Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH

Open Access

HGAT: smart contract vulnerability detection method based on hierarchical graph attention network

Chuang Ma[1], Shuaiwu Liu[1] and Guangxia Xu[2*]

Abstract

With the widespread use of blockchain, more and more smart contracts are being deployed, and their internal logic is getting more and more sophisticated. Due to the large false positive rate and low detection accuracy of most current detection methods, which heavily rely on already established detection criteria, certain smart contracts additionally call for human secondary detection, resulting in low detection efficiency. In this study, we propose HGAT, a hierarchical graph attention network-based detection model, in order to address the aforementioned issues as well as the shortcomings of current smart contract vulnerability detection approaches. First, using Abstract Syntax Tree (AST) and Control Flow Graph, the functions in the smart contract are abstracted into code graphs (CFG). Then abstract each node in the code subgraph, extract the node features, utilize the graph attention mechanism GAT, splice the obtained vectors to form the features of each line of statements and use these features to detect smart contracts. To create test data and assess HGAT, we leverage the open-source smart contract vulnerability sample dataset. The findings of the experiment indicate that this method can identify smart contract vulnerabilities more quickly and precisely than other detection techniques.

Keywords Smart Contract, BlockChain, Graph Attention Network, Vulnerability Detection, Security

Introduction

Through the consensus process, the blockchain has recently made sure that all nodes can achieve point-to-point communication without the requirement for a reliable third party [1, 2]. Blockchain is also frequently used in logistics, banking, edge computing, medical and other industries because to its tamper proof, traceable and decentralized qualities [3–8]. Smart contracts, which operate as an application atop the blockchain's top layer, offer a comprehensive transaction protocol for all types of virtual currency transactions, resolving the issue that the blockchain has not seen widespread use since its inception. Smart contracts are essentially pieces of code with contractual properties, which unavoidably results in programming flaws. Many contract attacks have been revealed in recent years [9]. Hackers targeted Coincheck, a significant Japanese digital currency exchange, in January 2018, and more than 534 million dollars' worth of NEM (New Economy Currency) on the site was moved unlawfully; Hackers attacked the decentralized lending platform Lendf.Me on April 19, 2020, and stole the contract's assets worth $25 million. On April 19, 2020, Lendf.Me, a decentralized lending agreement, was attacked by hackers, and the assets worth 25 million dollars in the contract were looted; More than 37 significant attacks in the blockchain ecological field were observed in the third quarter of 2022, with a total loss of roughly $450.4 million,

*Correspondence:
Guangxia Xu
xugx@gzhu.edu.cn
[1] School of Software Engineering, Chongqing University of Posts and Telecommunications, Chongqing 400065, China
[2] Advanced Institute of Cyberspace Technology, Guangzhou University, Guangzhou 510006, China

according to the Beosin EagleEye security early warning and monitoring platform, which not only resulted in significant financial losses for consumers but also posed a serious threat to the security and trust interests of smart contracts [10].

The main reasons for smart contract vulnerabilities are as follows: (1) The source code is transparent and accessible. Since smart contracts are immutable, program patches can no longer be used to change the smart contract that has been uploaded to the blockchain. Attacker entry requirements and attack costs are lowered because of this because attackers can exploit loophole contracts on the chain to launch attacks. (2) Developers' carelessness. Given that the smart contract was written using Solidity high-level Programming language, flaws in the Solidity design pattern and developers' writing styles could result. Developers, for instance, neglected to verify potential contract flaws, leading to integer overflow, arbitrary address writing, uninitialized variables, etc. (3) Design level of an EVM virtual machine. Vulnerabilities are brought on by the bytecode and EVM virtual machine design guidelines. For instance, a frequent vulnerability at the design level of the EVM virtual machine is the segment address attack.

Currently, the smart contract code audit approach mostly relies on experts for manual audit, but because manual audit is expensive and inefficient, experts struggle to fully extract the vulnerability features in each contract [11]. The existing research frequently simplifies the code into natural language or relies on the subject-matter expertise of specialists to extract code features. The process of translating the code into plain language is overly straightforward, which makes it simple to lose information. Experts frequently use a heuristic rule model that is too sophisticated, not scalable, and not universal, which results in low detection accuracy and efficiency of vulnerabilities. Studying effective vulnerability feature extraction techniques is crucial in light of the security issues with smart contracts.

