Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention[☆]

Da Chen[a], Lin Feng[a], Yuqi Fan[a,b,*], Siyuan Shang[a], Zhenchun Wei[a]

[a] School of Computer Science and Information Engineering, Anhui Province Key Laboratory of Industry Safety and Emergency Technology, Hefei University of Technology, Hefei 230009, China
[b] Anhui Provincial Key Laboratory of Network and Information Security, Anhui Normal University, Wuhu 241002, China

ARTICLE INFO

Article history:
Received 3 August 2022
Received in revised form 5 February 2023
Accepted 10 April 2023
Available online 17 April 2023

Keywords:
Smart contract vulnerability detection
Code graph
Graph convolutional networks
Edge attention
Residual block

ABSTRACT

Smart contracts are becoming the forefront of blockchain technology, allowing the performance of credible transactions without third parties. However, smart contracts on blockchain are not immune to vulnerability exploitation and cannot be modified after being deployed on the blockchain. Therefore, it is imperative to assure the security of smart contracts via intelligent vulnerability detection tools with the exponential increase in the number of smart contracts. The remarkably developing deep learning technology provides a promising way to detect potential smart contract vulnerabilities. Nevertheless, existing deep learning-based approaches fail to effectively capture the rich syntax and semantic information embedded in smart contracts for vulnerability detection. In this paper, we tackle the problem of smart contract vulnerability detection at the function level by constructing a novel semantic graph (SG) for each function and learning the SGs using graph convolutional networks (GCNs) with residual blocks and edge attention. Our proposed method consists of three stages. In the first stage, we create the SG which contains rich syntax and semantic information including the data–data, instruction–instruction and instruction–data relationships, variables, operations, etc., by building an abstract syntax tree (AST) from the code of each function, removing the unimportant nodes in the AST, and adding edges between the nodes to represent the data flows and the execution sequence of the statements. In the second stage, we propose a new graph convolutional network model EA-RGCN to learn the content and semantic features of the code. EA-RGCN contains three parts: node and edge representation via word2vec, content feature extraction with a residual GCN (RGCN) module, and semantic feature extraction using an edge attention (EA) module. In the third stage, we concatenate the code content features and the semantic features to obtain the global code feature and use a classifier to identify whether the function is vulnerable. We conduct experiments on the datasets constructed from real-world smart contracts. Experimental results demonstrate that the proposed semantic graph and the EA-RGCN model can effectively improve the performance in terms of accuracy, precision, recall, and F1-score on smart contract vulnerability detection.

© 2023 Elsevier Inc. All rights reserved.

1. Introduction

Blockchain has been widely applied in various domains such as cryptocurrencies (Nakamoto, 2008; Wood, 2014), financial industry, insurance, Internet of Things (IoT), healthcare, governance, etc. A blockchain is a distributed ledger managed by all the nodes in the blockchain network (Dinh et al., 2018). The miners follow a consensus protocol (Sankar et al., 2017) to protect the network against malicious or dishonest nodes.

Smart contract vulnerability. Smart contracts are running on blockchains and benefit from blockchains' decentralized and immutable characteristics. However, the immutability is a double-edged sword for smart contracts. Once deployed on the blockchain, a smart contract cannot be modified, such that the smart contract with vulnerabilities may be compromised by hackers, causing kinds of losses. For example, The DAO's code had vulnerabilities, and there was a bug in the smart contracts of the DAOs wallet which the hackers took advantage of, which caused the loss of about 3.6 million Ether or $70 million USD worth of Ether. The attack created a crisis of confidence between users and smart contracts. With the exponential increase in the

☆ Editor: Burak Turhan.
* Corresponding author at: School of Computer Science and Information Engineering, Anhui Province Key Laboratory of Industry Safety and Emergency Technology, Hefei University of Technology, Hefei 230009, China.
E-mail addresses: da.chen@mail.hfut.edu.cn (D. Chen), fenglin@hfut.edu.cn (L. Feng), yuqi.fan@hfut.edu.cn (Y. Fan), siyuan0102@mail.hfut.edu.cn (S. Shang), weizc@hfut.edu.cn (Z. Wei).

https://doi.org/10.1016/j.jss.2023.111705
0164-1212/© 2023 Elsevier Inc. All rights reserved.

number of smart contracts, it is imperative to employ intelligent vulnerability detection tools to assure the security of smart contracts.

Conventional smart contract vulnerability detection. Smart contract vulnerability detection based on conventional methods exploit symbolic execution and formal verification, such as Smartcheck (Tikhomirov et al., 2018), Oyente (Luu et al., 2016), Securify (Grishchenko et al., 2018), Slither (Feist et al., 2019), etc. However, in order to analyze smart contracts, domain experts need to define application-specific logic rules, which depend heavily on the experience, professionality, and understanding of domain knowledge. Therefore, the conventional vulnerability detection methods consume significant expert time, and the predefined logic rules are also prone to errors.

Deep learning based smart contract vulnerability detection. The remarkably developing deep learning technology provides a promising way to study smart contract vulnerabilities intelligently. Existing deep learning-based approaches characterize the codes as a sequence, a tree, or a graph. When the code is portrayed as a sequence, some deep learning models that can process sequences are utilized to detect smart contract vulnerabilities, and the model examples include N-Gram Language Modeling (Liu et al., 2018), Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997), Bidirectional LSTM (Bi-LSTM) (Tian et al., 2020), Bi-LSTM with Attention Mechanism (BLSTM-ATT) (Qian et al., 2020), Transformer (Vaswani et al., 2017), etc. Sequence-based code representation describes the statement execution order but cannot characterize the flow of data, the invocation relationships between instructions, and the detailed execution structure of the statements.

When the code is represented as a tree such as abstract syntax tree (AST), K-Nearest-Neighbor (KNN) model can be used to detect smart contract vulnerabilities (Xu et al., 2021). Tree-based code representation can depict the syntax information of the code but still fails to characterize the execution order of statements explicitly and lacks the information of data flow.

Graph-based code vulnerability detection methods in the domains other than smart contracts achieves favorable performance by building a graph to represent the relationship between the codes, such as control flow graph (CFG) (Yan et al., 2019) and code property graph (Yamaguchi et al., 2014). However, these graphs miss either data flows or control flows.

There is some research on smart contract code vulnerability detection based on graphs. Zhuang et al. (2020) constructed a normalized contract graph which includes the data flow and control flow and proposed temporal message propagation network (TMP) to process the graph. Liu et al. proposed CGE and AME. CGE (Liu et al., 2021b) combines the contract graph of Zhuang et al. (2020) and expert knowledge. AME (Liu et al., 2021a) uses attention to process the expert knowledge introduced by CGE. Wu et al. (2021) proposed Peculiar to use the crucial data flow for smart contract reentrancy detection. Graph-based code representation can better capture the association among statements and data. However, the existing smart contract graphs may miss some information in the code. For example, TMP, CGE, and AME may ignore some syntax information; Peculiar overlooks the instructions in the code, while the smart contracts are run via the instructions.

When processing the smart contracts via graphs, existing graph neural networks mainly learn the features of the nodes, while ignoring the global structure features of the graph. Note that the graph structure indicates the semantic information in the code, including data–data, instruction–instruction and instruction–data relationships. Furthermore, large training loss variation often occurs during the training of graph neural networks (GNNs).

Our method. Based on the analysis of the vulnerabilities, we observe that a risk comes from the incorrect code of a function. In this paper, we tackle the problem of smart contract vulnerability detection at the function level by constructing a novel semantic graph (SG) for each function and learning the SG using a graph convolutional network (GCN) with residual blocks and edge attention. We construct the SG based on the AST of a smart contract function, and the SG contains rich syntax and semantic information including the data–data, instruction–instruction and instruction–data relationships, variables, operations, etc. Specifically, we simplify the AST by deleting the useless nodes such as parameter list and the storage locations of variables, which cannot lead to vulnerabilities. We then add edges in the simplified AST to represent statement execution sequences and data flows. Different from the contract graph (Zhuang et al., 2020; Liu et al., 2021b,a), we build the SG based on the AST, which enables the SG to retain the rich syntax information. Unlike the crucial data flow (Wu et al., 2021), our SG contains control flows; meanwhile, our SG deletes some unnecessary nodes and edges.

For learning the SG, we propose a new graph convolutional network model (EA-RGCN) to learn the content and semantic features of the code. EA-RGCN contains three parts: node and edge representation via word2vec (Mikolov et al., 2013), content feature extraction with a residual GCN (RGCN) module, and semantic feature extraction using an edge attention (EA) module. We represent each SG node as a vector using word2vec and describe each SG edge as the average feature of the two adjacent nodes. In the RGCN, we add a residual connection between the outputs of two adjacent graph convolutional layers to learn the code content features, which characterizes the syntax and weak semantic information. In the EA, we perform multi-head attention on the edge sequence containing all the SG edges in the breadth first order to learn the semantic features in the code. Finally, we concatenate the code content features and the semantic features to obtain the global code feature and use a classifier to identify whether the function is vulnerable. Different from TMP (Zhuang et al., 2020), CGE (Liu et al., 2021b), and AME (Liu et al., 2021a), our EA-RGCN introduces residual block in GCN for easier and better optimization; furthermore, we propose edge attention to effectively capture the semantic features. Unlike Peculiar (Wu et al., 2021), our EA-RGCN can process the edges in the graph, which indicates the semantic information in the code, including data–data, instruction–instruction and instruction–data relationships.

In the experiments, different from TMP, CGE, and AME focusing on reentrancy, timestamp dependency, etc. and Peculiar targeting reentrancy, we investigate the performance of our method on four common smart contract vulnerabilities, i.e., arithmetic, reentrancy, timestamp dependency, and unchecked low calls.

Contributions. In summary, the key contributions of our work are:

- We propose a new way to build the code graph of a smart contract function, where the graph is called as an SG. Starting from the simplified smart contract function AST, we remove the useless nodes and then add edges to represent statement execution sequences and data flows.
- We propose a novel model EA-RGCN to learn the content and semantic features of the code from the SG for smart contract vulnerability detection at the function level. EA-RGCN extracts the code content features using RGCN, which adds residual blocks between two adjacent graph convolutional layers to avoid large fluctuation of training loss. EA-RGCN also extracts the semantic features by performing multi-head attention on the edge sequence in the breadth first order. The content and semantic features are concatenated as the global code feature, which is fed into a classifier for vulnerability identification.

- We conduct experiments on four common smart contract vulnerabilities, i.e., arithmetic, reentrancy, timestamp dependency, and unchecked low calls, using the datasets constructed from SmartBugs (Durieux et al., 2020). The experimental results demonstrate that the proposed SG and the EA-RGCN model can effectively improve the smart contract vulnerability detection performance in terms of accuracy, precision, recall, and F1-score on smart contract vulnerability detection tasks.

