded within the smart contracts. Following previous studies (Chen et al., 2023; Qian et al., 2023), we focus on six types of prevalent smart contract vulnerabilities below.

Reentrancy is one of the most damaging vulnerabilities that incurred the notorious DAO attack on Ethereum. It arises when contracts invoke one another, sharing similarities with the recursive call of a function. Exploiting a developer's oversight, an attacker manipulates a program to repeatedly execute their maliciously crafted code within a transaction until the available gas is exhausted. This malicious action can lead to substantial financial losses.

Timestamp dependence vulnerability occurs when a function relies on the block timestamp to execute the critical operations. For instance, this function may take the $block.timestamp$ of a future block to produce random numbers and determine the winner of a game. However, the miners who mine the block can manipulate the block's timestamp within a certain time range. As a result, malicious miners can exploit this situation by manipulating the block timestamp to gain unauthorized advantages.

Integer overflow/underflow vulnerability occurs when an integer variable undergoes a computation operation without proper consideration of its bounds. This oversight results in a value that surpasses the upper or lower limit of the variable type, deviating from the expected value as intended by the developer. If the developer neglects to verify the final value of the variable before subsequent operations, it can lead to financial losses.

Delegatecall vulnerability allows the caller to run the callee's contract code in their contract execution environment. Running the callee's function in the caller's environment may generate unexpected results if their execution environments are not the same.

Block number dependency vulnerability occurs when the block parameters are exploited by attackers to generate a specified random number. Since smart contracts cannot call built-in functions to generate a random number.

Unchecked external call vulnerability appears when the called contract has a runtime error and these functions do not stop execution. Also, this vulnerability is called mishandled exceptions (Tikhomirov et al., 2018).

3. Approach

In this section, we introduce EGFL, a novel vulnerability detection framework for smart contracts. First, Fig. 3 gives a brief overview of EGFL. Then, we describe the implementation details of EGFL, separately.

3.1. Overview

Compared to existing graph-based vulnerability detection methods, EGFL can effectively extract the global node information in the graph structure of bytecode. Subsequently, EGFL combines it with the local node feature to generate the enhanced graph feature. It contains comprehensive structural information in the smart contract bytecode, which helps to capture more valuable vulnerability features. By this, EGFL can improve smart contract vulnerability detection via the enhanced graph feature learning strategy.

As shown in Fig. 3, EGFL mainly comprises the following three phases:

- **Data processing.** Given the bytecode input X, we convert it into a CFG graph G and vectorize all nodes in latent feature

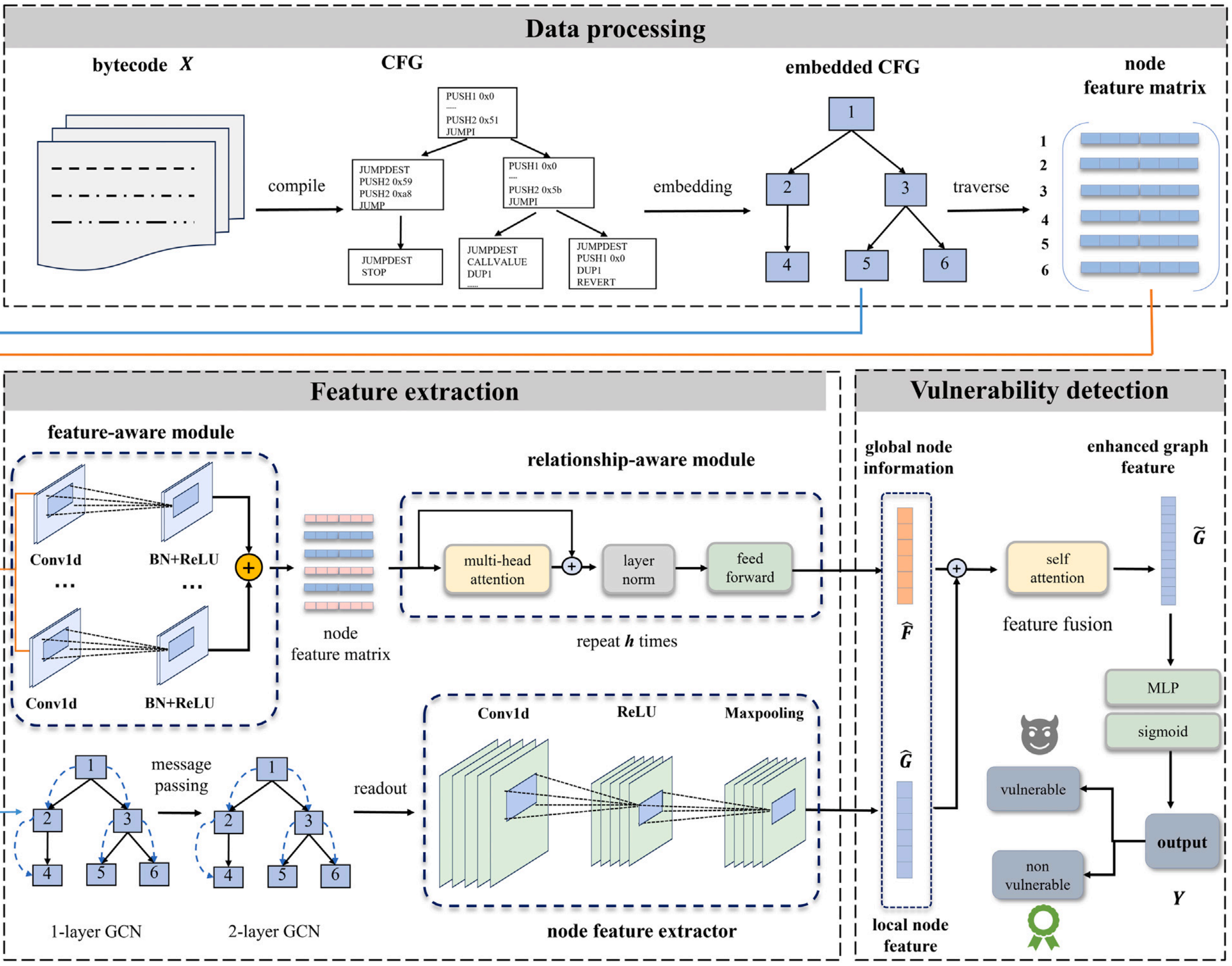


Fig. 3. The overview of our EGFL framework. Briefly, EGFL consists of three phases: data processing, feature extraction, and vulnerability detection.

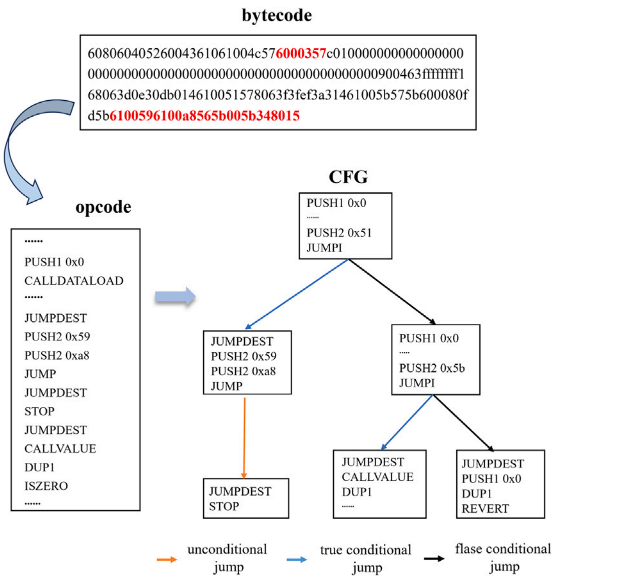
space. Meanwhile, we traverse all graph nodes in a breadth-first search way and then generate the node feature matrix F with linear structure, which helps to capture long-range dependencies between the nodes.

- **Feature extraction.** Based on the feature matrix F, we cooperate with the feature-aware and relationship-aware modules to handle long-range dependencies between critical nodes and then extract the global node information $\hat{F}$. Additionally, given the graph G input, the GCN model with a node feature extractor is applied to extract the local node feature $\hat{G}$.
- **Vulnerability detection.** The self-attention mechanism is used to fuse these two types of extracted features ($\hat{G}$ and $\hat{F}$) and generate an enhanced graph feature $\tilde{G}$. Subsequently, we feed it into the Multi-Layer Perceptron (MLP) and sigmoid function to approach the final prediction $\hat{Y}$.

3.2. Data processing

Previous studies (Pasqua et al., 2023; Zeng et al., 2022; Zhang et al., 2022a) have shown that the CFG can embody rich program structural information in smart contract code, such as semantic and syntactic structural information. Following these studies, we represent the bytecode as the CFG. Moreover, we build a linear node feature matrix from this CFG for capturing long-range dependencies between the nodes. The construction of the CFG and linear node feature matrix is introduced as follows.

Construction of the CFG. The construction process of the CFG is clearly illustrated in Fig. 4. Given the bytecode input X, we first


bytecode

```
608060405260043610610004c576000357c0100000000000000000
0000000000000000000000000000000000000000900463ffffffff1
68063d0e30db014610051578063f3fef3a31461005b575b600080f
d5b6100596100a8565b005b348015
```

opcode

```
......
PUSH1 0x0
CALLDATALOAD
......
JUMPDEST
PUSH2 0x59
PUSH2 0xa8
JUMP
JUMPDEST
STOP
JUMPDEST
CALLVALUE
DUP1
ISZERO
......
```

Fig. 4. An example to illustrate how to convert the smart contract bytecode into the CFG.

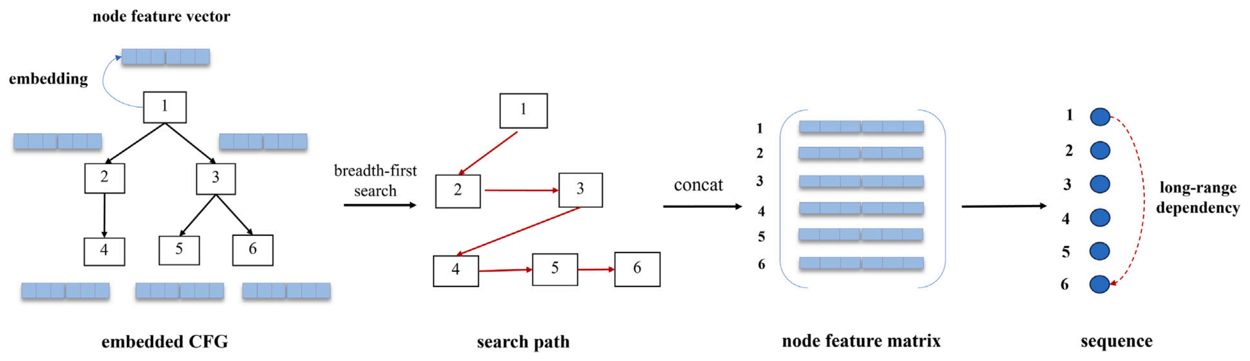


Fig. 5. The build process of the linear node feature matrix. In particular, each node in the feature matrix is treated as each token in the sequence structure.