Aiming at the above problems, this paper proposes a smart contract vulnerability detection model HGAT based on hierarchical graph attention network. Our main contributions are as follows:

1. A contract graph contraction strategy is suggested to shrink the contract graph's nodes in order to speed up model learning and avoid the model from being overfit.
2. A hierarchical graph attention network-based technique for detecting smart contract vulnerabilities is proposed. To increase the ability to extract discontinuous features, features are extracted at the statement level and the function level, respectively. The efficiency of this strategy has been tested on open source datasets.
3. It has been demonstrated through a large number of experiments that the HGAT model put out in this research performs better than the current detection techniques.

The organization of this paper is as follows: we introduce existing smart contract vulnerability detection methods and their flaws in Section "Related Work"; detailed implementation details of HGAT are introduced in Section "Method"; and Section "Experiments" compares HGAT with other detection methods; Summarize and prospect in Section "Conclusion".

Related work

Smart contract vulnerability detection

In recent years, The blockchain has gained popularity because to its immutability. However, once the smart contract becomes immutable, if there is a security problem in the smart contract on the chain, it will bring serious economic losses to people. Therefore, it is crucial to identify smart contract vulnerabilities. Smart contract vulnerability detection is mainly divided into two types: manual detection and automatic detection [12]. Faced with the deployment of thousands of smart contracts every day, experts in manual detection need to analyze the Opcode [13] of smart contracts first, and then match vulnerabilities according to the rules defined in advance. This does not satisfy the criteria of blockchain and smart contract development prospects and is not only time-consuming and expensive, but frequently ineffectual even when flaws are uncovered.

In accordance with the characteristics of the detection, automated detection is classified into three categories: code matching based on features, automatic detection based on formal verification, and automatic auditing techniques based on symbolic execution and symbolic abstraction [14–16]. Similar to string matching, feature-based code matching primarily abstracts known smart contract vulnerabilities into a semantic tag before matching with the smart contract to be detected. The method based on formal verification approach primarily makes use of mathematical techniques and automated detection tools to replicate the deployment and use of smart contracts and check for security vulnerabilities. Amani et al. [17] proposed to partition contracts into fundamental fast codes or build EVM bytecode sequences into linear code blocks using an Isabelle-based higher-order logic interaction theorem prover, and then use formal model verification to determine whether the smart contract code contains vulnerabilities. Karthikeyan et al. [18] proposed

a verification framework for the functional correctness of smart contracts based on formal verification. This framework converts smart contract code into F* language format and verifies various smart contract attributes to find hidden vulnerabilities. The approach based on symbolic execution and symbolic abstraction generates Opcode by compiling the smart contract using the Ethereum virtual machine, and then analyzes the Opcode to see whether there is a security vulnerability. Securify [19] is symbolic abstract analysis-based detection tools, while Mythril [20] and Oyente [16] are symbolic execution-based detection tools. A static analysis tool for Ethereum smart contracts called SmartCheck has been proposed by Tikhomirov et al. [21] It employs the ANTLR parser as well as a customized Solidity syntax. By utilizing ANTLR syntax and vulnerability patterns, it can find more sophisticated vulnerabilities.

There are several similarities between the traditional automated smart contract vulnerability detection techniques mentioned above [22, 23]. First, traditional smart contract vulnerability detection tools have low code coverage and high false positive and false negative rates. For instance, Oytente's code coverage is erratic and its rate of false positives for integer overflow vulnerabilities is significant. Additionally, because typical smart contract vulnerability detection tools rely on detection rules established by experts, their scalability is constrained, often necessitating secondary detection, and their detection effectiveness is low.