2. Related work

2.1. Conventional smart contract vulnerability detection

Smart contract vulnerability detection based on conventional methods exploit symbolic execution, formal verification, etc. Tikhomirov et al. (2018) proposed Smartcheck, a static analysis tool to detect the smart contract vulnerability. Smartcheck translates the smart contract source code into an XML-based intermediate representation and checks it against XPath patterns. Luu et al. (2016) proposed Oyente for smart contract vulnerability detection. Oyente constructs the CFG for a smart contract, and runs the contract symbolically to detect whether the contract is vulnerable. Grishchenko et al. (2018) proposed Securify, a framework to analyze and verify the functional correctness of smart contracts. Securify transfers the smart contract source code to F*, which is a functional programming language, to verify programs. Feist et al. (2019) proposed Slither, which is a static analysis framework to identify smart contract vulnerabilities. Slither converts the smart contract source code into an intermediate representation consisting of reduced instruction set and the lost semantic information. The intermediate representation of the smart contract is then used to analyze the potential vulnerabilities in the smart contract.

In order to analyze the smart contracts, domain experts need to define application-specific logic rules, which depends heavily on the experience, professionality, and understanding of domain knowledge. Therefore, the conventional vulnerability detection methods consume significant expert time, and the predefined logic rules are prone to errors as well.

2.2. Deep learning based smart contract vulnerability detection

Along with the notable development of deep learning, it is introduced in the smart contract vulnerability detection tasks. In general, deep learning-based smart contract vulnerability detection can be divided into three categories: sequence-based, tree-based, and graph-based methods.

2.2.1. Sequence-based methods

Sequence-based methods take smart contract vulnerability detection tasks as the common natural language processing problem and represent the source code as sequences. Tian et al. (2020) studied smart contract vulnerability detection by dividing a smart contract into three parts: source code, comment, and account related information, which are treated as word sequences. Word embedding is applied on these word sequences to generate smart contract features which are then fed into a Bi-LSTM for deciding whether the smart contract is vulnerable. Qian et al. (2020) treated smart contracts as contract snippets, where variable and function names are mapped to symbolic names. After that, each contract snippet in the symbolic representation is divided into a sequence of tokens via the lexical analysis. BLSTM-ATT is used to process the sequences of the smart contract to identify the smart contract vulnerability.

Sequence-based methods describe the statement execution order, but cannot characterize the data flows, the calling relationships between instructions, and the detailed execution logic of the statements.

2.2.2. Tree-based methods

Tree-based methods use the tree structure to represent the code and apply static analyses or machine learning to identify the smart contract vulnerability. Ma et al. (2019) proposed a method Rejection to detect smart contract vulnerabilities, which is based on the smart contract AST. Through traversing the nodes of the AST, Rejection compares vulnerability related notations with the predefined rules to determine whether the smart contract is vulnerable. Xu et al. (2021) constructed a feature vector composed of the values that measure the structural similarity between the ASTs of smart contracts with some vulnerabilities, and used a KNN model to decide whether the smart contract is vulnerable.

Tree-based code representation can depict the syntax information of the code, but fails to characterize the order of statements obviously and lacks the information of data flows.

2.2.3. Graph-based methods

Graph-based methods take a graph as the smart contract representation and apply the graph neural network (GNN) to learn the differences between graphs. Some researchers conduct code vulnerability detection in the domains other than smart contracts based on GNN. Cao et al. (2021) used a bidirectional graph neural network (BGNN) on the code composite graph for code vulnerability detection. BGNN achieves bidirectional message propagation between adjacent graph nodes by adding each forward edge a backward edge. Wang et al. (2021) applied a gated graph neural network (GGNN) with a multi-relational graph to detect code vulnerabilities. The GGNN can learn the graph through aggregating and updating the states for the same nodes across graphs. Yan et al. (2019) used DGCNN to embed information inherent in CFGs for malware detection.

There is some research on smart contract code vulnerability detection based on graphs. Zhuang et al. (2020) constructed a normalized contract graph for a smart contract to simplify the CFG of the smart contract by preserving specific vulnerability related information. The normalized contract graph is then fed into a TMP to learn the graph. Liu et al. (2021b,a) combined normalized contract graph with the expert pattern to concentrate on specific smart contract vulnerability detection tasks, where the expert pattern is a kind of expert-defined rules. Wu et al. (2021) proposed Peculiar to detect smart contract reentrancy vulnerability. Peculiar uses crucial DFG as the representation of smart contracts. Specifically, Peculiar simplifies the DFG by deleting useless data flows, which cannot lead to reentrancy. The smart contract code sequence and the crucial DFG are then fed into a graph code BERT (Guo et al., 2021) to train a pre-trained model which has the ability to identify the reentrancy vulnerability.

In general, the CFG ignores the data flows in the code, while the data error is a vital result of vulnerabilities; the DFG overlooks the instruction in the code, while the smart contracts are run via the instructions. Some composite graph-based code representations used in the domains other than smart contracts include too much information which cannot lead to vulnerabilities but may mislead the training of neural networks. As for the graph processing, existing GNNs mainly learn the features of the nodes while ignoring the global structure features of the graph. Note that the graph structure indicates the semantic information in the code, including data–data, instruction–instruction and instruction–data relationships.

The existing research demonstrates that the graph-based code representation can better capture the association among statements and data. Therefore, we characterize the smart contract code using the graph structure, and propose our smart contract representation called SG. In addition, we learn the SGs with a novel model EA-RGCN for smart contract vulnerability detection.

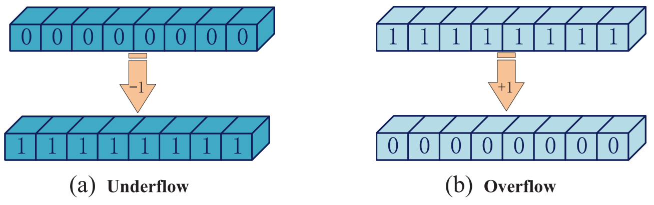


Fig. 1. Arithmetic error of an 8-bit unsigned integer.

3. Problem description

In this paper, we investigate four important smart contract vulnerabilities, i.e., arithmetic, reentrancy, timestamp dependency, and unchecked low calls, and study how to detect them at the function level based on deep learning. That is, for a given smart contract, we develop an automatic deep learning-based model φ to identify if each function in the smart contract is vulnerable or not. We represent the code of each function as a graph g. The label of a function is denoted as $\hat{y}$ and $\hat{y} = \varphi(g)$, where $\hat{y} = 1$ and $\hat{y} = 0$ indicate the function is vulnerable and clean, respectively.

Arithmetic. Arithmetic vulnerability is a kind of common defect in smart contracts and occurs in two cases. One is overflow, where we assign such a value to a variable that is more than the maximum permissible value. The other is underflow, where we assign such a value to a variable that is less than the minimum permissible value. Fig. 1 shows two examples when underflow and overflow occur in an 8-bit unsigned integer. In Fig. 1(a), an 8-bit unsigned integer variable is 0. If we minus 1 to this integer, the result will be 255. That is, underflow happens. The unsigned integer overflow example in Fig. 1(b) is similar to that in Fig. 1(a). Fig. 2 shows a real smart contract function with the arithmetic vulnerability. In this smart contract, statement withdraw(uint) (Line 7) dictates that the balance of sender is large than _amount with a subtraction. If _amount is larger than sender's balance (Line 8), underflow occurs, which results in an arithmetic error.

Reentrancy. Reentrancy is a well-known vulnerability which caused the famous DAO attack. An Ethereum smart contract can call the functions of another smart contract, and Ether is transferred through payment functions such as call.value(), send(), transfer(), and so on. Fig. 3 shows an example of a smart contract with the reentrancy vulnerability. Smart contract Attacker executes its attack function, which calls smart contract Victim's function withdraw to transfer the amount of Victim's Ether to Attacker. Note that the withdraw function in Victim uses the method call.value() (Line 7 on the right) to transfer Ether, which will call Attacker's payable function (Line 7 on the left). However, the payable function body is another call for withdrawing Victim's Ether, which leads to a loop of transferring Victim's Ether to Attacker until Victim's balance drops to 0.

Timestamp dependency. Timestamp dependency is another well-known vulnerability which involves the use of block timestamp statements. Once the block timestamp is taken as a dependency condition for critical operations like Ether transfer, some malicious miners may manipulate timestamp for illegal benefits. Fig. 4 shows an example of a smart contract with the timestamp dependency vulnerability. In smart contract MyToken shown in Fig. 4, the variable h in Line 9 defines the hash value of a certain block in the blockchain, determining which miner gets the bonus finally for generating a new block, while the variable seed in Line 8 decides the choice. The seed is calculated by three parts, i.e., block number, last payout, and timestamp. The block number and the last payout are decided by the values recorded on the blockchain, while the timestamp is determined by the miner himself. Generally, the timestamp should be set the same as the current time of the miner's local system. However, the miner can change the timestamp value by roughly 900 s and this operation does not violate the consensus protocol (Jiang et al., 2018). The miner can calculate a seed value which makes him get the final block-generation bonus by manipulating the timestamp value.

Unchecked low calls. It is a kind of vulnerability that happens when the return value of a function call is not checked strictly. Underlying function calls (call(), delegatecall(), callcode(), etc.) and transfer functions (call.value(), send(), transfer(), etc.) are those with return values needing to be checked strictly. Note that the execution of these functions costs gas, which is a kind of reward for the miner. When the gas is not enough to pay for the miner, the execution of the smart contract will be revoked and the false value will be returned. Fig. 5 shows an example of a smart contract with the unchecked low calls vulnerability. The function call in Line 8 of smart contract MyToken uses send() to transfer Ether. If the gas of smart contract MyToken is not enough, MyToken is not aware that the return value is false and then the financial loss incurs, since the return value is not checked.

It can be observed that a risk comes from the incorrect code of a function. On the one hand, it is obvious that the sources of some vulnerabilities lie in the code of the functions, such as arithmetic, timestamp dependency, and unchecked low calls. On the other hand, the vulnerabilities involving inter-functional relationships are also due to the incorrect code of some functions. Take the contract Victim with reentrancy vulnerability shown in the right side of Fig. 3 as an example. The reentrancy vulnerability is caused by the wrong execution sequence of Ether transfer and variable assignment operations shown in Lines 7 and 8. This statement sequence enables Attacker to make Victim stuck in executing the Ether transfer operation in Line 7 without going on to balance update operation in Line 8 and then exiting function withdraw. Consequently, Attacker can steal Ether from Victim. Contract Victim is free from Reentrancy, if we reverse the statement sequence of Lines 7 and 8 from the left side to the right side of Fig. 6. By using the updated statement sequence, the balance of Attacker is reduced and then the Ether transfer operation is performed, when executing function withdraw of contract Victim. In this way, each call of function withdraw reduces the corresponding amount of Ether from the balance of Attacker, and hence the attacker cannot steal the digital currency from Victim. Based on the analysis of the vulnerabilities, we find it is feasible to detect the smart contract vulnerabilities at the function level.