Table 1
The partial bytecode value, operation instruction (i.e., opcode), and the corresponding definition.

Definitions	Operation instructions	Bytecode
Arithmetic operation	ADD MUL SUB DIV SDIV MOD SMOD ADDMOD MULMOD EXP SIGNEXTEND	0 × 01 - 0 × 0B
Block information	BLOCKHASH COINBASE TIMESTAMP NUMBER DIFFICULTY GASLIMIT	0 × 40 - 0 × 45
Comparison	LT GT SLT SGT EQ	0 × 10 - 0 × 14
Flow	JUMP JUMPI PC	0 × 56 - 0 × 58
Push	PUSH1 - PUSH32	0 × 60 - 0 × 7F
Duplication	DUP1 - DUP16	0 × 80 - 0 × 8F
Exchange	SWAP1 - SWAP16	0 × 90 - 0 × 9F
Logging	LOG0 - LOG4	0xA0 - 0xA4

transform it into the human-readable opcode based on the definition in the Ethereum yellow paper (He et al., 2023; Liu et al., 2019). The opcode consists of a sequence of numbers interpreted by the Ethereum virtual machine that are the operation types to be executed. These instructions can represent the logic of the program execution, and partial instructions and corresponding definitions are given in Table 1. Based on the opcode sequence, we adopt a public automated tool in Qian et al. (2023) to generate the CFG graph $G = (V, E)$, which includes two parts: the nodes V (i.e., basic blocks) and control flow edges E. More details are given below.

Firstly, each graph node $v \in V$ contains a set of instruction statements (i.e., code statements), which are all derived from the opcode, as shown in the left of Fig. 4. We regard the branch instructions, such as JUMP, JUMPI, and RETURN, as the sign of the end of a node, which means that this node needs to connect subsequent nodes. Secondly, the control flow edge $e \in E$ reveals the possible execution flows between the interconnected nodes by the *conditional* and *call* statements. Besides, EGFL primarily considers three types of control flow edges. As shown in the right of Fig. 4, the orange, blue, and black arrows are used to indicate the unconditional jump instructions (i.e., JUMP), true conditional jump instructions (i.e., JUMPI) and false conditional jump instructions (i.e., JUMPI), respectively.

Construction of the embedded CFG. After getting the graph G, each node is represented as a vector in latent feature space, which can be understood and processed by deep neural networks. Briefly, we adopt the word2vec embedding algorithm (Pennington et al., 2014) to handle all statements in the nodes, and transform each node v into a k-dimensional feature vector ($v \in \mathbb{R}^k$). Following this, we obtain the embedded graph G, which can be used to extract the local node feature of the CFG.

Construction of the node feature matrix. As described in Fig. 5, a node feature matrix is constructed from the embedded graph G. It is worth noting that it is easier to capture long-range dependencies from the linear sequence structure compared to the graph structure (Fu and Tantithamthavorn, 2022). Therefore, we convert the CFG into the node feature matrix with a linear structure. Among them, each node in the feature matrix is considered as each token in the sequence structure.

Assuming that the generated CFG contains n nodes, we first traverse all nodes in a breadth-first search way (Chen et al., 2023), which traverses the possible program execution paths from the entry node of the CFG. This search process is finished when all the nodes have been traversed. However, when the CFG contains loops, it becomes challenging to traverse all nodes using the breadth-first search. To address this, we maintain a collection of visited nodes during the search process (Mi et al., 2021). When encountering a node, we first check if it already exists in the collection of visited nodes. If the node is not present, we add it to our node search path and include it in the collection. Subsequently, we proceed to explore other nodes connected by neighboring edges from the current node. Conversely, if the node is already in the collection, we skip it and move to another path to avoid redundant traversals. By this, we can avoid infinite loop paths in the CFG.

After finishing this search process, we obtain the node search path, which can be considered as the execution path of the CFG. It allows a linear representation of the control flow dependencies between nodes. However, it is worth noting that the node search path may not cover all possible execution paths, due to the limitation of the visited node collection (Zhang et al., 2023). Nevertheless, this limitation has minimal impact on our EGFL. The reason is that our primary goal is to establish long-range dependencies between nodes, rather than capturing all execution paths, from the node search path. Therefore, as long as this node search path covers all nodes, it does not affect the extraction of global node information. Next, based on this search path, all nodes are combined to generate the node feature matrix $F \in \mathbb{R}^{n \times k}$, consisting of n node feature vectors. These operations make it reasonable and possible to capture long-range dependencies between nodes from the node feature matrix, compared to the original CFG.

3.3. Feature extraction

After obtaining the feature matrix F and the embedded graph G, we extract the global node information $\hat{F}$ and the local node feature $\hat{G}$ in turn.

Extracting the global node information. The multi-head attention mechanism has been widely applied to establish long-range dependencies between tokens in sequence structure (Vaswani et al., 2017; Yang et al., 2023). More critically, it can automatically determine the importance of these dependencies, and learn corresponding global node information from important dependencies. Inspired by this, we introduce the multi-head attention mechanism and design the relationship-aware module to deal with long-range dependencies between nodes (i.e., tokens). As depicted in Fig. 3, the feature-aware module and relationship-aware module are coupled to extract the global node information $\hat{F}$. The detailed process is illustrated below.

Firstly, based on the feature matrix F, we feed it into the feature-aware module, which contains k_1 one-dimensional convolutional kernels (Conv1d). The Batch Normalization (BN) layer and the ReLU activation function are set to follow each convolutional kernel. Indeed, the feature-aware module captures the crucial nodes that are associated with vulnerabilities by performing k_1 convolutional operations.

In each convolutional operation, the convolution kernel is applied to scan all the node feature vectors $f_i \in \mathbb{R}^k$ in the feature matrix F with the window size and stride of n and 1. The BN and ReLU layers are then utilized to further abstract the key feature point $\hat{f}_{i,j}$. The calculation process is listed according to:

$$\hat{f}_{i,j} = ReLu(BN((Conv_j(f_i)|j \in k_1))), \tag{1}$$

$$Conv_j(f_i) = w_j \cdot f_i + b_j, \tag{2}$$

where w_j and b_j refer to the learned weight and bias of the jth convolution kernel $Conv_j(\cdot)$.

Particularly, this feature point in each node vector records the information that is more associated with vulnerabilities. After performing k_1 convolutional operations, we generate n new node vector $\hat{f}_i \in \mathbb{R}^{k_1}$, and obtain the new feature matrix $\hat{F} \in \mathbb{R}^{n \times k_1}$. The feature-aware module takes advantage of convolutional neural networks to ignore some vulnerability-irrelevant nodes, which helps to accurately capture the critical nodes (Liu et al., 2023a; Wen et al., 2023). Briefly, this module uses k_1 convolutional kernels (i.e., convolutional operations) to scan each node in the node feature matrix, and perform the convolutional operation with all nodes. It then obtains the importance of all nodes by the activation function, which can ignore irrelevant nodes and highlight critical nodes relevant to vulnerabilities. We aim to reduce the interference of irrelevant code information on the relationship-aware module, allowing it to focus more on building long-range dependencies associated with vulnerabilities. More details of this module are discussed in Section 5.3.

Next, the new feature matrix $\hat{F}$ is fed into the relationship-aware module. In detail, the multi-head attention mechanism builds all long-range dependencies between critical nodes and then determines the importance of these dependencies. By this, the global node information is extracted from the important long-range dependencies. Notably, as shown in the right of Fig. 6, the objective of the above operation is to automatically add virtual long-range edges between the critical nodes in the CFG.

The relationship-aware module consists of h blocks, as presented in Fig. 3. Each block comprises a total of three parts, containing one multi-head attention $MHA(\cdot)$, one layer normalization $LN(\cdot)$, and one feed forward $FF(\cdot)$. Among them, the feed forward consists of two linear transformations and a nonlinear activation (Yang et al., 2023). The residual connection is added over the multi-head attention.

Formally, given the input $\hat{F}_{i-1}$, we output the feature vector $\hat{F}_i \in \mathbb{R}^{k_1}$ in the ith block of the relationship-aware module according to:

$$\tilde{F}_i = MHA(\hat{F}_{i-1}) + \hat{F}_{i-1}, \tag{3}$$

$$\hat{F}_i = FF(LN(\tilde{F}_i)). \tag{4}$$

Finally, the global node information $\hat{F} \in \mathbb{R}^{k_1}$ is generated by repeating h blocks in total.

Extracting the local node feature. Given the embedded graph G input, we use the GCN model with a node feature extractor to extract the local node feature $\hat{G}$, as shown in Fig. 3. Our GCN model primarily includes a message passing phase and a readout phase.

Specifically, in the message passing phase, our GCN takes l convolutional layers to update the feature vectors v^{l+1} of all n graph nodes by:

$$v^{l+1} = \sigma(\hat{D}^{-\frac{1}{2}} \hat{A} \hat{D}^{-\frac{1}{2}} v^l W^l), \tag{5}$$

where the matrix $\hat{A} = A + I$ combines the adjacency matrix A and the self-loop matrix I, $\sigma(\cdot)$ is the sigma activation function, v^l and W^l represent the feature vectors of all nodes and trainable weight matrix in the lth layer, respectively. Following (Kipf and Welling, 2017; Zhuang et al., 2020), the number (l) of the convolutional layers is set to 2. Besides, the diagonal node degree matrix $\hat{D}$ is adopted to normalize the matrix $\hat{A}$. For each graph node v_i, it employs the same convolutional layer to update its feature vector v_i^{l+1} by aggregating the vectors of all neighboring nodes.

Finally, in the readout phase, we read out the feature vectors of all nodes and generate the local node feature $\hat{G}$. Among them, the node feature extractor $NFE(\cdot)$ is used to handle all node vectors, which can highlight the critical elements in each node vector and avoid overfitting. The node feature extractor contains the one-dimensional convolutional layer, ReLU activation function, and maxpooling layer in total. After updating all node vectors v_i^{l+1}, we then sum up them and generate the local node feature $\hat{G}$ of the CFG according to:

$$\hat{G} = \sum_{i=1}^{n} NFE(v_i^{l+1}). \tag{6}$$

3.4. Vulnerability detection

After obtaining the local node feature $\hat{G}$ and the global node information $\hat{F}$, we employ the self-attention mechanism to fuse them and then generate the enhanced graph feature $\tilde{G}$. Indeed, these two kinds of features ($\hat{G}$ and $\hat{F}$) can complement each other in smart contract vulnerability detection, and thereby we can combine them to enhance the performance of vulnerability detection. The specific calculations are indicated in the following.

Given two input features $\hat{G}$ and $\hat{F}$, we first employ the concatenation operation $\oplus$ to concat them together, and then use the self-attention mechanism $SA(\cdot)$ to generate the final graph feature $\tilde{G}$ by:

$$\tilde{G} = SA(\hat{G} \oplus \hat{F}). \tag{7}$$

Since the local node feature and global node information are heterogeneous, the self-attention mechanism is used to balance all features by automatically computing their coefficients. Notably, another advantage of the attention mechanism is that the attention weights could emphasize the importance of different features.