It's important to note that researchers have recently put forth approaches for detecting smart contract vulnerabilities that based on machine learning and neural networks. These techniques are more accurate and efficient than more traditional ones. A neural network-based smart contract fuzzer called ILF was proposed by He et al. [24]. It uses a symbolic execution engine to produce effective test and call sequences to direct the feature learning of neural network models, enabling effective vulnerability discovery. Wang et al. [25] proposed ContractWard, which uses machine learning technology to detect vulnerabilities in smart contracts. First, using the XGBoost training model and the SMOTETomek balanced training set, binary features are extracted from the condensed operation code of smart contracts. Second, the model to simultaneously detect multiple vulnerabilities is built using five machine learning techniques and two sampling algorithms. Huang et al. [26] proposed an unsupervised graph embedding algorithm that transforms the code graph into a quantitative comparable vector, transforms the vulnerability characteristics into the same dimension vector, and finds the potentially vulnerable smart contract by comparing its vector to the known vulnerable vector.

The internal logic of smart contracts, however, becomes more and more complex as the number of smart contracts rises, and the current detection methods have certain issues:

1. Existing detection methods will use the entire smart contract as input for detection, including code segments unrelated to vulnerabilities, such as keywords CALL, CALL.VALUE for reentrancy vulnerabilities, integer operations for integer overflow, etc. This will make detection more difficult and lower detection effectiveness.
2. Since source code is more logical and structural than natural language, many vulnerabilities occur across functions, such as function calls, which requires smart contract vulnerability detection methods to be able to handle discontinuous feature extraction.

Therefore, in order to further reduce the security risk of smart contracts, it is necessary to design a method with high detection efficiency and high detection accuracy.

Graph Attention Network (GAT)

Gori et al. first proposed the concept of Graph Neural Network (GNN) [27]. The core idea is to learn the representation of the target node or state by iteratively disseminating and gathering the information of surrounding nodes and finally reaching a stable state. Joan Bruna et al. [28] proposed that Graph Convolutional Networks (GCNs) can combine local graph structure and node features to achieve good performance in node classification tasks. Petar Velickovic et al. [29] proposed (Graph Attention Network, GAT), which uses an attention mechanism to weighted summation of adjacent node features. GAT is independent of the graph structure, and the weight of adjacent node features completely depends on the node features.

As graph neural networks advance, more and more use applications for them are emerging in areas including semantic segmentation, association extraction, and picture categorization. Additionally, it can be utilized to understand and recognize program behavior. A graph neural network is used to learn the program behavior that is challenging to learn from the prior deep learning model after building the graphic data of the program's data flow and control flow. In order to lower the high false positive rate of smart contract vulnerability identification, Ma et al. [30] proposed an abstract syntax tree (AST) based vulnerability detection method to efficiently find reentrancy vulnerabilities. Xu et al. [31] proposed a binary code similarity comparison method based on graph neural network, which discovered the bytecode containing the vulnerability keyword by compiling the

code into bytecode and matching it with the vulnerability keyword, in order to determine whether there is a vulnerability in the contract. This demonstrates the enormous potential of graph neural networks in the field of vulnerability mining. Zhuang et al. [32] proposed a new message propagation network to use GCNs for intelligent contract vulnerability identification. They also built a contract graph representing syntax and semantics. The disadvantage of this method is that the way GCNs aggregate the attributes of nearby nodes is intimately tied to the graph's structure, which restricts the trained model's capacity to generalize to other graph structures. Zhou et al. [33] proposed a general model based on graph neural networks for graph-level classification by learning a rich set of code semantic representations to efficiently extract useful features. Liu et al. [34] proposed a novel temporal message propagation network to extract graph features from normalized graphs, and combined graph features with a designed expert mode to improve vulnerability detection accuracy. After GAT introduces the attention mechanism, each graph node is only related to the adjacent nodes, and there is no need to obtain the information of the whole graph, which improves the generalization ability on different graph structures, breaks the limitations of GCN, and also provides Smart contract vulnerability detection with complex logic makes it possible. From this point of view, it is feasible to apply GAT to smart contract vulnerability detection.

Considering that the source code of smart contract is written in Solidity high-level programming language, which is as sparse as the code written in other languages, we have layered it on the basis of GAT, which is divided into code statement level and function level. The representation result of a function is obtained at the code statement level, while the representation result of the full smart contract is obtained at the function level. Finally, vulnerability categorization is performed using the learnt representation. Because HGAT is based on GAT, it not only has strong heterogeneous task processing capabilities but also improves the capacity to extract the specifics of the smart contract environment and lessen contract sparsity, as will be covered in Section "Generation of semantic features".