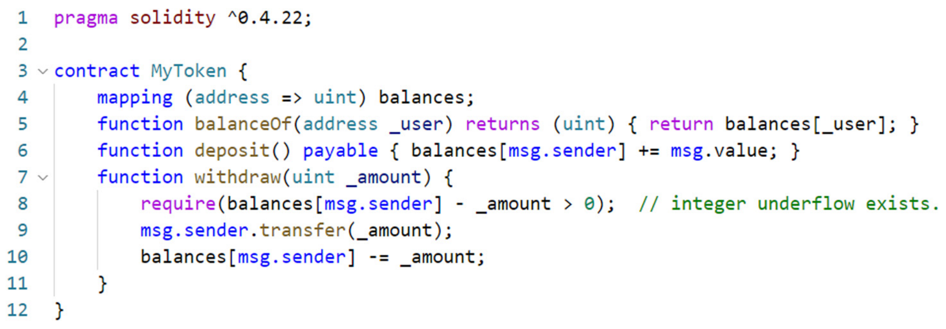
4. Method

In this section, we construct a novel SG as the representation of each smart contract function and propose a graph convolutional network model (EA-RGCN) to learn the SGs. As shown in Fig. 7, our method consists of three stages.

Stage 1 Create an SG for each smart contract function.
Stage 2 Propose model EA-RGCN to learn the content and semantic features of the code from the SG.
Stage 3 Validate the security of the smart contract function using a classifier with the concatenated code content and semantic features as the input.

4.1. Semantic Graph (SG)

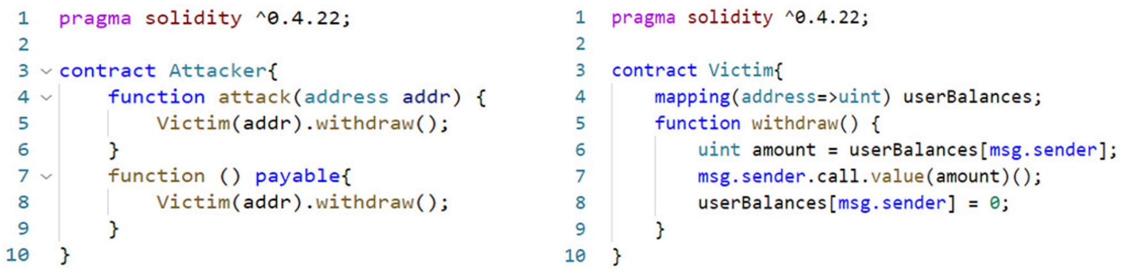
An AST contains the syntax information of the source code. However, the AST lacks the information like data flows and statement execution sequences. Furthermore, the AST includes some



```
pragma solidity ^0.4.22;

contract MyToken {
    mapping (address => uint) balances;
    function balanceOf(address _user) returns (uint) { return balances[_user]; }
    function deposit() payable { balances[msg.sender] += msg.value; }
    function withdraw(uint _amount) {
        require(balances[msg.sender] - _amount > 0);  // integer underflow exists.
        msg.sender.transfer(_amount);
        balances[msg.sender] -= _amount;
    }
}
```

Fig. 2. Example of a smart contract with the arithmetic vulnerability.



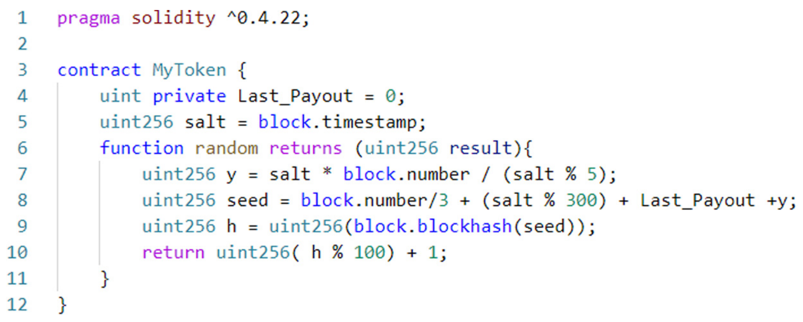
```
pragma solidity ^0.4.22;

contract Attacker{
    function attack(address addr) {
        Victim(addr).withdraw();
    }
    function () payable{
        Victim(addr).withdraw();
    }
}
```

```
pragma solidity ^0.4.22;

contract Victim{
    mapping(address=>uint) userBalances;
    function withdraw() {
        uint amount = userBalances[msg.sender];
        msg.sender.call.value(amount)();
        userBalances[msg.sender] = 0;
    }
}
```

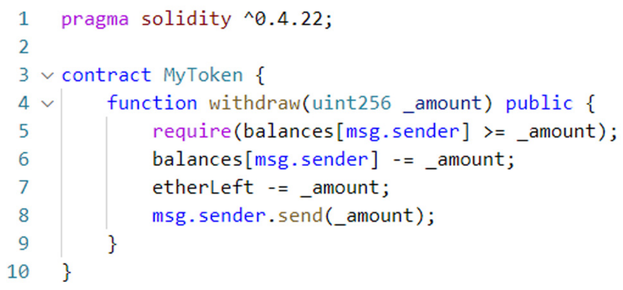
Fig. 3. Example of a smart contract with the reentrancy vulnerability.



```
pragma solidity ^0.4.22;

contract MyToken {
    uint private Last_Payout = 0;
    uint256 salt = block.timestamp;
    function random returns (uint256 result){
        uint256 y = salt * block.number / (salt % 5);
        uint256 seed = block.number/3 + (salt % 300) + Last_Payout +y;
        uint256 h = uint256(block.blockhash(seed));
        return uint256( h % 100) + 1;
    }
}
```

Fig. 4. Example of a smart contract with the timestamp dependency vulnerability.



```
pragma solidity ^0.4.22;

contract MyToken {
    function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount);
        balances[msg.sender] -= _amount;
        etherLeft -= _amount;
        msg.sender.send(_amount);
    }
}
```

Fig. 5. Example of a smart contract with the unchecked low calls vulnerability.

information which cannot lead to vulnerabilities but may mislead the training of neural networks. Therefore, we create the SG based on the AST by deleting the useless nodes to simplify the AST and adding edges to represent data and control flows.

The generation of an SG includes two steps. The first step is to build the AST of each smart contract function, and the second step is to obtain the SG based on the AST.

The AST of a smart contract. An AST characterizes the syntax information of the smart contract code, including (1) data, such as parameters and their attributes; (2) operations, such as statements, operators, and extended libraries provided by the programming language. We generate the AST of a smart contract function through the state-of-the-art AST builder: Python-Solidity-Parser (2021). For a given smart contract function in Fig. 8, the corresponding AST is shown in Fig. 9. Specifically, the AST root is "FuncDef", which indicates the beginning of a function. The first child node (param list in Fig. 9) of the root represents the parameter list of the function, such as tad and atk in Fig. 7 (Line 4). The second child node (transfer in Fig. 9) of the root is the function name. The third child node (body in Fig. 9) of the root indicates the start of the function body, which includes the statements in the function, i.e., param, funccall, arithmetic operations, and control instructions. Param indicates the subtree is a variable or constant declaration, such as tk shown in Fig. 8 (Line 10). Funccall represents that the subtree includes function call related statements, such as SafeMath.div shown in Fig. 8 (Line 10). The control instructions are related to control flows, such as if, for, while, do-while, and so on. For example, the ifstatement in Fig. 9 corresponds to the if in Fig. 8 (Line 7). Each node in the AST may have its own child nodes. For example, Line 6 in Fig. 8 uses a function named require to determine whether atk is less than or equal to balance. Therefore, Line 6 is a function call statement and we use funccall to indicate this statement. The child node

```
msg.sender.call.value(amount)();     userBalances[msg.sender] = 0;
userBalances[msg.sender] = 0;        msg.sender.call.value(amount)();
```

Fig. 6. Inappropriate (left) and correct (right) statement sequences of contract Victim in Fig. 3.

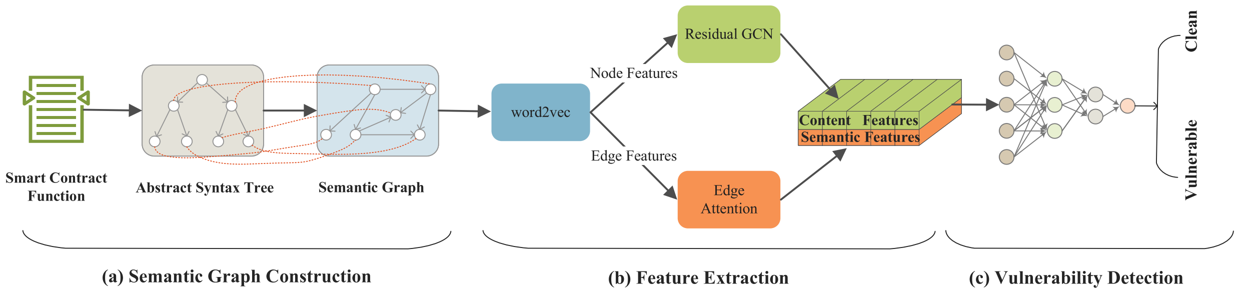


Fig. 7. The overall architecture of our proposed method.

```
pragma solidity ^0.4.20;

contract func {
    function transfer(address tad, uint256 atk){
            address cad = msg.sender;
            require(atk <= balance);
            if (mDi(true) > 0) {
                withdraw();
            }
            uint256 tk = SafeMath.div(SafeMath.mul(atk, tf), 100);
            uint256 tax = SafeMath.sub(atk, tk);
            Transfer(cad, tad, tax);
            return  true;
        }
}
```

Fig. 8. An example of a smart contract function.