Following this, the graph feature $\tilde{G}$ is fed into the MLP with a sigmoid activation function $\sigma(\cdot)$ to output the predicted probability $\hat{Y}$ according to:

$$\hat{Y} = \sigma(MLP(\tilde{G})). \tag{8}$$

The predicted probability $\hat{Y}$ is close to 1, which indicates that this smart contract X is likely to be vulnerable. In contrast, this input contract X is safe and non-vulnerable when the probability $\hat{Y}$ is near 0.

Finally, the cross-entropy loss function $L(\cdot)$ (Zhang et al., 2023) is applied to optimize our approach by:

$$L(\hat{Y}, Y) = -\sum_{i=1}^{S} Y_i \cdot \log(\hat{Y}_i), \tag{9}$$

where Y and S denote the truth label and the size of the training data, respectively. Our objective is to make the prediction $\hat{Y}$ as close as possible to the truth label Y by minimizing the cross-entropy loss between them.

4. Experimental setup

In this section, we detail our experimental setup, comprising the research questions, datasets, baselines, evaluation metrics, and implementation details.

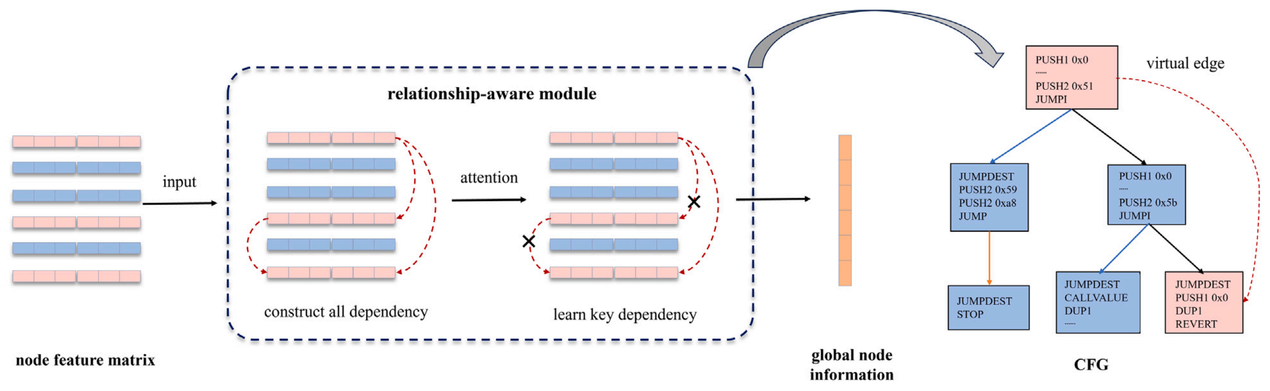


Fig. 6. The extraction process of global node information. The pink nodes indicate the critical nodes that might be vulnerable. The red dashed lines mean the long-range dependencies between the nodes. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.1. Research questions

To evaluate the EGFL from multiple perspectives, a large number of experiments are designed to answer the following Research Questions (RQ):

RQ1: How effective is EGFL in detecting vulnerabilities in smart contracts when compared to existing state-of-the-art methods?

This RQ aims to evaluate the performance of EGFL and existing state-of-the-art vulnerability detection methods on the same vulnerability dataset. For this purpose, fourteen representative efforts are selected as the baselines, including six static/dynamic analysis tools and eight deep learning-based detection methods. Next, we compare the performance of EGFL and these baselines on the benchmark vulnerability dataset, which contains six categories of vulnerabilities in smart contracts.

RQ2: What is the effect of different graph features on EGFL?

EGFL incorporates two types of graph features, i.e., the global node information and the local node feature, for vulnerability detection. In this RQ, our focus is to investigate the effect of these two kinds of features on EGFL. To analyze this, we first assess the individual contributions of all features to the performance of EGFL by removing each type of feature separately. Moreover, we examine the weights assigned to all two features within the EGFL when detecting different vulnerabilities.

RQ3: What is the effect of different feature extraction modules on EGFL?

EGFL uses the GCN model and node feature extractor to learn the local node feature. Meanwhile, the feature-aware and relationship-aware modules are combined to extract the global node information. This RQ aims to explore the impact of these deep learning-based feature extraction modules on the performance of EGFL.

4.2. Datasets

To answer the above research questions, a large number of experiments are performed on a widely-studied dataset (Qian et al., 2023), which comprises over 40K smart contracts written in the Solidity. These smart contracts are collected from the popular Ethereum platform (Durieux et al., 2020), and contain the following six types of common vulnerabilities: reentrancy (RE), timestamp dependence (TD), integer overflow/underflow (OF), delegatecall (DE), block number dependency (BN), and unchecked external call (UC). After obtaining the above vulnerability datasets, we manually confirm the labeling correctness for these smart contracts by rechecking the corresponding labeling strategy (Qian et al., 2023). Among them, there are 680 of 2385, 2242 of 4490, 1368 of 7183, 136 of 414, 263 of 1239, and 920 of 2004 vulnerable smart contracts in the RE, TD, OF, DE, BN, and UC vulnerability datasets, respectively. Although the data amount varies between each vulnerability dataset, this does not interfere with the training of the deep learning model (Chen et al., 2023; Liu et al., 2023a). To ensure the validity of the experimental results, we randomly allocate 80% of smart contract code to the training set, while the remaining 20% is reserved for the test set. To avoid the overfitting issue, we adopt the dropout operation (Tang et al., 2023) in model training. Briefly, this operation is added between the hidden layer of the GCN model and the relationship-aware module, and randomly discards the output of some neurons with a certain probability (i.e., 0.5). Besides, we monitor the performance of the training model through the early stopping operation (Yang et al., 2021). When the performance of the training model drops for five continuous epochs, the process of model training stops. By this, we can learn more robust vulnerability features.

Particularly, the above vulnerabilities have been widely applied in previous studies (Huang et al., 2021; Liao et al., 2022; Liu et al., 2023b) for smart contract vulnerability detection. As mentioned in Section 2.1, these vulnerabilities have typical vulnerability characteristics in Ethereum smart contracts, and have caused more than 70% of the economic losses on the Ethereum platform (Durieux et al., 2020; Feist et al., 2019; Luu et al., 2016; Torres et al., 2018). Therefore, we primarily evaluate the overall performance of our approach on the RE, TD, OF, DE, BN, and UC vulnerabilities.

4.3. Baselines

To assess the EGFL, fourteen representative methods in the field of smart contract vulnerability detection are chosen as baselines. Other vulnerability detection methods are not included in our comparison. The main reason is that some methods (Huang et al., 2021; Mi et al., 2021; Wang et al., 2020) do not open source their source codes or some efforts like Smartstate (Liao et al., 2023) and Smartdagger (Liao et al., 2022) focus on different types of smart contract vulnerabilities.

Static/dynamic analysis tools mainly use predefined patterns to analyze the smart contract and detect potential vulnerabilities. Among them, we select the following representative static analysis (Feist et al., 2019; Tikhomirov et al., 2018; Torres et al., 2018) and dynamic analysis (Breidenbach, 2017; Nguyen et al., 2020; Torres et al., 2021) tools as the baselines.

Osiris (Torres et al., 2018) was an instrumentation tool that used symbolic execution and taint analysis for vulnerability detection.

Smartcheck (Tikhomirov et al., 2018) first converted the smart contract code into an intermediate representation and then performed vulnerability detection.

Slither (Feist et al., 2019) employed abstract syntax trees to generate an intermediate representation of the smart contract, called SlithIR, which was used to analyze vulnerability.

Mythril (Breidenbach, 2017) incorporated taint analysis and control flow checking to detect potential vulnerabilities.

sFuzz (Nguyen et al., 2020) generated the execution traces and used vulnerability analysis patterns to identify vulnerabilities.

Confuzzius (Torres et al., 2021) primarily employed a hybrid fuzzy tester consisting of symbolic execution and fuzzy testing to analyze the potential vulnerabilities.

Deep learning-based detection methods can use deep learning techniques to automatically learn the vulnerability features and can be broadly divided into two categories: sequence-based methods (Sendner et al., 2023; Tann et al., 2018) and graph-based methods (Liu et al., 2021, 2023b; Qian et al., 2023; Zeng et al., 2022; Zhuang et al., 2020). These methods are described below.

Both Escort (Sendner et al., 2023) and SaferSC (Tann et al., 2018) took the opcode sequence as the model input. Following this, they adopted the techniques of natural language processing to detect vulnerabilities, containing the Gated Recurrent Unit (GRU) model (Chung et al., 2015) and the Long Short Term Memory (LSTM) model (Shi et al., 2015), respectively.

TMP (Zhuang et al., 2020), CGE (Liu et al., 2023b), and AME (Liu et al., 2021) constructed a simplified code semantic graph of smart contracts. Subsequently, TMP utilized a temporal message propagation network (Gilmer et al., 2017) to extract the deep graph feature for vulnerability detection. CGE and AME combined the code semantic graph and expert patterns to identify the vulnerabilities.

Both SCGCN (Zeng et al., 2022) and SMS (Qian et al., 2023) were fed with CFG. Then, these two baselines used the GCN and GAT models to learn graph features for vulnerability detection, respectively.

While there are existing vulnerability detection methods (Fu and Tantithamthavorn, 2022; Li et al., 2021; Tang et al., 2023) available for other software vulnerabilities, they have not yielded satisfactory results in detecting vulnerabilities in smart contracts. Here, the best-performing method, IVDetect (Li et al., 2021), is still chosen as our baseline for comparison. IVDetect utilized the GCN model to learn the graph feature from the program dependency graph.

4.4. Evaluation metrics

We evaluate the performance of all methods using four evaluation metrics, including accuracy, recall, precision, and F1 score. These metrics are widely used in the field of software vulnerability detection (Chen et al., 2023; Liu et al., 2023a; Pasqua et al., 2023; Tang et al., 2023; Zhang et al., 2022b). Note that a model with high accuracy, recall, precision, and F1 score is regarded as effective in detecting vulnerabilities in smart contracts. The calculation process for these metrics is listed below:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}, \tag{10}$$

$$recall = \frac{TP}{TP + FN}, \tag{11}$$

$$precision = \frac{TP}{TP + FP}, \tag{12}$$

$$F1score = 2 \times \frac{recall \times precision}{recall + precision}, \tag{13}$$

where TP and TN refer to the number of vulnerable and non-vulnerable smart contracts that are correctly predicted. FP denotes the number of vulnerable smart contracts that are predicted to be secure and non-vulnerable. FN means the number of safe smart contracts that are predicted to be vulnerable.