Method

This paper proposes a smart contract vulnerability detection method based on hierarchical graph attention network. This method is divided into 4 stages: 1) In the contract graph generation stage, an abstract syntax tree (AST) is generated according to the smart contract source code. Each node represents the declared variable or incoming parameter, and the edge represents the calling relationship between them. Generate Contract Graph (SCG) by combining semantic information of Control Flow Graph (CFG); 2) In the shrinking stage of graph nodes, in view of the increasingly complex logic of the source code of smart contracts and the diversity of information in the contract graph, it is not conducive to model learning. Therefore, in this step, the contract graph is shrunk to aggregate the node information; 3) In the semantic feature generation stage, input to the hierarchical graph attention network to vectorize the node features; 4) In the stage of obtaining the result, the vector output by the hierarchical graph attention network is input to the Softmax layer for classification, and the final prediction result is obtained. The overall design of the method is shown in Fig. 1. The above four stages are described in detail below.

Contract chart generation

To better model smart contracts, we abstract smart contracts using a graph structure. This article uses SolidityParser tool to generate AST for smart contract source code. SolidityParser generates a parser from Solidity grammars and uses the parser to identify the structure of the program. However, AST still cannot make full use of the structural information of code fragments. Therefore, CFG is used to better capture the rich semantics between nodes, and CFG is generated by using the calling relationship between functions, and then control flow, data flow and function call are combined into abstract syntax. On the tree, generate SCG. Where SCG = (V, E), V represents the node set.

Each declared variable or function parameter in the smart contract represents a node $V_i(0 \leq i \leq n)$, and the node includes variable calls, function calls, data flow and other characteristics. Nodes represent important

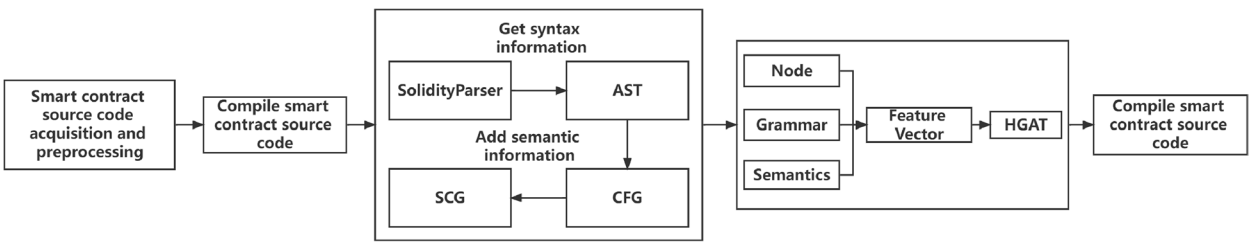

Fig. 1 Overall design steps of the method

elements in the smart contract Solidity, such as function and variable names. They are mainly divided into three categories: primary node F, secondary node S, and callback node C. The main node is a function that has an important impact on smart contract vulnerability detection, including Solidity's built-in functions and custom functions, such as the call.value function of sending Ether, the withdrawal function of withdraw, etc.; Secondary nodes represent important variables, such as balance balances[address], Ether boundary value amountLimit used for verification, etc. Since most vulnerabilities in smart contracts are associated with callback functions, callback nodes are used as the callback function of the attacked contract in indicated in the SCG. E represents an edge set, each edge is a triple $E_i = (V_i, V_j, \text{isLink})$, V_i, V_j represent node i and node j, isLink represents whether node i and node j are connected, and the edge describes the data flow in the contract And the transmission process of control flow, while reflecting the variable transfer and function call. Figure 2 shows the process of contract graph generation.

As seen in Fig. 2, the left side of the figure is an example of smart contract code, and the right side is a contract graph created from a smart contract. The primary node F consists of three functions: deposit, withdraw and mash. sender.call. Secondary nodes include variables like balances and bal. Create the contract graph by scanning it from top to bottom to identify all primary and secondary nodes, and then add details like control flow and data flow according to the order of execution. Consider the deposit function's execution sequence as an illustration. Balances[msg. sender] is the secondary node S1, amount is the secondary node S2, and the deposit function is the primary node F1. Both the out-degrees and in-degrees of S1 contain e2, representing S1 for auto increment operation. Using the control flow edge e3, one can determine whether S2 is greater than 0. When the condition is satisfied, it will proceed to the F2 node to execute the withdraw function.