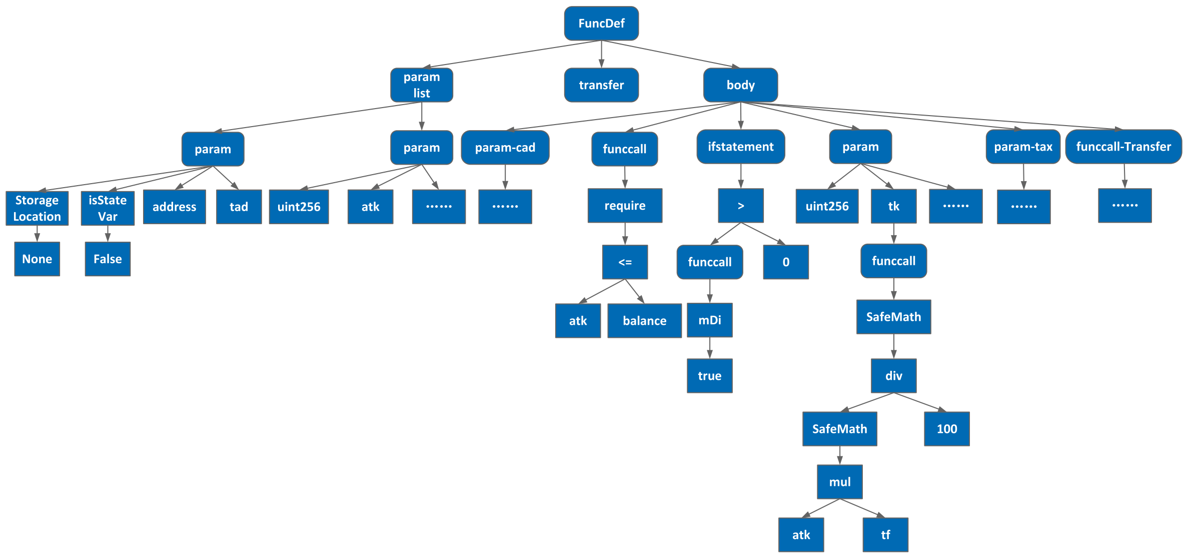


Fig. 9. The AST of the given smart contract function shown in Fig. 8.

of the funccall is the symbol of arithmetic operator "$<=$", which children are operands atk and balance.

SG generation algorithm. Algorithm SG-Generation shown in Algorithm 1 simplifies the smart contract AST by deleting the useless nodes, and then constructs the SG based on the simplified AST by adding edges between nodes to represent data flows and control flows in the smart contract code. Specifically, the input of algorithm SG-Generation is an AST which includes statements, operators, extended libraries, etc. The output of algorithm SG-Generation is the constructed SG $= \left(V, E'\right)$, which includes node set V and edge set E'. Algorithm SG-Generation includes 3 phases.

Phase 1: We simplify the AST by deleting the useless nodes including the param list and the variable attributes except the name and type of the variable (Line 1, Algorithm 1). SG node set V is initialized as the AST node set. We then add a new node *Enter*

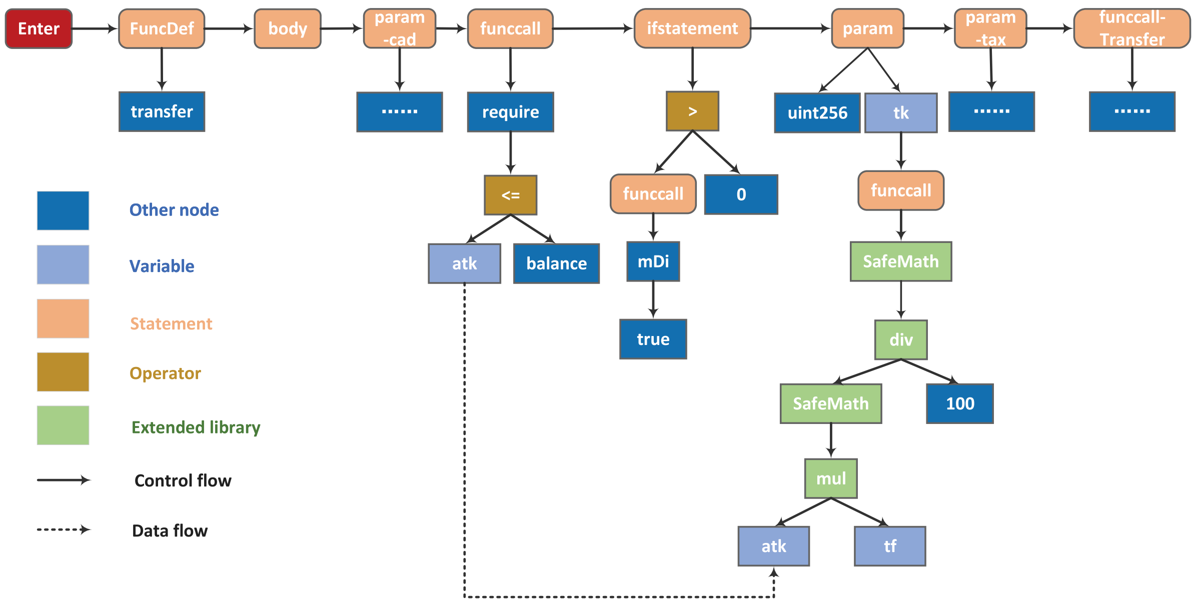


Fig. 10. The SG of the given contract smart function shown in Fig. 8.

Algorithm 1: SG-Generation

```
  Input: AST
  Output: SG= $(V, E')$
1 Delete the parameter list and the variable attributes
    except variable name and type from AST;
2 Initialize $V$ as the node set of the simplified AST;
3 Initialize $E'$ as the edge set of the simplified AST;
4 $V \leftarrow V+Enter$;
5 $E' \leftarrow E' + <Enter$, AST.root$>$;
6 ControlFlowEdgeGeneration(simplified AST, $E'$');
7 DataFlowEdgeGeneration(simplified AST, $E'$');
8 return $V, E'$.
```

Algorithm 2: ControlFlowEdgeGeneration

```
   Input: $tree, E'$
1  if $tree$.root has children then
2      $cn \leftarrow tree$.root.firstchild;
3      while $cn$.nextsibling is not null do
4          if $cn$ is a statement then
5              $E' \leftarrow E' + < cn, cn.nextsibling >$;
6              ControlFlowEdgeGeneration(the subtree of tree
                 with $cn$ being the root, $E'$');
7          end
8          $cn \leftarrow cn.nextsibling$;
9      end
10 end
```

Algorithm 3: DataFlowEdgeGeneration

```
   Input: $tree, E'$
1  Sort the $tree$ nodes in the depth first search order, and put
     them in a list $Ln$;
2  for $node$ in $Ln$ do
3      if $node$ is a variable then
4          while $node$.next is not null do
5              $node' \leftarrow node$.next;
6              if $node == node'$ then
7                  $E' \leftarrow E' + < node, node' >$;
8              end
9          end
10     end
11 end
```

in node set V to represent the start of the SG (Line 4, Algorithm 1). We set E' as the edge set of the simplified AST, and add the directed edge $< Enter$, AST.root$>$ into E' (Line 5, Algorithm 1).

Phase 2: We construct the control flows via algorithm ControlFlowEdgeGeneration shown in Algorithm 2. The input of algorithm ControlFlowEdgeGeneration is the simplified AST or its subtree and edge set E' which will be updated by the algorithm. During the construction of control flows, we traverse the tree in the depth first search (DFS) order. For a node cn, we check whether cn is a statement (Line 4, Algorithm 2). If yes, we add an arc from cn to its next sibling into edge set E' to represent the execution sequence of statements (Line 5, Algorithm 2), and then run algorithm ControlFlowEdgeGeneration recursively with the subtree having cn as the root (Line 6, Algorithm 2). In this way, the SG can characterize the statement execution sequence.

Phase 3: We construct the data flows via algorithm DataFlowEdgeGeneration shown in Algorithm 3. The input of algorithm DataFlowEdgeGeneration is a tree and edge set E' which will be updated by the algorithm. We first sort the tree nodes in the depth first search order, and put them in a list Ln. We then traverse Ln to find the same variables, and add edges between them to represent the data flows in the smart contract code.

After these three phases, algorithm SG-Generation gets SG $=(V, E')$. Fig. 10 shows the SG of the given example in Fig. 8. The nodes are divided into statements, operators, extended libraries, variables, and the other nodes. The edges indicate control flows and data flows, which are shown by solid and dotted lines, respectively. In this way, the constructed SG can not only contain the syntax information of the smart contract code, but also characterize the rich semantic information, including the data flows, statement execution order, and instruction–data relationships. For example, the upmost part of the SG shown in Fig. 10 is built by algorithm ControlFlowEdgeGeneration from Fig. 9 and

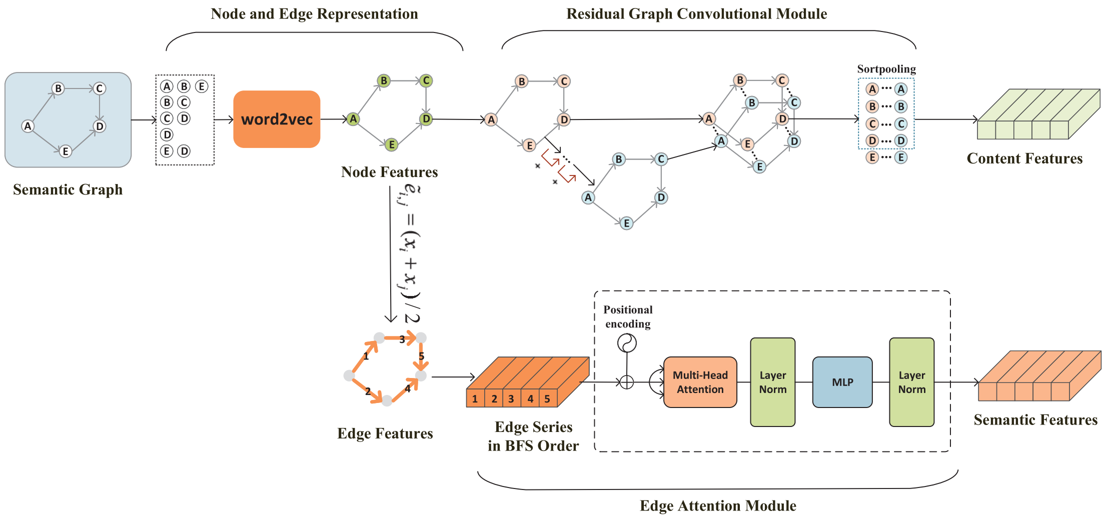


Fig. 11. EA-RGCN model.

describes the control flow characterizing the order of statement execution. The dotted arc between two atk nodes created by algorithm DataFlowEdgeGeneration from Fig. 9 depicts the data flow of variable atk.