4.5. Implementation details

To ensure a fair comparison, the experiments of all baselines are replicated on the same benchmark dataset by using the same hyper-parameter settings as given in their paper. Following previous settings (Qian et al., 2023), all methods are fed with the smart contract bytecode. Since each node of CFG contains a set of operation statements

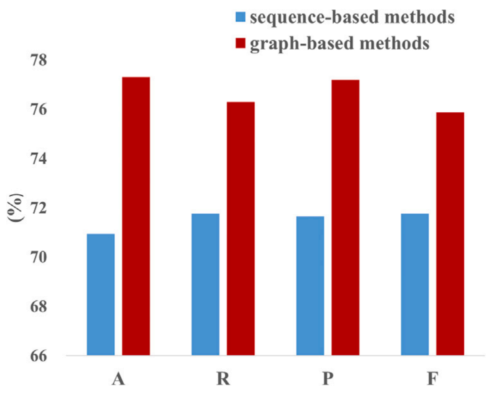


Fig. 7. Average performance comparison between all graph-based methods (SCGCN, TMP, CGE, AME, SMS, and EGFL) and sequence-based methods (SaferSC and Escort) across all vulnerabilities.

(i.e., tokens), the word2vec algorithm (Pennington et al., 2014) is used to vectorize each token, and compute the average of all token vectors as the initial embedding vector of the node. The implementation of our approach is based on Python 3.8 with some open-source packages including TensorFlow, Keras, and Scikit-learn. All experiments are conducted in an Intel Core i5 CPU at 2.5 GHz and a GeForce RTX 3090 GPU. The hyperparameters of the experiment are given below. The dimension (k) of the embedding vectors of all graph nodes is set to 256. The number of the GCN layers and the attention head in the relationship-aware module are set to 2 and 10, respectively. Besides, the vector dimensions of both the local node feature and the global node information are set to 200. In the model training, EGFL is updated by using the Adam optimizer with a learning rate of 0.001, a batch size of 32, and 100 maximum epochs. Each experiment is repeated five times, and the average value is considered as the outcome.

5. Experimental results

In this section, we answer the above research questions one by one based on the experimental results.

5.1. RQ1: Effectiveness of EGFL

To evaluate the effectiveness of EGFL, we compare EGFL against some representative baseline methods across six types of vulnerabilities, including RE, TD, OF, DE, UC, and BN. Also, the average (Avg) performance of these methods on all datasets is recorded for a comprehensive comparison. Notably, some baselines, such as Mythril and Slither, cannot detect certain categories of vulnerabilities in smart contracts, which are denoted as “n/a”. These special cases are not included in the calculation of the average performance. We record all the results in Table 2 and analyze the results from the following perspectives.

Comparison with static/dynamic analysis tools. We first compare the EGFL with six static/dynamic analysis tools, and the performances of these tools are listed in columns 3 through 8 of Table 2. Firstly, all static/dynamic analysis tools cannot support the detection task for all vulnerabilities, whereas our EGFL can. For example, Osiris, Slither, Mythril, and sFuzz are unable to identify the UC vulnerability (see lines 19 to 22). This observation suggests that existing static/dynamic analysis tools are limited to detecting various types of smart contract vulnerabilities. Secondly, EGFL dramatically outperforms existing static/dynamic analysis tools in all vulnerability scenarios. For instance, sFuzz has a relatively poor performance and drops EGFL by 35.19%, 60.13%, 57.89%, and 60.28% in the average accuracy, recall, precision, and F1 score, separately. Even compared

Table 2
Comparison results between EGFL with fourteen baselines in terms of Accuracy (**A**), Recall (**R**), Precision (**P**), and F1 score (**F**). Results reported in bold are the best performance. "n/a" denotes that the baseline cannot detect this vulnerability scenario. Smartcheck and Confuzzius are abbreviated as Scheck and Confuz, separately.

Dataset		Method														
		Scheck	Osiris	Slither	Mythril	sFuzz	Confuz	SaferSC	Escort	SCGCN	TMP	CGE	AME	SMS	IVDetect	EGFL
RE	A(%)	54.65	56.73	74.02	64.27	55.69	73.81	70.54	66.67	73.21	76.45	78.76	81.06	83.85	71.72	**90.32**
	R(%)	16.34	63.88	73.50	75.51	14.95	73.66	72.35	72.78	73.18	75.30	76.88	78.45	77.48	75.86	**91.71**
	P(%)	45.71	40.94	74.44	42.86	10.88	69.56	71.24	68.48	74.47	76.04	77.83	79.62	79.46	72.56	**89.26**
	F(%)	24.07	49.90	73.97	54.68	12.59	71.22	71.27	71.25	73.82	75.67	77.35	79.03	78.46	73.25	**90.47**
TD	A(%)	47.73	66.83	68.52	62.40	33.41	n/a	65.76	65.45	75.91	78.84	80.55	82.25	89.77	71.49	**90.72**
	R(%)	79.34	55.42	67.17	49.80	27.01	n/a	58.34	65.23	77.55	76.09	78.18	80.26	**91.09**	71.35	88.41
	P(%)	47.89	59.26	69.27	57.50	23.15	n/a	68.57	64.16	74.93	78.68	80.05	81.42	89.15	75.58	**92.71**
	F(%)	59.73	57.28	68.20	53.37	24.93	n/a	65.67	63.78	76.22	77.36	79.10	80.84	**90.11**	74.76	90.03
OF	A(%)	53.91	68.41	n/a	n/a	45.50	60.24	70.46	68.14	67.53	70.85	72.05	73.24	79.36	69.25	**88.93**
	R(%)	68.54	34.18	n/a	n/a	25.97	66.43	71.43	70.23	70.93	69.47	70.53	71.59	72.98	70.52	**87.54**
	P(%)	42.81	60.83	n/a	n/a	25.88	53.89	68.48	69.56	69.52	70.26	70.81	71.36	78.14	69.46	**89.31**
	F(%)	52.70	43.77	n/a	n/a	25.92	52.77	70.23	69.14	70.22	69.86	70.67	71.47	75.47	69.86	**88.56**
DE	A(%)	62.41	n/a	68.97	75.06	64.37	71.35	66.02	63.25	65.76	69.11	70.98	72.85	78.82	66.04	**84.48**
	R(%)	56.21	n/a	52.27	62.07	47.22	63.17	71.49	67.24	69.74	70.37	69.89	69.40	73.69	69.71	**79.53**
	P(%)	45.56	n/a	70.12	72.30	58.62	63.23	66.52	64.88	69.01	68.18	69.22	70.25	76.97	69.15	**86.25**
	F(%)	50.33	n/a	59.89	66.80	52.31	56.48	68.89	65.44	69.37	69.26	69.54	69.82	75.29	68.24	**83.67**
UC	A(%)	78.31	n/a	n/a	n/a	n/a	66.46	72.71	79.54	72.85	73.48	n/a	n/a	65.58	71.77	**87.82**
	R(%)	37.16	n/a	n/a	n/a	n/a	70.67	77.31	77.54	69.48	78.66	n/a	n/a	75.16	70.09	**84.36**
	P(%)	15.78	n/a	n/a	n/a	n/a	59.68	70.82	80.44	70.25	75.92	n/a	n/a	66.31	70.67	**83.72**
	F(%)	18.49	n/a	n/a	n/a	n/a	62.57	73.92	79.51	69.82	66.77	n/a	n/a	72.84	69.80	**85.26**
BN	A(%)	n/a	n/a	n/a	n/a	65.32	59.48	80.84	81.84	70.69	70.71	n/a	n/a	74.27	70.10	**86.03**
	R(%)	n/a	n/a	n/a	n/a	14.79	65.28	78.68	78.61	70.69	70.71	n/a	n/a	74.27	78.08	**85.15**
	P(%)	n/a	n/a	n/a	n/a	31.58	72.78	82.93	83.72	78.75	78.84	n/a	n/a	75.03	80.08	**86.77**
	F(%)	n/a	n/a	n/a	n/a	19.46	69.45	80.73	81.42	66.67	66.73	n/a	n/a	66.75	71.76	**85.93**
Avg	A(%)	59.40	63.99	72.46	67.24	52.86	66.27	71.06	70.82	70.99	73.24	75.58	77.35	78.61	70.06	**88.05**
	R(%)	51.52	51.16	57.53	62.46	25.99	67.84	71.60	71.94	71.93	73.43	73.87	74.93	77.45	72.60	**86.12**
	P(%)	39.55	53.68	57.40	57.55	30.02	63.83	71.43	71.87	72.82	74.65	74.48	75.66	77.51	72.92	**88.00**
	F(%)	41.06	50.32	55.14	58.28	27.04	62.50	71.79	71.76	71.02	70.94	74.16	75.29	76.49	71.28	**87.32**

to Confuzzius, EGFL still outperforms it by 21.78%, 18.27%, 24.18%, and 24.82% in the corresponding metrics.

These experimental results further demonstrate the great advantages of our approach over existing static/dynamic analysis tools. This may come from the fact that these tools generally analyze vulnerable code according to a certain predefined specification, thus disregarding a whole family of potential vulnerabilities in smart contracts. As a result, the ability of static/dynamic analysis tools can be constrained when dealing with such vulnerability scenarios.

Comparison with deep learning-based detection methods. We then benchmark EGFL against seven deep learning-based detection methods. Among them, SaferSC and Escort belong to the sequence-based methods. SCGCN, TMP, CGE, AME, and SMS are the graph-based methods. Quantitative results of these deep learning-based baselines are summarized in columns 9 to 15 of Table 2, and we have the following observations.

Firstly, EGFL consistently outperforms all deep learning-based baselines in terms of average performance. More specifically, EGFL gains an average accuracy, recall, precision, and F1 score of 88.05%, 86.12%, 88.00%, and 87.32%, respectively. It achieves an average improvement of 9.44%, 8.67%, 10.49%, and 10.83% over the SMS, as the existing top-performing baseline. Furthermore, among all 24 combination cases across the six vulnerability scenarios, EGFL performs better than SMS in the majority of cases (22 out of 24). These results highlight the effectiveness of EGFL in detecting smart contract vulnerabilities over existing vulnerability detection methods. The strength of EGFL lies in its ability to enhance the graph feature of the bytecode by efficiently fusing the global node information and local node feature. This enhanced graph feature can represent comprehensive structure information in the smart contract bytecode, which helps to capture more vulnerable code patterns.

Secondly, we find that the overall performance of graph-based vulnerability detection methods surpasses that of sequence-based vulnerability detection methods. To facilitate an effective comparison, we calculate the average performance of the graph-based methods and sequence-based methods across all vulnerability scenarios, as depicted in Fig. 7. It is evident that the graph-based methods outperform sequence-based methods, particularly in terms of accuracy and precision metrics. The reason behind this distinction is that sequence-based methods treat bytecode as a sequence structure, potentially overlooking the graph structural information (i.e., program structural information) embedded within the bytecode. This highlights the substantial advantages of graph-based methods in smart contract vulnerability detection.

Comparison with other software vulnerability methods. Finally, we compare EGFL with IVDetect, which serves as a representative software vulnerability detection method, and present the results in column 16 of Table 2. As can be seen, IVDetect does not achieve satisfactory results, despite its outstanding performance in detecting vulnerabilities in C/C++ programs. For instance, IVDetect only achieves an average accuracy, recall, precision, and F1 score of 70.06%, 72.60%, 72.92%, and 71.28% across all vulnerabilities. In comparison, EGFL outperforms IVDetect, exhibiting an improvement of 17.99%, 13.62%, 15.08%, and 16.04% in the corresponding metrics. Moreover, other methods in the fields of smart contract vulnerability detection, such as TMP, AME, and SMS, continue to outperform IVDetect in terms of overall performance (see lines 27 to 30). The performance gap of IVDetect is attributed to the variations in diverse vulnerability features and characteristics encountered in the software programs.