Graph node shrinkage

After the contract graph is generated, the contract graph needs to be embedded. However, different functions have varying degrees of importance when it comes to identifying smart contract vulnerabilities. For instance, some functions are only used to initialize variables, so the likelihood that they will generate vulnerabilities is very low; And the information in the contract graph is diverse, which is not conducive to model learning. Therefore, the model should be shrunk before inputting the contract graph into the model. Specifically, the characteristics of the secondary node S and the callback node C in the contract graph are passed to the adjacent main node F, and finally only the main node is retained, which is recorded as node X_i.

Figure 3 is a contract diagram of Fig. 2 after feature shrinking of secondary nodes and callback nodes. In the specific operation, each secondary node or callback node is shrunk to the closest primary node. Taking the secondary node S1 as an example, the out-degrees of S1 include e3, e5, and e2, and the in-degrees and

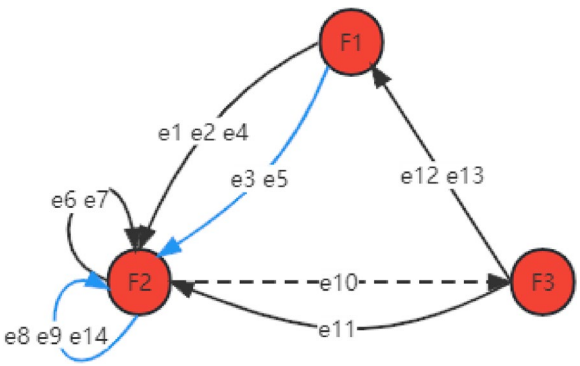

Fig. 3 Contract after shrinkage

```
mapping(address => uint) balances;
uint public totalSupply;

function deposit(uint amount) public payable {
    balances[msg. sender] += msg.value;
    if(amount > 0)
        withdraw(amount)
}

function withdraw(uint weiToWithdraw) public {
    uint bal = balances[msg.sender];
    require(bal > 0);
    require(bal > = weiToWithdraw);
    (bool sent,) = msg.sender.call {value: bal}("");
    require(sent, "Failed to send Ether");
    balances[msg.sender]= 0;
}
```

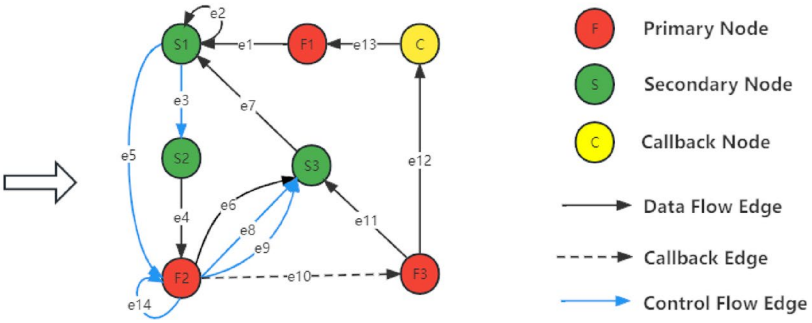

Fig. 2 Contract Graph Generation

out-degrees of the secondary node S1 both include e2. The in-degrees of S2 includes e3, but S2 is not the primary node. Find the edge e4 with S2 as the outgoing degree, we can see e4 and e5 both point to the primary node F2, Therefore, secondary nodes S1 and S2 can be shrunk to the primary node F2. When S1 and S2 are deleted, the associated edge e1, which connects the primary nodes F1 and S1, will also point to the primary node F2. This completes the shrinking of the secondary node S2. After the contraction is completed, the characteristics of the contract graph nodes include three parts: the characteristics of the original main node F_i, the characteristics aggregated by the secondary node S_i, and the characteristics aggregated by the callback node C_i, as shown in formula 1.