4.2. Model EA-RGCN

We propose a novel graph convolutional network model EA-RGCN to learn the code content features and semantic features. EA-RGCN includes three parts: node and edge feature representation using word2vec (Mikolov et al., 2013), code content feature learning using the RGCN module, and semantic feature learning with the EA module. The smart contract code features mainly consist of the local code content features and global semantic features. Through word2vec and RGCN modules, we learn the code content features from the semantic graph. Through EA module, we learn the semantic features from the semantic graph. Specifically, we represent the feature of each SG node as a vector using word2vec and describe the feature of each SG edge as the average feature of the two adjacent nodes. The node representation generated by word2vec includes part of the topology information of the graph, which contains partial semantic information. RGCN learns the code content features from the semantic graph, which characterize the syntax and some of the semantic information. EA extracts the semantic features from the edges by capturing the graph structure features, which indicate the semantic information in the code, including data–data, instruction–instruction and instruction–data relationships. The overall architecture of EA-RGCN is shown in Fig. 11.

4.2.1. Node and edge feature representation

We perform word2vec to represent the feature of each SG node. Specifically, for each node, we obtain its adjacent node list, which is fed into a pre-trained word2vec with skip-gram. The output of word2vec is the node representation matrix $X \in R^{N\times C}$, where N is the number of SG nodes and C is the dimension of node representation. word2vec learns continuous feature representation for nodes in the graph and preserving the knowledge gained from the neighboring nodes. The node representation generated by word2vec includes part of the topology information of the graph, which contains partial semantic information.

We start to traverse the SG starting from node *Enter* in the breadth first search (BFS) order to obtain the edge series $\mathcal{E}$. For each edge connecting two adjacent nodes i and j, the edge representation is the average of the two nodes' representations, i.e.,

$$\tilde{e}_{i,j} = \left(x_i + x_j\right)/2 \tag{1}$$

where x_i is the representation of node i and $\tilde{e}_{i,j}$ is the representation of edge $\langle i, j\rangle$. We then construct representation matrix $\tilde{E} \in R^{E\times C}$ for edge series $\mathcal{E}$, where E is the number of edges in $\mathcal{E}$.

The different types of edges are implicitly reflected by the edge features, because different types of edges connect distinct types of nodes and different types of nodes have nonidentical features. As can be seen from Fig. 10, the nodes are divided into 5 categories, i.e., statements, operators, extended libraries, variables, and the other nodes. Different types of nodes have different features after the feature representation via word2vec. The edge feature is the average feature of the two adjacent nodes. Consequently, different types of edges are naturally distinct. In this way, the impacts of different types of edges can be accumulated during further feature extraction.

4.2.2. Residual graph convolutional module

In the RGCN module, we extract the feature of each SG node, such that we can learn the code content features, which characterize the syntax and some of the semantic information. We obtain the adjacent matrix A of the SG$=(V, E')$ of a smart contract function, which is a symmetric 0/1 matrix. We then define the augmented adjacency matrix $\tilde{A}$ as $\tilde{A} = A + I$ to represent the adjacent matrix of the SG with self-loops, where $I \in R^{N\times N}$ denotes the unit matrix and $\tilde{A} \in R^{N\times N}$. $\tilde{D} \in R^{N\times N}$ is the diagonal degree matrix and can be calculated as:

$$\tilde{D} = \sum_{1\le u\le N, v=1}^{N} \tilde{A}_{u,v} \tag{2}$$

where $\tilde{A}_{u,v}$ is the element (u, v) in $\tilde{A}$.

We use GCN to process the SG and define each graph convolutional layer as follows:

$$Z = f\left(\tilde{D}^{-1}\tilde{A}XW\right) \tag{3}$$

where $W \in R^{C\times K}$ is the weight matrix of the GCN model and f denotes the activation function. To avoid the large fluctuation of training loss when training GCN, we add residual connection between the outputs of two adjacent GCN layers. Specifically, the output of the tth layer is added to the output of the $(t+1)$th GCN layer, which is defined as:

$$Z^{t+1} = f\left(\tilde{D}^{-1}\tilde{A}Z^{t}W^{t}\right) + Z^{t} \tag{4}$$

where $Z^0 = X$, $W^t \in R^{N\times c^t}$, and $Z^t \in R^{N\times c^{t-1}}$. c^t denotes the dimension of the output of the tth GCN layer. Note that the dimensions of Z^t and $Z^{(t+1)}$ are the same.

We finally concatenate the output of all the RGCN layers as matrix Z_n, and apply sort pooling on Z_n, whose rows are sorted by the non-ascending order of the last elements. We then keep the top K rows in Z_n and remove the other rows to preserve the most effective node features denoted as $T \in R^{K\times \sum_1^M c^t}$, where M is the number of GCN layers.

4.2.3. Edge attention module

In the EA module, we perform multi-head attention on edge series $\mathcal{E}$ containing all the SG edges in the BFS order to capture the SG edges' features, such that we can learn the semantic features in the code by capturing the graph structure features, which indicate rich semantic information in the code, including data–data, instruction–instruction and instruction–data relationships.

We extract the edges' features $\tilde{E}$ with the edge attention module. We keep the top K edge features of $\tilde{E}$ to obtain the new edge feature matrix $\tilde{E}' \in R^{K\times C}$. For the graph which has less than K edges, we pad $\tilde{E}$ with element 0 to obtain $\tilde{E}'$.

Multi-head attention parallelly calculates the similarity between each edge and the other edges. Therefore, during the parallel attention calculation, the sequence information of the edges is lost. That is, the edges will have the same attention calculation results, no matter how we permutate the order of the edges. Note that Transformer (Vaswani et al., 2017) uses positional encoding to restore the location information when performing multi-head attention. Consequently, similar to Transformer, we adopt positional encoding to give an order to each edge, such that we can maintain the sequence information of the edges. In order to mark the edge series with positional information, we also use positional encoding (Vaswani et al., 2017) to give an order to each edge. The positional encoding is defined as:

$$\begin{cases} PE_{(position,2l)} = \sin\left(position/10000^{2l/d_E}\right) \\ PE_{(position,2l+1)} = \cos\left(position/10000^{2l/d_E}\right) \end{cases} \tag{5}$$

where *position*, l, and d_E denote the position of the edge in the SG, the dimension of positional encoding, and the dimension of edge feature, respectively. We then obtain the new edge feature vector $\tilde{E}'$ with position information as follows:

$$\tilde{E}'' = \tilde{E}' + PE \tag{6}$$

We apply multi-head attention mechanism on $\tilde{E}''$. The input consists of queries and keys with dimension d_k, and values with dimension d_v. We compute the attention function on a set of queries simultaneously. The queries are packed together into a matrix Q. The keys and values are also packed together into matrices K' and V'. The attention relevance of the edge series is calculated as:

$$\text{Att}\left(Q, K', V'\right) = \text{softmax}\left(\frac{QK'^T}{\sqrt{d_k}}\right)V' \tag{7}$$

Multi-head attention enables the model to pay joint attention to the information from different parts of the sequence. Multi-head attention can be represented as follows:

$$\begin{aligned}\text{MultiHead}(Q, K', V') = \text{Concat}\Big(&Att_{l+1}(QW_{l+1}^{Q}, K'W_{l+1}^{K}, V'W_{l+1}^{V}),\\ &Att_{l}(QW_{l}^{Q}, K'W_{l}^{K}, V'W_{l}^{V})\Big)\end{aligned} \tag{8}$$

Table 1
Smart contract datasets.

DataSet	Arithmetic	Reentrancy	Timestamp dependency	Unchecked low calls
Vulnerable functions	2812	2989	723	103
Normal functions	3018	3219	854	134
Total functions	5830	6208	1577	237

where $W_l^Q \in R^{d_E\times d_q}$, $W_l^K \in R^{d_E\times d_k}$, and $W_l^V \in R^{d_E\times d_v}$ are the parameter matrices of calculating queries, keys, and values in the lth head attention. After obtaining the multi-head attention result H, we apply a normalization layer (LN) followed by an Multilayer Perceptron (MLP). We finally use an LN before output. We then obtain S, the result of EA module, by

$$S = LN\left(\text{ReLU}\left(W_{p+1}\cdot \text{Relu}\left(W_p\cdot LN(H) + b_p\right) + b_{P+1}\right)\right) \tag{9}$$

where $W_p \in R^{d_p\times d_{p+1}}$ is parameter matrix of MLP, $b_p \in R^{d_p\times 1}$ is bias of the MLP, and d_p denotes the dimensions of the hidden layer in the MLP.

4.3. Classification module

Module RGCN obtains a matrix T of size $K\times\sum_1^M c^t$ to represent the SG nodes' features, and module EA obtains a matrix S of size $K\times C$ to characterize the SG edges' features. In the classification module, we concatenate the nodes' features and the edges' features as the global code feature, and use a classifier to identify whether the function is vulnerable. Fig. 12 shows the structure of the classification module.

First, we perform row-wise reshaping on T and S to obtain a $K(\sum_1^M c^t)\times 1$ vector $\tilde{T}$ and a $KC\times 1$ vector $\tilde{S}$, respectively, which are concatenated into a matrix G as the global code feature. Second, we use a Conv1D layer to sequentially apply filters on the global code feature descriptors, where the filter size and the step size are both $\sum_1^M c^t$. Third, we apply several Conv1D layers and max-pooling layers to learn local patterns from the global code feature. Finally, we adopt multiple fully-connected layers followed by a sigmoid function to obtain the classification result.

5. Performance evaluation

5.1. Experimental setup

We evaluate the performance of our proposed method on the real-world vulnerable smart contracts from SmartBugs (Durieux et al., 2020), which are the same as (Zhuang et al., 2020; Wu et al., 2021). Our datasets consist of 13 852 labeled smart contract functions. Among them, there are 2812 and 3018 smart contract functions with and without arithmetic vulnerability, respectively; 2989 and 3219 smart contract functions with and without reentrancy vulnerability, respectively; 723 and 854 smart contract functions with and without timestamp dependency vulnerability, respectively; and 103 and 134 smart contract functions with and without unchecked low calls vulnerability, respectively. The numbers of vulnerable and normal functions are shown in Table 1.

We randomly select 80% of the dataset as the training set, 10% of the dataset as the validation set to validate the model's performance after each epoch, and the remaining 10% of the dataset as the test set. We repeat the experiment for 30 times, and report the averaged results.

Implementation details. All of our experiments are run on a desktop with an Intel Core i7-9700 3.00 GHz CPU, an NVIDIA GeForce 2080Ti GPU, and 32 GB memory. The SG construction and EA-RGCN model are written in Python and run under PyTorch.