Answer to RQ1: EGFL outperforms the existing state-of-the-art methods in terms of average performance. Specifically, EGFL gains an improvement of 9.44%-35.19%, 8.67%-60.13%, 10.49%-57.98% and 10.83%-60.28% in the accuracy, recall, precision, and F1 score, respectively.

Table 3
The performance comparison of EGFL with two variants (including “w/o Local” and “w/o Global”) on six types of vulnerabilities.

Dataset		Method		
		w/o Local	w/o Global	EGFL
RE	A(%)	76.78	83.05	**90.32**
	R(%)	77.97	80.32	**91.71**
	P(%)	74.37	81.96	**89.26**
	F(%)	78.23	82.33	**90.47**
TD	A(%)	78.65	85.42	**90.72**
	R(%)	80.69	83.04	**88.41**
	P(%)	77.64	84.39	**92.71**
	F(%)	78.75	86.37	**90.03**
OF	A(%)	77.57	84.04	**88.93**
	R(%)	70.83	81.16	**87.54**
	P(%)	72.28	78.73	**89.31**
	F(%)	71.58	82.93	**88.56**
DE	A(%)	76.67	75.02	**84.48**
	R(%)	74.16	71.93	**79.53**
	P(%)	79.04	77.82	**86.25**
	F(%)	77.63	73.71	**83.67**
UC	A(%)	78.17	74.78	**87.82**
	R(%)	79.27	75.42	**84.36**
	P(%)	78.06	75.99	**83.72**
	F(%)	79.44	76.32	**85.26**
BN	A(%)	80.23	72.17	**86.03**
	R(%)	79.54	77.82	**85.15**
	P(%)	80.95	77.66	**86.77**
	F(%)	78.24	73.22	**85.93**
Average	A(%)	78.01	79.08	**88.61**
	R(%)	77.08	78.28	**86.80**
	P(%)	77.06	79.43	**89.38**
	F(%)	77.31	79.15	**88.18**

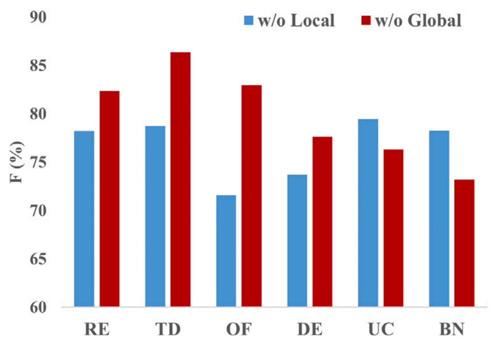


Fig. 8. The F1 score comparison of the two variants (including “w/o Local” and “w/o Global”) on six vulnerability scenarios.

5.2. RQ2: Effect of different graph features

EGFL can effectively extract both the local node feature and global node information in the graph. These two types of graph features are fused to generate an enhanced graph feature for smart contract vulnerability detection. It is thus interesting to evaluate the effect of these two types of features on the EGFL from the following perspectives.

The individual contribution of each graph feature on the EGFL. We first evaluate the individual contributions of each graph feature on the performance of EGFL. To explore this, we introduce two variants of EGFL which remove each category of feature, respectively, and only keep the remaining feature to identify vulnerabilities. Among them, the “w/o Local” variant keeps the global node information, and the “w/o Global” variant retains the local node feature. We compare the performance of these two variants across all six vulnerability scenarios to assess their contribution to EGFL. Quantitative results are presented in Table 3 and Fig. 8, and we have the following observations.

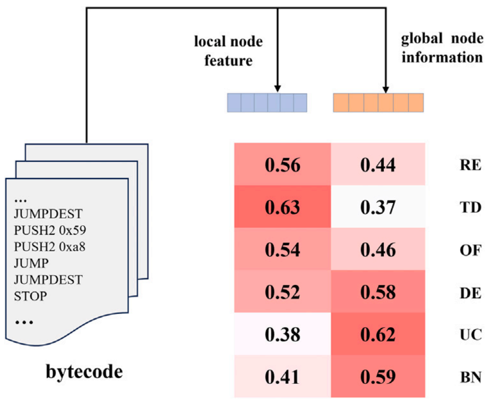


Fig. 9. The weights assigned to two different features (including the local node feature and the global node information) in EGFL when detecting all vulnerabilities. Note that the redder the color, the higher the feature weights. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Firstly, both variants that utilize only one category of graph feature exhibit a degradation in performance compared to EGFL. For instance, “w/o Local” achieves an average accuracy, recall, precision, and F1 score of 78.01%, 77.08%, 77.06%, and 77.31% across all vulnerabilities, respectively (see lines 27 to 30). This variant experiences a decrease of 10.60%, 9.72%, 12.32%, and 10.87% compared to EGFL, respectively. Similarly, “w/o Global” also witnesses a decline of 9.53%, 8.52%, 9.95%, and 9.03% in the corresponding metrics. These findings demonstrate that our approach benefits from the synergy of both the local node feature and the global node information, since these two types of features can be fused to enhance the learning of the graph features.

Secondly, “w/o Local” performs better than “w/o Global” on the DE, UC and BN vulnerability scenarios (see lines 15 to 26). In contrast, “w/o Global” beats “w/o Local” on the remaining three kinds of vulnerabilities. This finding suggests that the local node feature and global node information can lead to different roles in vulnerability detection. In other words, these two types of features can complement each other in smart contract vulnerability detection. Therefore, it is indeed necessary to combine both the local node feature and the global node information to identify smart contract vulnerabilities.

Furthermore, the local node feature contributes more to the average performance of EGFL than global node information. As presented in lines 27 to 30 of Table 3, “w/o Global” achieves an average accuracy, recall, precision, and F1 score of 79.08%, 78.28%, 79.43%, and 79.15% on all vulnerability scenarios. It surpasses the performance of “w/o Local”, demonstrating an average improvement of 1.07%, 1.20%, 2.37%, and 1.84% in the respective metrics.

The weights of all graph features in vulnerability detection. As mentioned above, the local node feature and global node information play distinct roles in vulnerability detection. We aim to further assess the roles in a visualization way to facilitate understanding. Specifically, the local node feature is associated with the local features of nodes through neighborhood aggregation, while the global node information captures the long-range dependencies between the nodes. To analyze these roles, we examine the weights assigned to these two types of features within EGFL when detecting different vulnerabilities.

Following (Liu et al., 2021), we calculate the inner product between each feature and the enhanced graph feature, as discussed in Section 3.4. These inner products serve as the corresponding weights assigned to each feature for vulnerability detection, as described in Fig. 9. For instance, when using EGFL to detect the RE vulnerability, the weights assigned to the local node feature and the global node information are 0.56 and 0.44, respectively. This indicates that the local node feature plays a more crucial role in detecting the RE vulnerability, compared to the global node information. Additionally, the

Table 4
Performance comparison between EGFL with its variants on the four kinds of vulnerability scenarios.

Dataset		Method														
		1-GCN	3-GCN	GGNN	GAT	2-NFE	3-NFE	No-FA	150-FA	6-head	15-head	RNN	GRU	LSTM	BERT	EGFL
RE	A(%)	83.88	90.58	88.42	**90.85**	84.56	84.32	88.95	90.46	78.39	84.12	81.04	83.84	82.58	89.42	90.32
	R(%)	82.69	85.47	86.47	84.83	80.95	72.47	86.83	89.26	79.93	86.51	77.35	79.84	82.06	81.64	**91.71**
	P(%)	81.47	85.36	84.81	86.46	82.48	75.58	**90.97**	85.39	82.59	87.96	83.42	82.57	80.91	83.94	89.26
	F(%)	82.34	87.94	86.55	85.37	83.75	73.95	89.86	88.36	81.76	86.10	80.23	83.52	81.53	82.39	**90.47**
TD	A(%)	83.66	89.94	85.28	89.46	74.58	80.64	84.03	87.48	81.87	87.60	69.73	74.68	72.95	**93.55**	90.72
	R(%)	84.48	87.46	86.58	90.06	82.43	80.97	83.95	84.18	77.94	84.52	77.43	77.79	81.03	**93.32**	88.41
	P(%)	82.21	87.39	87.15	89.17	73.05	82.83	76.57	85.38	79.68	85.05	67.38	73.64	71.17	**93.61**	92.71
	F(%)	83.26	87.65	83.52	90.09	76.74	81.59	80.06	85.59	81.82	86.16	65.84	75.68	75.75	**92.93**	90.03
OF	A(%)	84.43	**89.21**	87.57	79.36	82.32	75.07	84.46	86.18	83.18	88.91	81.73	83.89	84.49	85.64	88.93
	R(%)	79.47	83.38	80.16	82.88	77.68	70.38	82.05	86.88	79.68	86.26	76.16	79.37	78.99	74.32	**87.54**
	P(%)	80.48	87.94	82.26	88.19	84.17	75.96	82.55	**90.37**	80.44	85.81	76.94	86.74	83.57	85.47	89.31
	F(%)	81.58	**95.99**	81.59	85.55	83.82	73.06	84.28	84.86	78.73	83.07	71.43	83.37	84.16	81.09	88.56
DE	A(%)	77.48	81.72	78.93	82.88	80.44	74.76	80.86	80.84	78.60	84.33	68.53	79.63	79.11	82.76	**84.48**
	R(%)	75.02	73.55	78.05	77.73	79.85	71.89	80.02	81.63	78.11	**84.67**	65.76	78.62	79.83	77.93	79.53
	P(%)	78.13	83.72	81.83	81.03	78.96	72.52	81.58	83.15	80.09	85.46	69.73	79.83	78.42	84.61	**86.25**
	F(%)	76.93	81.18	80.36	79.32	79.77	72.2	80.83	80.96	80.09	**84.43**	67.18	79.82	79.14	81.15	83.67
Avg	A(%)	82.36	87.86	85.05	85.64	80.48	78.70	84.58	86.24	80.51	86.24	75.26	80.51	79.78	87.84	**88.61**
	R(%)	80.42	82.47	82.82	83.88	80.23	73.93	83.21	85.49	78.91	85.49	74.18	78.91	80.48	81.80	**86.80**
	P(%)	80.57	86.10	84.01	86.21	79.67	76.72	82.92	86.07	80.70	86.07	74.37	80.70	78.52	86.91	**89.38**
	F(%)	81.03	85.69	83.01	85.08	81.02	75.20	83.76	84.94	80.60	84.94	71.17	80.60	80.15	84.39	**88.18**

local node feature always receives higher weights in the RE, TD, and OF vulnerability scenarios. The global node information obtains higher weights in the DE, UC, and BN vulnerabilities. This observation further emphasizes that these two kinds of graph features can play distinct roles in vulnerability detection.