$$X_i = f(F_i, \sum S_i, \sum C_i) \tag{1}$$

Generation of semantic features

The framework of the HGAT model proposed in this paper is shown in Fig. 4. The model is divided into 5 layers, namely: graph embedding layer, sentence layer, function layer, splicing layer, and Softmax layer. Each statement in source code is related to its context and follows a hierarchy similar to that of documents. HGAT retains the hierarchical structure of the source code and captures the semantic characteristics of different levels in the source code hierarchically. Because each statement has syntax and semantic characteristics, vulnerabilities are often caused by a keyword or function call, which

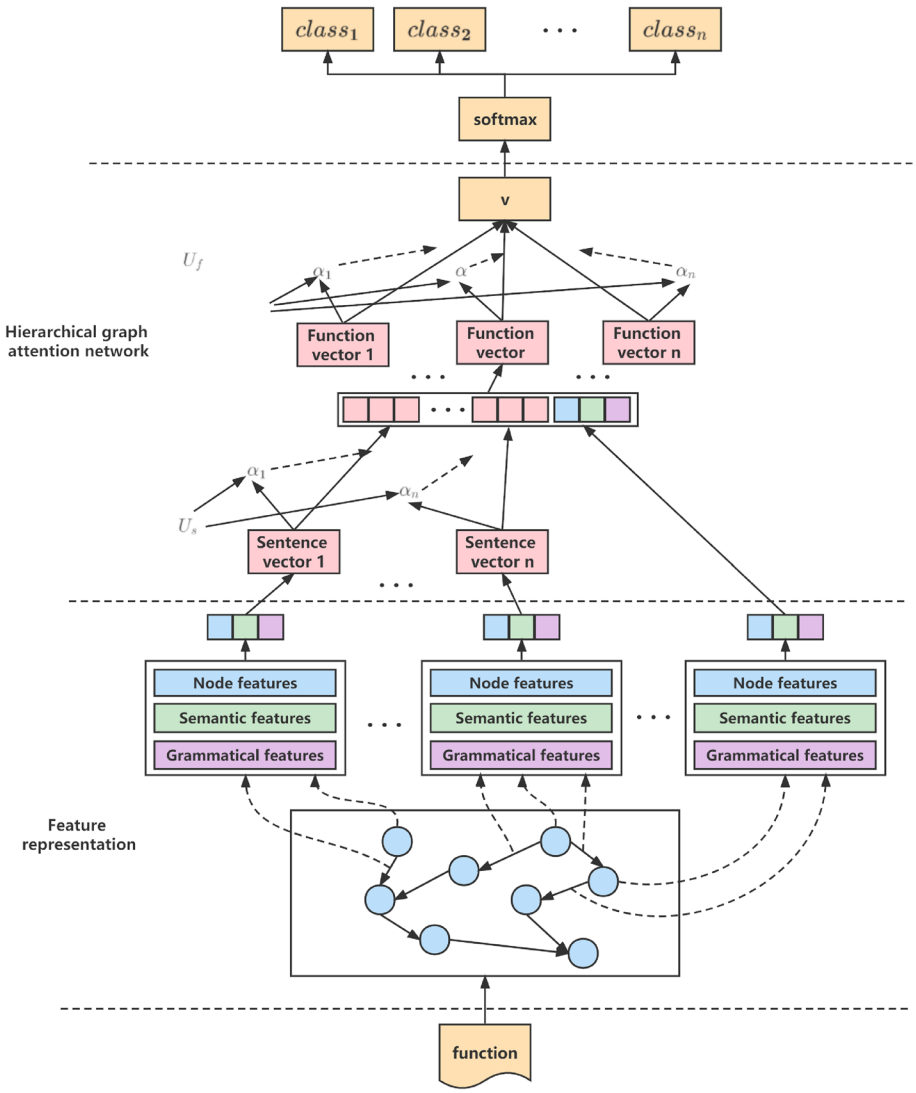

Fig. 4 HGAT Model Structure

are included in the semantic and syntax characteristics. In a function of the smart contract, learning contributes different sentence weights to detecting vulnerabilities, and the representation of sentences is learned through weighted fusion. Then, the sentences in each function are spliced to obtain different weights of different functions in the smart contract, and the weighted fusion is used to obtain the document representation.