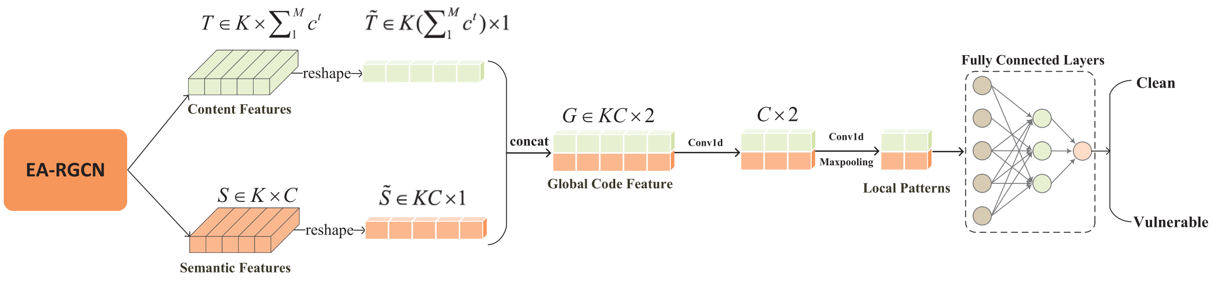


Fig. 12. Classification module.

Table 2
Parameters of EA-RGCN model.

Model parameter	Value
RGCN layers	12
Graph convolution size	[1, 11(middle), 1]
Learning rate	[0.0001, 0.0004]
Optimizer	Adam
K	115
L2 regularization	10
Batch size	5
Dropout rate	0.5
Feature channel	100
Number of attention heads	2

Performance metrics. We evaluate our method's performance in terms of Accuracy, Precision, Recall, and F1-score, which are calculated via Eqs. (10)–(13):

Accuracy: the ratio of the sum of TP and TN to the total number of samples

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

where *TP*, *FP*, *TN*, and *FN* are the numbers of true positive samples, false positive samples, true negative samples, and false negative samples, respectively.

Precision: the proportion of all classified positive samples that are positive.

$$Precision = \frac{TP}{TP + FP} \tag{11}$$

Recall: the proportion of positive samples that are classified as positive.

$$Recall = \frac{TP}{TP + FN} \tag{12}$$

F1-score: a combination of precision and recall for the overall performance.

$$F1{-}score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{13}$$

We further evaluate the effectiveness of our method using the area under the receiver operating characteristic curve (ROC-AUC), where ROC is a plot regarding True Positive Rate (TPR) versus False Positive Rate (FPR) at different thresholds.

Parameter settings. We use Adam optimizer in our model with cyclical learning rates used in Smith (2017), and we adopt grid search to find the best parameters with the following settings shown in Table 2.

5.2. Experimental results

In this section, we first analyze the impact of different components in our model on the performance, and then compare our code representation SG with CFG and DFG. Finally, we compare our EA-RGCN model with the existing smart contract vulnerability detection methods.

5.2.1. Impact of different components in model EA-RGCN

Edge series generation order. The edge series, as the input of EA, plays an important role in model EA-RGCN, since we obtain the edges' features from the edge series. In our method, we adopt BFS to obtain the edge series. Whereas, another order to create the edge series is DFS. Therefore, we study the impact of different edge series generation orders on the performance of model EA-RGCN. Specifically, we compare BFS-EA-RGCN with DFS-EA-RGCN, which use BFS and DFS to generate the edge series, respectively.

Table 3 shows that BFS-EA-RGCN achieves obviously better performance than DFS-EA-RGCN on all the four vulnerability detection tasks. For example, the accuracy, precision, recall, and F1-score of BFS-EA-RGCN are about 38.69∼45.97%, 45.47∼57.07%, 30.47∼41.77%, and 40.56∼54.33% higher than those of DFS-EA-RGCN, respectively. The ROC-AUC scores of BFS-EA-RGCN are about twice of those of DFS-EA-RGCN on all the four vulnerability detection tasks. We need to convert a two-dimensional code graph structure into a one-dimensional sequence. Meanwhile, we need to keep the graph structure information as much as possible during the conversion. DFS traverses the edges as far as possible along each branch before backtracking, which focuses on the local information of an edge, i.e., some incomplete code execution segments. The edge series in the BFS order is generated level by level and hence preserves the semantic information of the code. Accordingly, BFS can effectively help EA extract the graph structure features, which indicate rich semantic information in the code, including data–data, instruction–instruction and instruction–data relationships. On the contrary, DFS loses the graph information, which disables EA to achieve the desired effect of obtaining the semantic features. Note that the features of a semantic graph include semantic features as well as code content features. Therefore, EA-RGCN with BFS significantly outperforms EA-RGCN with DFS, and we choose to obtain the edge series in the BFS order.

RGCN and EA in model EA-RGCN. RGCN and EA learn the node and the edge features from the SG, respectively. In order to evaluate the impact of RGCN module on the model performance, we conduct the comparison among EA-RGCN, EA, and EA-GCN. EA excludes the node feature extraction module, and EA-GCN obtains the node features using GCN without the residual connection. With the purpose of investigating the impact of EA module, we compare EA-RGCN with RGCN. Note that there is no need to generate the edge series, if EA is not included in the model. The experimental results are shown in Table 4.

EA-RGCN outperforms EA-GCN in terms of accuracy, precision, recall, F1-score on the four vulnerability detection tasks. For example, in the arithmetic, timestamp dependency, and unchecked low calls vulnerability detection tasks, EA-RGCN improves the accuracy by about 4.02% over EA-GCN. In the reentrancy detection

Table 3
Performance of BFS-EA-RGCN and DFS-EA-RGCN in terms of accuracy, recall, precision, F1-score, and ROC-AUC.

(a) Performance on arithmetic and reentrancy vulnerability detection tasks.

Methods	Arithmetic					Reentrancy				
	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC
DFS-EA-RGCN	51.78	45.69	58.53	49.47	0.5234	48.95	37.85	52.19	40.09	0.4874
BFS-EA-RGCN	**90.47**	**91.16**	**89.00**	**90.03**	**0.9620**	**94.00**	**94.92**	**93.96**	**94.42**	**0.9886**

(b) Performance on timestamp dependency and unchecked low calls vulnerability detection tasks.

Methods	Timestamp dependency					Unchecked low calls				
	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC
DFS-EA-RGCN	50.69	45.10	59.61	46.70	0.4865	43.65	30.57	53.75	35.67	0.4413
BFS-EA-RGCN	**91.98**	**95.06**	**91.66**	**93.35**	**0.9601**	**83.33**	**81.48**	**89.50**	**84.65**	**0.8915**

Table 4
Impact of modules RGCN and EA on the model performance in terms of accuracy, recall, precision, F1-score, and ROC-AUC, where EA (w/o PE) denotes EA module without positional encoding.

(a) Performance on arithmetic and reentrancy vulnerability detection tasks.

Methods	Arithmetic					Reentrancy				
	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC
EA (w/o PE)	84.40	86.26	81.60	83.82	0.9250	91.50	93.63	92.23	92.92	0.9786
EA	85.62	88.06	82.61	85.22	0.9281	92.70	94.43	92.93	93.63	0.9856
RGCN	83.33	87.62	77.64	82.26	0.9013	93.61	93.33	91.92	92.59	0.9856
EA-GCN	86.45	87.55	85.34	86.40	0.9378	93.77	93.81	93.86	93.82	0.9876
EA-RGCN	**90.47**	**91.16**	**89.00**	**90.03**	**0.9620**	**94.00**	**94.92**	**93.96**	**94.42**	**0.9886**

(b) Performance on timestamp dependency and unchecked low calls vulnerability detection tasks.

Methods	Timestamp dependency					Unchecked low calls				
	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC
EA (w/o PE)	87.37	85.42	86.53	85.97	0.9402	69.52	64.65	70.99	65.61	0.7954
EA	88.62	88.37	89.09	88.66	0.9419	78.73	74.71	87.40	79.62	0.8530
RGCN	87.03	88.33	85.56	86.86	0.9367	68.57	66.44	77.36	69.58	0.7391
EA-GCN	88.13	87.84	89.38	88.54	0.9458	79.52	78.86	84.38	80.37	0.8660
EA-RGCN	**91.98**	**95.06**	**91.66**	**93.35**	**0.9601**	**83.33**	**81.48**	**89.50**	**84.65**	**0.8915**

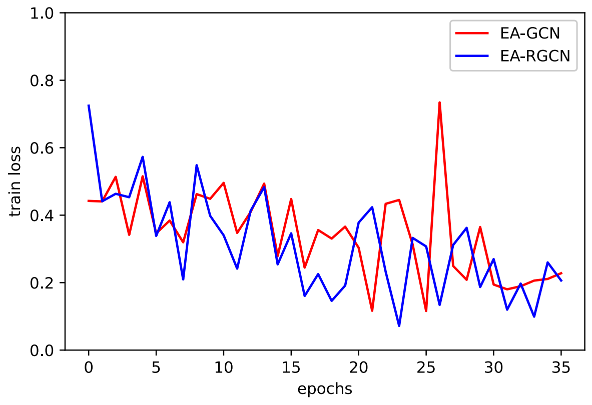


Fig. 13. The training loss of EA-RGCN and EA-GCN on arithmetic vulnerability detection.

task, the performance improvement of EA-RGCN over EA-GCN is up to 1.11%. The ROC-AUC of EA-RGCN is 0.016 higher than that of EA-GCN in average. GCN with the residual blocks can avoid large fluctuation of training loss and hence make the model easy to optimize. Take the training process of EA-RGCN and EA-GCN on arithmetic vulnerability dataset for instance, which is shown in Fig. 13. It can be observed that the model without residual blocks suffers from great training loss variation. In general, Table 4 and Fig. 13 show that residual blocks play an important role in the model. Therefore, we choose to use residual blocks in our model to help the model optimize better and more easily.

EA-RGCN also outperforms EA. Specifically, EA-RGCN achieves up to 4.85%, 6.77%, 6.39%, 5.03%, and 0.0385 better performance than EA on the four vulnerability detection tasks in terms of accuracy, precision, recall, F1-score and ROC-AUCA, respectively. Table 4 shows that EA-GCN obtains better performance than EA. The comparison among EA-RGCN, EA-GCN, and EA demonstrate that module RGCN can significantly improve the performance when extracting the node feature from the SG, and hence outperforms GCN in the four vulnerability detection tasks.

The performance of EA-RGCN is better than RGCN. Especially, in the unchecked low calls vulnerability detection, EA-RGCN achieves 14.76%, 15.04%, 12.14%, and 15.07% improvement over RGCN in terms of accuracy, precision, recall, and F1-score, respectively. In the other three detection tasks, EA-RGCN also outperforms RGCN up to 7.14% in terms of accuracy, precision, recall, and F1-score. In addition, EA-RGCN obtains a 0.059 better ROC-AUC score than RGCN in average. The SG edges represent the data–data, instruction–instruction and instruction–data relationship in the code, and the proposed EA module can well extract the semantic feature of the code from the edges. Without the EA module, the vulnerability detection performance will be remarkably reduced.