Answer to RQ2: The global node information and local node feature in EGFL can lead to distinct roles in smart contract vulnerability detection. Besides, EGFL outperforms that only using the local node feature by 9.53%, 8.52%, 9.95%, and 9.03% in accuracy, recall, precision, and F1 score, separately.

5.3. RQ3: Effect of different feature extraction modules

EGFL adopts two groups of deep learning-based feature extraction modules to learn two types of feature information. Among them, the graph neural network and node feature extractor are applied to extract the local node feature. The feature-aware and relationship-aware modules are combined to extract the global node information. To reveal more details of the EGFL framework, we further investigate the influence of these modules on EGFL.

Study on the modules for extracting local node feature. We first investigate the impact of the graph neural network. By default, our GCN model incorporates two convolutional layers to update the feature vectors of all graph nodes. In comparison, four variants are set which replace the default GCN model with other graph neural networks, including "1-GCN", "3-GCN", "GGNN", and "GAT", as presented in columns 3 through 6 of Table 4. Among them, the "1-GCN" and "3-GCN" variants mean the GCN model with one and three convolutional layers, separately. The "GGNN" and "GAT" are the Gated Graph Neural Network (GGNN) (Li et al., 2017) and the GAT model (Veličković et al., 2018), respectively, which are other advanced graph neural networks.

Clearly, our default GCN model demonstrates superior performance compared to the other variants. For instance, it outperforms "1-GCN" by 6.25%, 6.38%, 8.81%, and 7.16% in the average accuracy, recall, precision, and F1 score (see lines 19 to 22). Furthermore, as the number of GCN layers increases from 2 to 3, there is a noticeable decline in the overall performance. This observation suggests that the GCN model might encounter an over-smoothing problem. Specifically, as more convolutional layers are added to the GCN model, the feature vectors of all nodes become similar and indistinguishable. Consequently, this leads to performance degradation of EGFL. Even compared to the GGNN and GAT models, our default GCN model still has an advantage. The reason may be that GCN updates graph features by aggregating node information with neighboring node information, which better captures the semantics between nodes and improves code representation.

Then, we study the impact of the node feature extractor on EGFL. By default, our node feature extractor has a basic block that consists of the one-dimensional convolutional layer, ReLU activation function, and maxpooling layer. We create the "2-NFE" and "3-NFE" variants which contain two and three basic blocks, separately. Unfortunately, both variants fail to achieve satisfactory performance. The reason may be that applying more basic blocks in the node feature extractor does not contribute to the extraction of critical elements in each node vector, thereby degrading the learning of the local node feature.

Study on the modules for extracting global node information. Firstly, we examine the effect of the relationship-aware module, which mainly comprises the multi-head attention mechanism. Specifically, we introduce four representative models, i.e., Recurrent Neural Network (RNN) (Wang and Niepert, 2019), GRU (Chung et al., 2015), LSTM (Shi et al., 2015), and Bidirectional Encoder Representations from Transformers (BERT) (Devlin et al., 2019), from the field of natural language processing and create the corresponding variants. Among them, both the "BERT" and our approach apply the structure of the multi-head attention mechanism, and perform better than the remaining variants. This suggests that the multi-head attention mechanism does handle long-range dependencies between nodes and captures the corresponding global node information. However, the BERT model often limits the input length of a given code, which may result in the absence of some vulnerability-related code information (Zhang et al., 2023). This limitation makes the "BERT" variant perform less well than our approach.

Furthermore, our default multi-head attention mechanism contains 10 attention heads, and sets two variants ("6-head" and "15-head") that have 6 and 15 attention heads separately. As can be seen, our EGFL consistently achieves more satisfactory performances on all vulnerabilities compared to the other seven variants. This finding indicates that employing more attention heads leads to more significant performance gains in our approach. The reason behind this is that a larger number of attention heads enhances the ability to handle long-range dependencies between nodes, and thereby learn the global node information. However, it is worth noting that the performance of EGFL does not consistently improve beyond 10 attention heads.

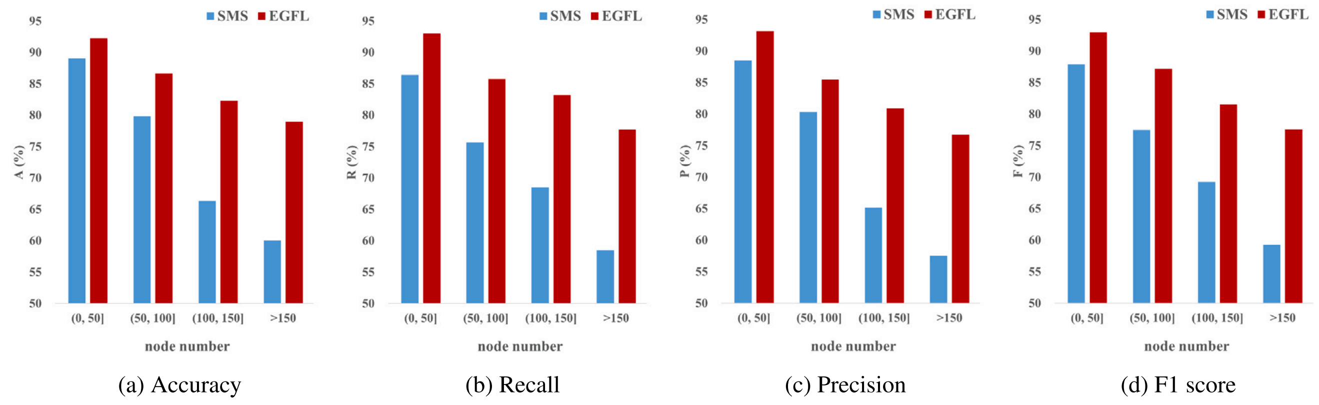


Fig. 10. The performance (accuracy, recall, precision, and F1 score) comparison of EGFL and the SMS with different numbers of graph nodes.

Subsequently, we investigate the impact of the feature-aware module on the performance of EGFL. As mentioned in Section 3.3, the feature-aware module is designed to ignore some vulnerability-irrelevant nodes with the help of convolutional neural networks before the relationship-aware module. Indeed, this module is equal to a semantic compressor that reduces the dimensionality of the node feature matrix and filters out irrelevant code information. In default, after the feature-aware module, the dimension of each node feature vector is reduced from 256 to 200. As a comparison, we set the "150-FA" variant with the dimension of each node feature vector reduced to 150. Also, we create the "No-FA" variant where there is no feature-aware module. The "150-FA" variant achieves a decline of 2.37%, 1.31%, 3.31%, and 3.24% in the average accuracy, recall, precision, and F1 score. This suggests that compressing too much code information is not an ideal design. In addition, the disadvantages of "No-FA" over EGFL and "150-FA" prove the effectiveness of the feature-aware module in capturing the critical nodes.

Answer to RQ3: Different settings of feature extraction modules can affect the detection performance of EGFL. The default settings of EGFL can obtain satisfactory performance.

6. Discussion

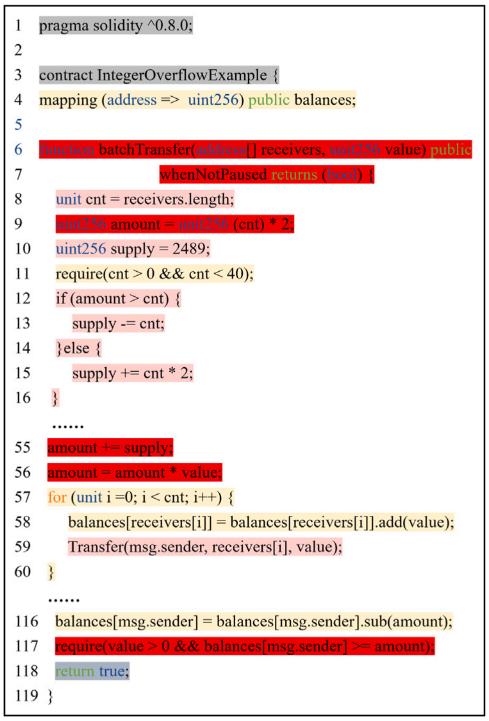
In this section, we first discuss the advantages of EGFL. Then, the efficiency analysis and potential threats are presented, separately.

6.1. Why does EGFL work?

Compared to existing vulnerability detection methods, our EGFL mainly has the following two advantages.

First, given the graph structure of bytecode, EGFL can effectively capture long-range dependencies between the nodes, and utilize the global node information and local node feature to enhance the detection performance. Regrettably, existing vulnerability detection methods, such as the state-of-the-art SMS, cannot do this. SMS primarily utilizes the GAT model to focus on the local node feature of the bytecode, while ignoring the global node information. To investigate this, the experiments are conducted on a test set (Qian et al., 2023) that focuses on the RE vulnerability mentioned in Section 4.2. By default, we represent the bytecode as a CFG. Based on the number of nodes within the CFG, the data inside the test set are divided into four node intervals. We compare the performance of the EGFL against SMS in terms of four evaluation metrics.

SMS achieves more optimal performance in the CFG with fewer nodes, as shown in Fig. 10. For instance, in the CFG containing no more than 50 nodes, SMS achieves an accuracy and F1 score of nearly


```
pragma solidity ^0.8.0;

contract IntegerOverflowExample {
mapping (address =>  uint256) public balances;

function batchTransfer(address[] receivers, unit256 value) public
                         whenNotPaused returns (bool) {
   unit cnt = receivers.length;
   uint256 amount = unit256 (cnt) * 2;
   uint256 supply = 2489;
   require(cnt > 0 && cnt < 40);
   if (amount > cnt) {
      supply -= cnt;
   }else {
      supply += cnt * 2;
   }
  ......
  amount += supply;
  amount = amount * value;
  for (unit i =0; i < cnt; i++) {
     balances[receivers[i]] = balances[receivers[i]].add(value);
     Transfer(msg.sender, receivers[i], value);
  }
  ......
   balances[msg.sender] = balances[msg.sender].sub(amount);
   require(value > 0 && balances[msg.sender] >= amount);
   return true;
}
```

Fig. 11. An example of real-world smart contracts with the integer overflow vulnerability. The red-shaded and gray-shaded denote the maximum and minimum attention weights, respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

90%. However, as the number of nodes increases, the performance of SMS significantly declines. Notably, when the number of nodes exceeds 150, SMS only obtains an accuracy and F1 score of approximately 60%. In contrast, EGFL consistently outperforms SMS across all four node intervals. Particularly, EGFL exhibits a substantial advantage over SMS in the CFG with a larger number of nodes. This suggests that some vulnerabilities in smart contracts can be identified only when establishing long-range dependencies between the nodes, especially for the CFG with more than 100 nodes. Therefore, we are inspired to extract the global node information in CFG and utilize it to improve the performance of vulnerability detection.