First, input according to the contract contraction chart generated in Section "Graph node shrinkage", assuming that a contract shrinkage graph has N nodes, the input graph pays attention to the feature T of each node of the network as:

$$t = \overrightarrow{t_1}, \overrightarrow{t_2}, ..., \overrightarrow{t_n}, t_i \in \mathbb{R}^T \tag{2}$$

where T is the dimension of node characteristics. In the sentence layer, it only targets the statement logic inside the function, including features such as sequence flow and control flow inside the function. The final output of all node features is expressed as:

$$t^{'} = \overrightarrow{t_1}, \overrightarrow{t_2}, ..., \overrightarrow{t_n}, \overrightarrow{t_i} \in \mathbb{R}^{T^{'}} \tag{3}$$

For each shrinking node X_i in the contract shrinking graph, the importance weights of neighbor nodes will be learned. In order to obtain sufficient expressive ability to learn the attention weights, a learnable linear transformation W is required in this process to achieve feature enhancement. Assuming that the input feature is converted into the weight matrix W of the output feature, the attention value of node j on node i is:

$$e_{ij} = a(W_{t_i}, W_{t_j}) \tag{4}$$

where a is a single-layer feedforward neural network, which is the attention function for calculating node correlation. It is then activated using the *LeakyReLU* activation function:

$$l_{ij} = LeakyReLU(a^T[W_{t_i}||W_{t_j}]) \tag{5}$$

where $\|$ is the splicing operation, and a^T is the transpose of a. After finding the attention values of all adjacent nodes of each shrinking node X_i, use the Softmax function to normalize the attention weights to obtain the sentence-level attention weights:

$$\alpha_{ij} = softmax(e_{ij}) = \frac{exp(l_{ij})}{\sum\limits_{k \in N_i} l_{ik}} \tag{6}$$

Most of the smart contract vulnerabilities occur in callback functions, but these callback functions may not be called in their contexts. Only the call relationship between functions is taken into account in the function layer. The eigenvector of the function call flow on the statement layer's output splice serves as the input of the function layer:

$$n_{ij} = a(W_{t_i}, W_{t_j}) \tag{7}$$

After that, the Softmax function is also used for normalization:

$$\beta_{ij} = softmax(n_{ij}) \tag{8}$$

Through the two-layer attention network, the weighted sum method is used for feature extraction, and the output of the model is:

$$t_i^{'} = \sigma(\sum_{j \in N_i} \beta_{ij} W t_j) \tag{9}$$

where σ is the activation function, and β_{ij} is the attention value of node i and node j. In this model, a multi-head attention network is used, and k groups of attention coefficients are independently calculated and averaged to prevent overfitting. The final output of the model is:

$$t_i^{'} = \sigma(\frac{1}{K}\sum_{K=1}^{K}\sum_{j \in N_i} \beta_{ij}^k W^k \overrightarrow{t_j}) \tag{10}$$

Generation of semantic features

After obtaining the output result of the double-layer graph attention network, it is input into the fully connected layer and mapped to the sample label space. The calculation formula of the fully connected layer is as follows:

$$\hat{t} = T^{'} w + b \tag{11}$$

where $T^{'}$ is the output of all node features, w is the parameter matrix of the fully connected layer, and b is the bias. In order to reduce the isomorphism bias generated by the graph structure data in the classification model, after the output of the fully connected layer is obtained, it is input to the softmax layer for normalization, and the final prediction result is:

$$t_i^{'} = softmax(\hat{t}) \tag{12}$$

Through the softmax function, the output can be converted into a probability, classified, and the smart contract vulnerability classification label $t^{'}$ can be obtained to complete the prediction target.