The performance of EA is better than EA (w/o PE). Specifically, in the unchecked low calls vulnerability detection, EA gains 9.21%, 10.06%, 16.41%, and 14.01% improvement over EA (w/o PE) in terms of accuracy, precision, recall, and F1-score, respectively. In the other three vulnerability detection tasks, EA still achieves 0.70∼2.95% improvement over EA (w/o PE) in terms of accuracy, precision, recall, and F1-score. In the EA module, we perform multi-head attention on the edge series to capture the SG edges' features. Multi-head attention parallelly calculates the similarity between each edge and the other edges. Therefore, during the attention calculation, the sequence information of the edges is lost. To this end, we utilize positional encoding to restore the location information before multi-head attention is performed.

Table 5
Performance of different graph representations in terms of accuracy, recall, precision, F1-score, and ROC-AUC score.

(a) Performance on arithmetic and reentrancy vulnerability detection tasks.

Methods	Arithmetic					Reentrancy				
	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC
EA-RGCN(DFG)	84.47	85.24	84.06	84.63	0.9180	92.45	92.92	92.25	92.51	0.9858
EA-RGCN(CFG)	84.48	86.91	81.01	83.80	0.9118	93.35	93.98	92.77	93.33	0.9881
EA-RGCN(SG)	**90.47**	**91.16**	**89.00**	**90.03**	**0.9620**	**94.00**	**94.92**	**93.96**	**94.42**	**0.9886**

(b) Performance on timestamp dependency and unchecked low calls vulnerability detection tasks.

Methods	Timestamp dependency					Unchecked low calls				
	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC
EA-RGCN(DFG)	84.27	84.99	82.27	83.29	0.9273	63.49	58.01	81.92	65.71	0.6844
EA-RGCN(CFG)	88.06	86.48	90.51	88.40	0.9489	79.68	76.90	86.20	80.54	0.8765
EA-RGCN(SG)	**91.98**	**95.06**	**91.66**	**93.35**	**0.9601**	**83.33**	**81.48**	**89.50**	**84.65**	**0.8915**

The experimental results also verify the effectiveness of positional encoding.

In general, the experimental results shown in Table 4 illustrate that the two parallel modules RGCN and EA can promote the vulnerability detection, since RGCN can capture the code content features and EA can extract the rich semantic features. Our model EA-RGCN integrates both modules and concatenates the features learned by the two modules, such that the model can extract more vulnerability related features.

5.2.2. Comparison study

Comparison of different graph-based code representations. We evaluate the performance of our proposed SG against CFG (Yan et al., 2019) and DFG (Wu et al., 2021). Specifically, we compare SG+EA-RGCN, CFG+EA-RGCN, and DFG+EA-RGCN. The experimental results in Table 5 show that our proposed SG outperforms CFG and DFG in terms of accuracy, precision, recall, F1-score, and ROC-AUC score. SG achieves up to 5.99%, 8.58%, 7.99%, and 6.23% improvement over CFG in terms of accuracy, precision, recall, and F1-score, respectively. SG also gains 0.0192 better ROC-AUC score than CFG in average. SG also obtains a significant improvement over DFG. In the unchecked low calls vulnerability detection, SG achieves 25.32%, 23.47%, 7.58%, and 19.84% improvement over DFG in terms of accuracy, precision, recall, and F1-score, respectively. In the other three vulnerability detection tasks, the performance improvement of SG over DFG can reach 1.55∼10.07%. In terms of the ROC-AUC score, SG gains 0.071 improvement over DFG in average. The comparison among SG, CFG, and DFG demonstrates that our proposed smart contract representation SG can significantly improve the performance on the four vulnerability detection tasks. CFG ignores the data flows in the code, while the data error is a vital result of vulnerabilities. DFG overlooks the instructions in the code, while the smart contracts are run via the instructions. Our proposed SG can fully represent the statement execution sequences and data flows. Therefore, the SG can characterize more information about the code than CFG and DFG.

Comparison of different smart contract vulnerability detection methods. We investigate the performance of our method SG+EA-RGCN against four state-of-the-art conventional methods (Smartcheck Tikhomirov et al., 2018, Oyente Luu et al., 2016, Securify Grishchenko et al., 2018, Slither Feist et al., 2019) and six state-of-the-art deep learning based methods (Transformer Vaswani et al., 2017, DGCNN Zhang et al., 2018, TMP Zhuang et al., 2020, Peculiar Wu et al., 2021, CGE Liu et al., 2021b, AME Liu et al., 2021a). Smartcheck is an extensible static analysis tool for smart contract vulnerability detection. Oyente is a smart contract vulnerability detection tool applying symbolic verification on the CFG of the smart contract. Securify uses formal verification to detect the smart contract defects. Slither checks malicious smart contracts by converting them into an intermediate representation statically. Transformer is widely used to process sequences. In the experiment, we use Transformer on the sequence obtained from the code AST. DGCNN is a popular model to process graphs. In the experiment, we run DGCNN on the SG proposed in this paper. TMP (Zhuang et al., 2020) and Peculiar (Wu et al., 2021) are two state-of-the-art methods for smart contract vulnerability detection. TMP learns normalized contract graphs of smart contracts via a temporal message propagation network. Peculiar uses pre-trained graph code BERT to process the crucial data flow of smart contracts. CGE (Liu et al., 2021b) combines the graph feature with the expert knowledge to target reentrancy and timestamp dependency vulnerabilities, while AME (Liu et al., 2021a) uses attention to process the expert knowledge introduced by CGE. There is no expert knowledge for the arithmetic and unchecked low calls vulnerabilities. Therefore, we only evaluate CGE and AME on reentrancy and timestamp dependency vulnerability detection. Table 6 shows the experimental results.

In general, the conventional methods perform worse than the deep learning based methods. Slither achieves the best performance among the four conventional methods on the three tasks of reentrancy, timestamp dependency and unchecked low calls vulnerability detection. However, Slither cannot detect the arithmetic vulnerability. In the experiments, our method gains 21.51% accuracy improvement over Slither in average on the three vulnerability detection tasks. Conventional methods use application-specific predefined rules which are prone to errors. In contrast, deep learning based methods can learn the code features to improve the performance of smart contract vulnerability detection.

From the experimental results of deep learning based methods, we have the following observations. First, the graph-based methods DGCNN, TMP, Peculiar, CGE, AME, and EA-RGCN achieve better performance than Transformer which processes the AST of the code as a sequence. For example, DGCNN gains 4.93% accuracy and 0.1546 ROC-AUC score improvement over Transformer on arithmetic and reentrancy detection tasks in average. Compared with the graph, the AST cannot characterize the execution order of statements and lacks the relationship between data. The result demonstrates that treating smart contract code as a sequence results in inferior performance on smart contract vulnerability detection tasks. Therefore, it is better to represent the source code as a graph.

Second, the performance of TMP and Peculiar are not stable on the four vulnerability detection tasks. Peculiar achieves better performance on reentrancy vulnerability detection than TMP, but is outperformed by TMP on timestamp dependency and unchecked low calls detection tasks. TMP is proposed to process reentrancy and timestamp dependency vulnerability, and the performance comparison shows that the generalization ability of TMP should be improved. Peculiar aims especially on the

Table 6
Performance of different methods of Transformer, DGCNN, TMP, Peculiar, CGE, AME, and EA-RGCN(SG) in terms of accuracy, recall, precision, and F1-score. '–' denotes not applicable.

(a) Performance on arithmetic and reentrancy vulnerability detection tasks.

Methods	Arithmetic					Reentrancy				
	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC
Smartcheck (Tikhomirov et al., 2018)	50.38	60.92	2.12	4.10	–	50.34	90.47	0.76	1.51	–
Oyente (Luu et al., 2016)	50.64	80.77	1.68	3.29	–	60.60	50.78	43.16	46.66	–
Securify (Grishchenko et al., 2018)	–	–	–	–	–	63.34	52.49	50.23	51.34	–
Slither (Feist et al., 2019)	–	–	–	–	–	68.33	66.19	70.87	68.45	–
Transformer (Vaswani et al., 2017)	77.45	83.26	67.04	74.27	0.8149	87.45	83.70	81.62	82.50	0.9071
DGCNN (Yan et al., 2019)	81.90	85.58	77.13	81.02	0.8906	92.85	93.29	92.07	92.66	0.9860
TMP (Zhuang et al., 2020)	82.31	86.35	81.34	83.77	0.8978	92.99	93.50	92.98	93.20	0.9840
Peculiar (Wu et al., 2021)	87.70	88.03	87.62	87.79	0.9165	92.40	93.39	86.79	90.12	0.9843
CGE (Liu et al., 2021b)	–	–	–	–	–	93.67	93.63	92.89	93.21	0.9872
AME (Liu et al., 2021a)	–	–	–	–	–	93.88	93.75	93.42	93.49	0.9878
EA-RGCN(SG)	**90.47**	**91.16**	**89.00**	**90.03**	**0.9620**	**94.00**	**94.92**	**93.96**	**94.42**	**0.9886**

(b) Performance on timestamp dependency and unchecked low calls vulnerability detection tasks.

Methods	Timestamp dependency					Unchecked low calls				
	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	ROC-AUC
Smartcheck (Tikhomirov et al., 2018)	50.00	50.21	45.68	47.94	–	57.21	66.78	59.63	63.00	–
Oyente (Luu et al., 2016)	53.00	52.17	59.68	59.37	–	–	–	–	–	–
Securify (Grishchenko et al., 2018)	–	–	–	–	–	65.00	62.88	61.21	63.04	–
Slither (Feist et al., 2019)	66.43	67.37	66.54	66.95	–	70.00	68.54	70.11	69.32	–
Transformer (Vaswani et al., 2017)	86.18	81.09	89.05	84.88	0.9198	55.84	59.46	53.66	56.39	0.6964
DGCNN (Yan et al., 2019)	84.75	88.66	81.64	84.43	0.9226	49.37	36.09	64.86	44.60	0.6433
TMP (Zhuang et al., 2020)	87.66	88.12	89.98	89.01	0.9512	57.21	67.85	72.45	70.01	0.6132
Peculiar (Wu et al., 2021)	81.18	79.19	82.29	80.29	0.9045	59.17	51.98	52.77	49.79	0.5680
CGE (Liu et al., 2021b)	89.22	88.24	90.21	89.21	0.9362	–	–	–	–	–
AME (Liu et al., 2021a)	90.34	87.56	91.45	89.46	0.9535	–	–	–	–	–
EA-RGCN(SG)	**91.98**	**95.06**	**91.66**	**93.35**	**0.9601**	**83.33**	**81.48**	**89.50**	**84.65**	**0.8915**

reentrancy vulnerability detection task, and the crucial data flow used by Peculiar only considers the data–data relationship in a smart contract, while ignoring the instruction–instruction and instruction–data relationships in the code. Therefore, the performance of Peculiar is not stable on the detection tasks of vulnerabilities other than reentrancy.