Second, EGFL has advantages in the learning of the global node information. Practically, the graph neural networks with stacked multiple layers can be leveraged to learn the global node information, such as the "3-GCN" variant in Section 5.3. Regrettably, this can lead to the overfitting issue i.e. the graph features of different smart contracts

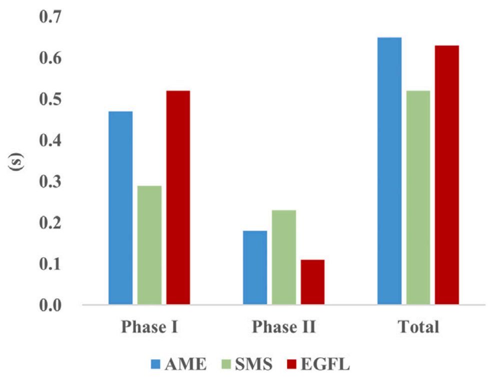


Fig. 12. Average time cost of EGFL and two representative methods (AME and SMS) on all vulnerability datasets.

have a similar feature space (Liu et al., 2023a), which makes it hard to distinguish between vulnerable and non-vulnerable smart contracts. For example, "3-GCN" is not as good as our EGFL in terms of overall performance, as shown in Table 4. The GGNN model (Li et al., 2017), i.e., the "GGNN" variant in Section 5.3, can also be applied to extract the global node information, since it uses the RNN model to handle all the nodes and establish the long-range dependencies. This process needs to store intermediate states of nodes in the memory, which may make it hard to generalize to large-scale smart contracts (Dong et al., 2023). Another strategy is to simplify the graph. To obtain more accurate vulnerability features, existing vulnerability detection methods (CGE (Liu et al., 2023b) and AME (Liu et al., 2021)) generally eliminate some irrelevant node information by simplifying the graph. This operation is actually an attempt to capture some global node information, although they do not explicitly mention this. However, this graph simplification operation may remove some potentially important node information, thus affecting the final detection performance.

To understand whether our approach is concerned with vulnerable code statements, the attention heatmap of a real-world smart contract example is given in Fig. 11. Following (Wen et al., 2023), we assign a color shade to each code statement based on the attention weight of its corresponding node vector. Each node in the CFG corresponds to a line of code statement. A code statement with a redder shade indicates a higher attention weight, suggesting a higher likelihood of vulnerability. This example is a complete version of the one depicted in Fig. 1. As mentioned in Section 2, it involves an integer overflow vulnerability. Specifically, there is no overflow judgment for the "amount" variable (line 9). If the "amount" variable is manipulated to overflow (lines 6, 7, 55, and 56), the attackers can bypass the code responsible for verifying the "balance" (line 117). Consequently, attackers can transfer a substantial number of tokens at a minimal expense. Vulnerable code statements can be found on lines 6, 7, 9, 55, 56, and 117. As can be seen, EGFL concentrates on vulnerable code statements (marked in red), enabling accurate detection of this vulnerability.

6.2. *Efficiency analysis of EGFL*

Here, we analyze the efficiency of EGFL in detecting smart contract vulnerabilities, and the efficiency is measured with the time cost. We compare EGFL with two state-of-the-art deep-learning detection methods (AME (Liu et al., 2021) and SMS (Qian et al., 2023)). Similar to Zhang et al. (2023), after training the deep learning model, we use the trained model to calculate the average time cost of three methods (AME, SMS, and our EGFL) on all vulnerability datasets. As described in Fig. 12, the time cost consists of two phases, i.e., feature extraction (Phase I) and vulnerability detection (Phase II). For the feature extraction phase, all methods utilize the GCN or GAT model to learn the corresponding graph features. Besides, EGFL extracts the global node information with the feature-aware and relationship-aware modules. AME adopts the MLP network to extract the expert patterns. Thus, these two methods need some additional cost. In the vulnerability detection phase, SMS takes more time to get predicted results, since it uses the complex single-modality student network to handle the extracted graph features. Overall, SMS has less time overhead in detecting vulnerabilities, but does not have a significant gap over EGFL. In contrast, the detection performance of EGFL has a great improvement compared to SMS. We think the extra time overhead of EGFL is worth it.

6.3. *Threats to validity*

In this part, we investigate three categories of validities, including internal validity, external validity, and construct validity, respectively.

Internal validity may be threatened by implementation errors in our work, such as the representation of the smart contract bytecode. We adopt a public automated tool in Qian et al. (2023) to analyze the bytecode and construct a CFG to represent the bytecode. This public tool mainly considers three categories of control flow edges and a limited number of cases regarding jump destinations. Therefore, we have to acknowledge that the precision of the constructed CFG may not be high enough. For example, there is a case where the jump destination is not an opcode parameter, which affects the precise representation of bytecode semantics. Some recent work (Pasqua et al., 2023) discusses this task of building the precise CFG. Inspired by it, we plan to build a more precise CFG, aiming to further improve the performance of vulnerability detection. Moreover, since smart contracts have high-security requirements, it is difficult for us to collect and obtain enough vulnerable smart contracts (Liao et al., 2023; Vacca et al., 2021). To mitigate this, we design EGFL to mine as many vulnerability-related code features as possible from the benchmark vulnerability dataset. Meanwhile, we will try to expand the data volume of the vulnerability dataset, aiming to enable EGFL to learn more vulnerability features.

External validity might be threatened by generalization issues. Existing vulnerability detection efforts (Chen et al., 2023; Liu et al., 2023a, 2021; Qian et al., 2023) focus on detecting four types of vulnerabilities in smart contracts (including RE, TD, OF, and DE). Indeed, they have achieved relatively satisfactory results in these vulnerability scenarios. However, it may be more practical for these efforts to identify more kinds of vulnerabilities in smart contracts. Therefore, we investigate the vulnerability patterns of massive smart contracts on Ethereum and introduce two common vulnerabilities (including BN and UC). We compare EGFL with existing efforts on these six kinds of vulnerabilities. Nevertheless, we acknowledge that our approach cannot identify all vulnerabilities in Ethereum smart contracts. In future work, we plan to explore more categories of vulnerabilities in smart contracts to improve the generalizability of our approach.

Construct validity of our performance evaluation may be a concern. In other words, using only one or two evaluation indicators may mislead us into drawing the wrong conclusions. To address this, we introduce four common evaluation metrics, containing the accuracy, precision, recall, and F1 score, respectively. These metrics are widely used in the field of smart contract vulnerability detection (Chen et al., 2023; He et al., 2023; Liu et al., 2023a; Pasqua et al., 2023). By utilizing each metric to evaluate EGFL from multiple perspectives, we strengthen the rigor of our analysis of the experimental results.

7. Related work

In this section, we mainly introduce some related efforts in the field of smart contract vulnerability detection. Besides, we give other software vulnerability detection efforts.

7.1. Smart contract vulnerability detection

Recently, detecting vulnerabilities in smart contracts has gained increasing attention, as it can ensure the security of the blockchain system (Cai et al., 2023; Liao et al., 2023, 2022). As a result, a large number of efforts are proposed to identify vulnerabilities based on the smart contract bytecode, and they are roughly categorized into the following two groups.

Static/dynamic analysis tools mainly design various predefined patterns or specifications to identify vulnerable smart contracts. More specifically, in the static analysis, Luu et al. (2016) proposed Oyente to detect vulnerabilities by analyzing the control flow information of the smart contract bytecode, and this tool has not been maintained for many years. As an improved version of Oyente, Osiris (Torres et al., 2018) utilized symbolic execution and taint analysis to identify vulnerabilities, especially integer-like vulnerabilities. Tikhomirov et al. (2018) converted the contract code into an intermediate representation (e.g., XML-based parse tree) and checked vulnerabilities with XPath schema queries. Feist et al. (2019) adopted abstract syntax trees to generate an intermediate representation for analyzing the vulnerabilities. In the dynamic analysis, Mythril (Breidenbach, 2017) was designed to combine taint analysis and control flow checking based on symbolic execution to analyze vulnerabilities. Nguyen et al. (2020) adopted fuzzy testing techniques to generate smart contract execution traces for vulnerability detection. Torres et al. (2021) designed a hybrid test fuzzifier consisting of evolutionary fuzz testing and constraint-solving, and used it to identify vulnerabilities. These tools focus on analyzing smart contracts using predefined specifications, which makes it possible for them to overlook some potential vulnerabilities (Chen et al., 2023; Liu et al., 2023a).

Deep learning-based detection methods could employ deep learning techniques to automatically learn the vulnerable features for vulnerability detection. For instance, both SaferSC (Tann et al., 2018) and Escort (Sendner et al., 2023) transformed the bytecode into the opcode sequence and took it as the model input. Next, SaferSC and Escort introduced the typical LSTM (Shi et al., 2015) and GRU (Chung et al., 2015) models to extract the vulnerable features. Besides, Wang et al. (2020) employed machine learning-based classification algorithms for vulnerability detection. Nevertheless, these sequence-based methods assume that the bytecode is a sequence of tokens without considering the graph structure of the bytecode (Huang et al., 2021; Zeng et al., 2022). Therefore, existing vulnerability detection methods converted the bytecode into the graph structure. For example, Huang et al. (2021) represented the bytecode as the CFG and identified vulnerable smart contracts by measuring the similarity of graph features. Qian et al. (2023) and Zeng et al. (2022) utilized the GAT and GCN models respectively to handle the CFG and then obtained the results of vulnerability detection. Unfortunately, these graph neural networks are limited to dealing with long-range dependencies between graph nodes (Hellendoorn et al., 2019; Wen et al., 2023). This limitation makes existing graph-based methods ignore the global node information of the graph, which hinders the learning of the vulnerability features.

Furthermore, some deep learning-based efforts leverage the source code to detect the vulnerabilities. For example, both Zhuang et al. (2020) and Liu et al. (2021) constructed a code semantic graph that contained data flow and control flow information, and proposed TMP (Zhuang et al., 2020) and AME (Liu et al., 2021), respectively. Among them, TMP introduced a temporal message propagation network (Gilmer et al., 2017) to extract deep graph features. AME combined the deep graph features and multiple expert patterns to identify vulnerabilities. Similar to AME, CGE (Liu et al., 2023b) predicted the vulnerabilities by incorporating the extracted graph features and one expert pattern. However, these efforts are limited by the fact that over 98% of smart contract source code is not open-sourced while all bytecode is always accessible (Mi et al., 2021; Pasqua et al., 2023). It is thus crucial to identify smart contract vulnerabilities based on the bytecode.

7.2. Other software vulnerability detection

We also pay attention to some efforts (Dong et al., 2023; Fu and Tantithamthavorn, 2022; Li et al., 2021; Tang et al., 2023) used for other programs, such as Java, C, and C++. For instance, Dong et al. (2023) designed the relational graph convolutional network to learn both semantic and syntactic information from the code property graph of the source code. Fu and Tantithamthavorn (2022) treated the source code as the sequence structure, and proposed LineVul that employed a pre-trained model (i.e., CodeBERT) to embed the sequence tokens for vulnerability detection. Li et al. (2021) proposed IVDetect to capture the surrounding context relevant to the vulnerability through the program dependency graph. Next, IVDetect utilized the feature attention GCN model to detect vulnerabilities. Tang et al. (2023) used the sequence embedding and graph neural networks to extract the local semantic features and structured information of the CFG, respectively. Then, these two types of extracted features were combined to detect vulnerabilities. However, due to the inherent logic variations among different software vulnerabilities, these methods have not achieved the expected optimal results in smart contract vulnerability detection.