Third, our method outperforms the other graph-based methods, i.e., TMP, Peculiar, CGE and AME. Compared with TMP and Peculiar, our model EA-RGCN achieves 9.90% accuracy and 0.089 ROC-AUC score improvement over TMP, and obtains 9.83% better accuracy and 0.10 higher ROC-AUC score than Peculiar in average. CGE and AME are not applicable to arithmetic and unchecked low calls vulnerability detection, due to their requirements of expert knowledge. In reentrancy and timestamp dependency detection, CGE and AME achieve better performance than the other baseline methods. However, our method obtains 1.55% better accuracy and 0.012 higher ROC-AUC score than CGE, and improves AME by 0.88% in accuracy and 0.00037 in ROC-AUC score in average. The experimental results show that our method SG+EA-RGCN achieves stable performance on all the four vulnerabilities detection tasks. Our proposed SG uses a graph to represent the smart contract code. When constructing an SG, we first remove the unnecessary nodes in the AST and then add edges between the nodes to represent the data flows and the execution sequence of the statements, such that the instruction–instruction, data–data, and instruction–data relationship in the code can be fully characterized. Our model EA-RGCN further extracts the rich syntax and semantic feature which is the concatenation of the code content features obtained by RGCN and the semantic features extracted by EA.

6. Conclusion

In this paper, we tackled the smart contract vulnerability detection problem at the function level. First, we proposed a new graph representation called semantic graph (SG) for each function in smart contracts. The SG removes the useless nodes from the AST of a smart contract function and adds edges to construct execution sequences and data flows. The SG can fully characterize the instruction–instruction, data–data, and instruction–data relationship in the smart contract code. Second, we proposed a novel model EA-RGCN to extract the content and semantic features of the code from the SG. In contrast to the existing models, our model can extract code content features from the SG nodes using the residual GCN (RGCN) module and semantic features from the SG edges with the edge attention (EA) module. Finally, the two features are concatenated as the global code feature which is fed into a classifier to obtain the vulnerability identification result. We conducted experiments on four important vulnerabilities including arithmetic, reentrancy, timestamp dependency, and unchecked low calls. The experimental results demonstrated that the graph is a better representation of the smart contract code than the sequence and the tree. The proposed SG can characterize more information about the code than the state-of-the-art graph-based representations. Furthermore, the experimental results illustrated that both RGCN and EA contribute to the vulnerability detection. Therefore, our proposed method achieves superior performance in terms of accuracy, precision, recall, F1-score, and ROC-AUC on smart contract vulnerability detection compared to the state-of-the-art conventional methods and deep learning based approaches. The source code of the proposed method will be publicly available at: https://github.com/frankdadale/EA-RGCN.

CRediT authorship contribution statement

Da Chen: Conceptualization, Methodology, Software, Writing – original draft. **Lin Feng:** Investigation, Supervision. **Yuqi Fan:** Conceptualization, Writing- Original Draft, Writing – review & editing. **Siyuan Shang:** Data curation, Validation. **Zhenchun Wei:** Writing – review & editing.

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

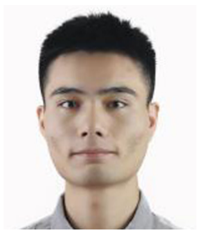
Data will be made available on request

Acknowledgments

This work is partially supported by National Natural Science Foundation of China (62002097), the Major Science and Technology Projects in Anhui Province (202003a05020009), and the open fund of Intelligent Interconnected Systems Laboratory of Anhui Province (PA2021AKSK0114), Hefei University of Technology.

References

Cao, S., Sun, X., Bo, L., Wei, Y., Li, B., 2021. BGNN4VD: Constructing bidirectional graph neural-network for vulnerability detection. Inf. Softw. Technol. 136, 106576.
Dinh, T.T.A., Liu, R., Zhang, M., Chen, G., Ooi, B.C., Wang, J., 2018. Untangling blockchain: A data processing view of blockchain systems. IEEE Trans. Knowl. Data Eng. 30 (7), 1366–1385.
Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P., 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). Seoul, South Korea, 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 (WETSEB). Montreal, Quebec, Canada, pp. 8–15.
Grishchenko, I., Maffei, M., Schneidewind, C., 2018. A semantic framework for the security analysis of Ethereum smart contracts. In: International Conference on Principles of Security and Trust. Thessaloniki, Greece, pp. 243–269.
Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S.K., Clement, C.B., Drain, D., Sundaresan, N., Yin, J., Jiang, D., Zhou, M., 2021. GraphCodeBERT: Pre-training code representations with data flow. In: 9th International Conference on Learning Representations (ICLR).
Hochreiter, S., Schmidhuber, J., 1997. Long short-term memory. Neural Comput. 9 (8), 1735–1780.
Jiang, B., Liu, Y., Chan, W., 2018. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In: 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). Montpellier, France, pp. 259–269.
Liu, H., Liu, C., Zhao, W., Jiang, Y., Sun, J., 2018. S-Gram: Towards semantic-aware security auditing for Ethereum smart contracts. In: 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). Montpellier, France, pp. 814–819.
Liu, Z., Qian, P., Wang, X., Zhu, L., He, Q., Ji, S., 2021a. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. In: 19th International Joint Conference on Artificial Intelligence (IJCAI). Montreal, Canada, pp. 2751–2759.
Liu, Z., Qian, P., Wang, X., Zhuang, Y., Qiu, L., Wang, X., 2021b. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Trans. Knowl. Data Eng. 1.
Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A., 2016. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS). Vienna, Austria, pp. 254–269.
Ma, R., Jian, Z., Chen, G., Ma, K., Chen, Y., 2019. ReJection: A AST-based reentrancy vulnerability detection method. In: 13th Chinese Conference on Trusted Computing and Information Security (CTCIS). Shanghai, China, pp. 58–71.
Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. 26, pp. 3111–3119,
Nakamoto, S., 2008. Bitcoin: A peer-to-peer electronic cash system. Decentralized Bus. Rev. 21260.
2021. Python-solidity-parser. URL https://github.com/ConsenSys/python-solidity-parser.
Qian, P., Liu, Z., He, Q., Zimmermann, R., Wang, X., 2020. Towards automated reentrancy detection for smart contracts based on sequential models. IEEE Access 8, 19685–19695.
Sankar, L.S., Sindhu, M., Sethumadhavan, M., 2017. Survey of consensus protocols on blockchain applications. In: 4th International Conference on Advanced Computing and Communication Systems (ICACCS). Coimbatore, India, pp. 1–5.
Smith, L.N., 2017. Cyclical learning rates for training neural networks. In: 17th IEEE Winter Conference on Applications of Computer Vision (WACV). Santa Rosa, California, pp. 464–472.
Tian, G., Wang, Q., Zhao, Y., Guo, L., Sun, Z., Lv, L., 2020. Smart contract classification with a Bi-LSTM based approach. IEEE Access 8, 43806–43816.
Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y., 2018. SmartCheck: Static analysis of Ethereum smart contracts. In: 1st ACM/IEEE International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). Gothenburg, Sweden, pp. 9–16.
Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I., 2017. Attention is all you need. In: 31st International Conference on Neural Information Processing Systems (NIPS). Long Beach, California, USA, pp. 6000–6010.
Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y., Bian, L., Wang, Z., 2021. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. Inf. Forensics Secur. (TIFS) 16, 1943–1958.
Wood, G., 2014. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Pap. 151 (2014), 1–32.
Wu, H., Zhang, Z., Wang, S., Lei, Y., Lin, B., Qin, Y., Zhang, H., Mao, X., 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In: 32nd International Symposium on Software Reliability Engineering (ISSRE). Wuhan, China, pp. 378–389.
Xu, Y., Hu, G., You, L., Cao, C., 2021. A novel machine learning-based analysis model for smart contract vulnerability. Secur. Commun. Netw. (SCN) 2021, 12.
Yamaguchi, F., Golde, N., Arp, D., Rieck, K., 2014. Modeling and discovering vulnerabilities with code property graphs. In: IEEE Symposium on Security and Privacy (S&P). Berkeley, California, USA, pp. 590–604.
Yan, J., Yan, G., Jin, D., 2019. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In: 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). Portland, Oregon, USA, pp. 52–63.
Zhang, M., Cui, Z., Neumann, M., Chen, Y., 2018. An end-to-end deep learning architecture for graph classification. In: Proceedings of 32nd AAAI Conference on Artificial Intelligence, Vol. 32. New Orleans, Louisiana, USA, pp. 4438–4445.
Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., He, Q., 2020. Smart contract vulnerability detection using graph neural network. In: 29th International Joint Conference on Artificial Intelligence (IJCAI). Yokohama, Japan, pp. 3283–3290.



Da Chen received his B.S. degree in Computer Science and Engineering from Anhui Agricultural University, China, in 2021. He is currently a graduate student in Computer Science at School of Computer Science and Information Engineering at Hefei University of Technology, Hefei, China. His research interests include blockchain, software security, and deep learning, etc.

Lin Feng was born in 1979. She received her Ph.D. degree from Hefei University of Technology in 2014. She is an associate professor with the School of Computer Science and Information at Hefei University of Technology. Her research interests include blockchain, vehicular ad-hoc network, and wireless network.

Yuqi Fan is an associate professor at School of Computer Science and Information Engineering at Hefei University of Technology, China. He received his Ph.D. in Computer Science and Engineering from Wright State University in 2009. He received both B.S. and M.S. degrees in computer science and engineering from Hefei University of Technology in 1999 and 2003, respectively. His research interests include blockchain, computer networks, cloud computing, cyber–physical systems, etc.

Siyuan Shang received both B.S. and M.S. degree in Computer Science and Engineering from Hefei University of Technology, Hefei, China in 2019 and 2022. His research interests include smart contract security and deep learning, etc.

Zhenchun Wei was born in 1978. He received his Ph.D. degree from Hefei University of Technology in 2007. He is an associate professor with the School of Computer Science and Information Engineering at Hefei University of Technology. His research interests include Internet of Things, edge computing, wireless sensor networks, distributed intelligence, and blockchain.