8. Conclusions and future work

In the paper, we proposed EGFL, a vulnerability detection framework for smart contracts with enhanced graph feature learning. Compared to existing vulnerability detection methods, EGFL could effectively learn the global node information from the graph structure of bytecode. Also, EGFL could utilize it to enhance the learning of the graph feature, resulting in a more comprehensive capture of the vulnerability features. We conducted a comprehensive evaluation of EGFL against existing state-of-the-art vulnerability detection methods on the benchmark dataset comprising six categories of vulnerabilities in smart contracts. Experimental results demonstrated the effectiveness of EGFL in detecting vulnerabilities in smart contracts through the enhanced graph feature learning strategy. Furthermore, the ablation study revealed more details about EGFL and explained its effectiveness in identifying smart contract vulnerabilities.

In the future, we plan to apply our approach to a broader range of real-world vulnerability scenarios, thereby enhancing the generalizability of EGFL across various types of vulnerabilities in smart contracts. To achieve this, we will collect the smart contract bytecode from diverse blockchain platforms, including Ethereum and VNT Chain.

CRediT authorship contribution statement

Jianxin Cheng: Writing – original draft, Visualization, Methodology, Investigation, Data curation, Conceptualization. **Yizhou Chen:** Writing – review & editing, Validation, Resources, Investigation, Formal analysis. **Yongzhi Cao:** Writing – review & editing, Validation, Supervision, Project administration, Funding acquisition. **Hanpin Wang:** Project administration, Supervision, Validation.

Declaration of competing interest

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

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by the National Key R&D Program of China under Grant 2021YFF1201102 and the National Natural Science Foundation of China under Grants 62172016 and 61932001.

References

Breidenbach, M., 2017. A framework for bug hunting on the ethereum blockchain. https://github.com/ConsenSys/mythril/. Accessed on 10 December 2023.

Cai, J., Li, B., Zhang, J., et al., 2023. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. J. Syst. Softw. 195, 111550.

Chen, D., Feng, L., Fan, Y., et al., 2023. Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention. J. Syst. Softw. 202, 111705.

Chung, J., Gulcehre, C., Cho, K., et al., 2015. Gated feedback recurrent neural networks. In: International Conference on Machine Learning. PMLR, pp. 2067–2075.

Devlin, J., Chang, M.W., Lee, K., et al., 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics. pp. 4171–4186.

Dong, Y., Tang, Y., Cheng, X., et al., 2023. SedSVD: Statement-level software vulnerability detection based on relational graph convolutional network with subgraph embedding. Inf. Softw. Technol. 158, 107168.

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

Feist, J., Grieco, G., Groce, A., 2019. Slither: A static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain. pp. 8–15.

Fu, M., Tantithamthavorn, C., 2022. Linevul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 608–620.

Gilmer, J., Schoenholz, S.S., Riley, P.F., et al., 2017. Neural message passing for quantum chemistry. In: International Conference on Machine Learning. PMLR, pp. 1263–1272.

He, J., Li, S., Wang, X., et al., 2023. Neural-FEBI: Accurate function identification in ethereum virtual machine bytecode. J. Syst. Softw. 199, 111627.

Hellendoorn, V.J., Sutton, C., Singh, R., et al., 2019. Global relational models of source code. In: International Conference on Learning Representations. pp. 1–12.

Huang, J., Han, S., You, W., et al., 2021. Hunting vulnerable smart contracts via graph embedding based bytecode matching. IEEE Trans. Inf. Forensics Secur. 16, 2144–2156.

Kipf, T.N., Welling, M., 2017. Semi-supervised classification with graph convolutional networks. In: International Conference on Learning Representations. pp. 1–14.

Li, Y., Tarlow, D., Brockschmidt, M., et al., 2017. Gated graph sequence neural networks. In: International Conference on Learning Representations. pp. 1–20.

Li, Y., Wang, S., Nguyen, T.N., 2021. Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 292–303.

Liao, Z., Hao, S., Nan, Y., et al., 2023. SmartState: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 980–991.

Liao, Z., Zheng, Z., Chen, X., et al., 2022. SmartDagger: A bytecode-based static analysis approach for detecting cross-contract vulnerability. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 752–764.

Liu, H., Fan, Y., Feng, L., et al., 2023a. Vulnerable smart contract function locating based on multi-relational nested graph convolutional network. J. Syst. Softw. 204, 111775.

Liu, Z., Qian, P., Wang, X., et al., 2021. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. In: Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence. pp. 2751–2759.

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

Liu, H., Yang, Z., Jiang, Y., et al., 2019. Enabling clone detection for ethereum via smart contract birthmarks. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension. pp. 105–115.

Luu, L., Chu, D.H., Olickel, H., et al., 2016. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269.

Mi, F., Wang, Z., Zhao, C., et al., 2021. VSCL: automating vulnerability detection in smart contracts with deep learning. In: 2021 IEEE International Conference on Blockchain and Cryptocurrency. pp. 1–9.

Nguyen, T.D., Pham, L.H., Sun, J., et al., 2020. Sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 778–788.

Pasqua, M., Benini, A., Contro, F., et al., 2023. Enhancing ethereum smart-contracts static analysis by computing a precise control-flow graph of ethereum bytecode. J. Syst. Softw. 200, 111653.

Pennington, J., Socher, R., Manning, C.D., 2014. Glove: Global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing. pp. 1532–1543.

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

Sendner, C., Chen, H., Fereidooni, H., et al., 2023. Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning. In: Proceedings of the 2023 Conference on Network and Distributed System Security Symposium. pp. 1–18.

Shi, X., Chen, Z., Wang, H., et al., 2015. Convolutional LSTM network: A machine learning approach for precipitation nowcasting. In: Proceedings of the 28th International Conference on Neural Information Processing Systems. pp. 802–810.

Tang, W., Tang, M., Ban, M., et al., 2023. CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection. J. Syst. Softw. 199, 111623.

Tann, W.J.W., Han, X.J., Gupta, S.S., et al., 2018. Towards safer smart contracts: A sequence learning approach to detecting security threats. arXiv preprint arXiv:1811.06632.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., et al., 2018. Smartcheck: Static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. pp. 9–16.

Torres, C.F., Iannillo, A.K., Gervais, A., et al., 2021. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In: 2021 IEEE European Symposium on Security and Privacy. IEEE, pp. 103–119.

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

Vacca, A., Di Sorbo, A., Visaggio, C.A., et al., 2021. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. J. Syst. Softw. 174, 110891.

Vaswani, A., Shazeer, N., Parmar, N., et al., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 30, 6000–6010.

Veličković, P., Cucurull, G., Casanova, A., et al., 2018. Graph attention networks. In: International Conference on Learning Representations. pp. 1–12.

Viglianisi, E., Ceccato, M., Tonella, P., 2020. A federated society of bots for smart contract testing. J. Syst. Softw. 168, 110647.

Wang, C., Niepert, M., 2019. State-regularized recurrent neural networks. In: International Conference on Machine Learning. PMLR, pp. 6596–6606.

Wang, W., Song, J., Xu, G., et al., 2020. Contractward: Automated vulnerability detection models for ethereum smart contracts. IEEE Trans. Netw. Sci. Eng. 8 (2), 1133–1144.

Wen, X.C., Chen, Y., Gao, C., et al., 2023. Vulnerability detection with graph simplification and enhanced graph representation learning. In: Proceedings of the 45th International Conference on Software Engineering. pp. 2275–2286.

Yang, Z., Keung, J., Yu, X., et al., 2021. A multi-modal transformer-based code summarization approach for smart contracts. In: 2021 IEEE/ACM 29th International Conference on Program Comprehension. pp. 1–12.

Yang, Z., Keung, J.W., Yu, X., et al., 2023. On the significance of category prediction for code-comment synchronization. ACM Trans. Softw. Eng. Methodol. 32 (2), 1–41.

Yuan, D., Wang, X., Li, Y., et al., 2023. Optimizing smart contract vulnerability detection via multi-modality code and entropy embedding. J. Syst. Softw. 202, 111699.

Zeng, Q., He, J., Zhao, G., et al., 2022. Ethergis: A vulnerability detection framework for ethereum smart contracts based on graph learning features. In: 2022 IEEE 46th Annual Computers, Software, and Applications Conference. pp. 1742–1749.

Zhang, J., Liu, Z., Hu, X., et al., 2023. Vulnerability detection by learning from syntax-based execution paths of code. IEEE Trans. Softw. Eng. 49, 4196–4212.

Zhang, L., Wang, J., Wang, W., et al., 2022a. Smart contract vulnerability detection combined with multi-objective detection. Comput. Netw. 217, 109289.

Zhang, F., Yu, X., Keung, J., et al., 2022b. Improving stack overflow question title generation with copying enhanced codebert model and bi-modal information. Inf. Softw. Technol. 148, 106922.

Zhuang, Y., Liu, Z., Qian, P., et al., 2020. Smart contract vulnerability detection using graph neural network. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence. pp. 3283–3290.

Jianxin Cheng received the B.S. degree from the School of Computer, Hubei University, Wuhan, China, in 2018, and the master's degree in the School of Computer at Wuhan University, Wuhan, China, in 2021. Now, he is pursuing the Ph.D. degree with the School of Computer Science, Peking University, China. His research interests include vulnerability detection, code representation, and software engineering.

Yizhou Chen received the master's degree in the School of Computer at Wuhan University, Wuhan, China, in 2022. Now, he is pursuing the Ph.D. degree with the School of Computer Science, Peking University, China. His research interests include vulnerability detection, code representation, and software engineering.

Yongzhi Cao received the B.S. and M.S. degrees from Central China Normal University, Wuhan, China, in 1997 and 2000, respectively, and the Ph.D. degree from Beijing

Normal University, Beijing, China, in 2003, all in mathematics. He is currently a Professor of Computer Science with Peking University, Beijing. From 2003 to 2007, he was a Postdoctoral Researcher with Tsinghua University, Beijing, and from 2007 to 2015, he was an Associate Professor of Computer Science with Peking University, Beijing. He has published more than 80 papers in academic journals and conferences such as AAAI, IJCAI, Artif. Intell., ACM Trans. Embed. Comput. Syst., IEEE Trans. Autom. Contr., IEEE Trans. Comput., IEEE Trans. Fuzzy Syst., IEEE Trans. Syst. Man Cybern., Part B: Cybern., Inform. Comput., J. Comput. Syst. Sci., and Theor. Comput. Sci. His current research interests include formal methods, privacy and security, and reasoning about uncertainty in artificial intelligence. He is a senior member of IEEE.

Hanpin Wang is a Professor of Computer Science. He obtained his Ph.D. in Mathematics from Beijing Normal University in 1993. His research interests include formal semantics and verification of computer systems, especially cloud storage systems. He has also special research interests in algorithms and computational complexity, including dichotomy and approximation of counting problems. Dr. Wang has published more than 90 research papers, and some of them are published in important conferences and journals, such as ICALP, IFIP, TCS, DBLP and Information and Computation.