Calculate Loss using the cross-entropy loss function:

$$Loss = -\sum t_i \ln y_i^{'} \tag{13}$$

Experiments

Data set and experimental platform

The experimental environment of this paper is Ubuntu 18.04 operating system, 3.7 GHz Intel Core i7 CPU processor, 16 GB memory, RTX 3060Ti graphics card, 8 GB video memory. We use SmartBugs Wild dataset [35] to verify the advantages and disadvantages of the methods in this paper. This is a large-scale dataset of smart contract vulnerabilities based on Solidity. There are about 203716 smart contracts in this data collection, 47518 real and distinct sol files, and 9693457 lines. The sol files in the SmartBugs Wild dataset are finally divided into two categories: smart contracts with vulnerabilities and smart contracts without vulnerabilities. The number of smart contracts with vulnerabilities is 35151, and the number of smart contracts without vulnerabilities is 168565. The vulnerable smart contract contains six types of vulnerabilities, namely, stack call depth attack vulnerability (Callstack depth attack), integer up overflow vulnerability, integer down overflow vulnerability, reentrancy vulnerability, timestamp dependency vulnerability, and transaction order dependency vulnerability. The dataset we utilize consists of 7018 smart contracts, including 1640 integer overflows vulnerabilities, 1988 integer underflows vulnerabilities, 1671 timestamp dependence vulnerabilities, and 1719 Reentrancy vulnerabilities. We divided these smart contracts into 5615 training sets and 1403 test sets.

Evaluation indicators

This paper makes a comparative experiment from two aspects. To demonstrate the effectiveness of our HGAT model, the method in this study is first compared to other smart contract vulnerability detection techniques and evaluated with metrics including accuracy rate (ACC), recall rate (TPR), accuracy rate (PRE), and F1 value. Then, by contrasting the detection times, the detection effectiveness of HGAT was confirmed.

Analysis of experimental results

We set cross-entropy as the loss function and 0.001 as the learning rate for the model. The loss value is essentially stable after the model has been trained for many epochs, showing that the model can converge successfully. The accuracy and loss curves of Reentrancy, Timestamp, Integer Overflow and Integer Underflow vulnerabilities detection during model training are shown in Fig. 5. Figure 5 shows that HGAT has high detection performance for Reentrancy, Timestamp, Integer Overflow and Integer Underflow vulnerabilities. Table 1 lists the HGAT detection indications.

In order to verify the effectiveness of the method in this paper, we compared it with four static analysis tools, Oyente, Mythril, Securify, Slither, and two deep learning detection methods, LSTM and GCN. It can be seen from Fig. 6 that the performance of HGAT is better than other detection methods. Oyente's detection result is less than optimal and its PRE is only 38.43% since it relies on detection rules established by experts and does not completely imitate Ethereum's execution environment. Mythril is similar in performance to Oyente. Securify uses three modes to locate errors. Only in the warning mode does it need to perform manual secondary detection, which improves the accuracy of Securify, but TPR and PRE are still low. Slither can draw the inheritance topology diagram of the contract, the contract method call relationship diagram, etc., and the detection accuracy rate reaches 77.34%. LSTM only aggregates the feature information of the source code context, so it ignores the function call, so the accuracy rate is low, only 53.68%. GCN performs well in various indicators, but due to the weak generalization ability of GCN, the accuracy is only 70.02%. The HGAT proposed in this paper surpasses the other 6 methods in 4 indicators, and the detection accuracy rate is greatly improved, reaching 85.64%, and the recall rate is also improved. It can be seen from the comparative experimental results that the method in this paper has good detection performance for Reentrancy vulnerabilities of smart contracts. Figure 7 shows the detection results of Timestamp vulnerabilities by each detection tool. ACC, PRE, TPR and F1 of the proposed HGAT model were 87.98%, 89.69%, 85.59% and 85.26%, indicating that HGAT was superior to other models in detection accuracy. Since the Timestamp vulnerability occurs when the function is called, it can also be seen that HGAT's ability to extract discontinuous features is due to other detection methods. Figures 8 and 9 show the detection results of various detection tools for Integer Overflow and Integer Underflow vulnerabilities. It can be seen that HGAT performs better than other models. Compared with Slither, the detection accuracy increased by 11.70% and 10.20%, respectively. The detection accuracy of GCN increased by 11.48% and 9.02%, respectively.

The ROC curve for both this method and the comparison method are shown in Fig. 10. The AUC is the area under the ROC curve, which is used as a performance indicator. The larger the AUC value, the better the performance. The AUC of Oyente and Securify is modest, which indicates poor performance, as can be seen in the figure. HGAT has the largest AUC value, and its

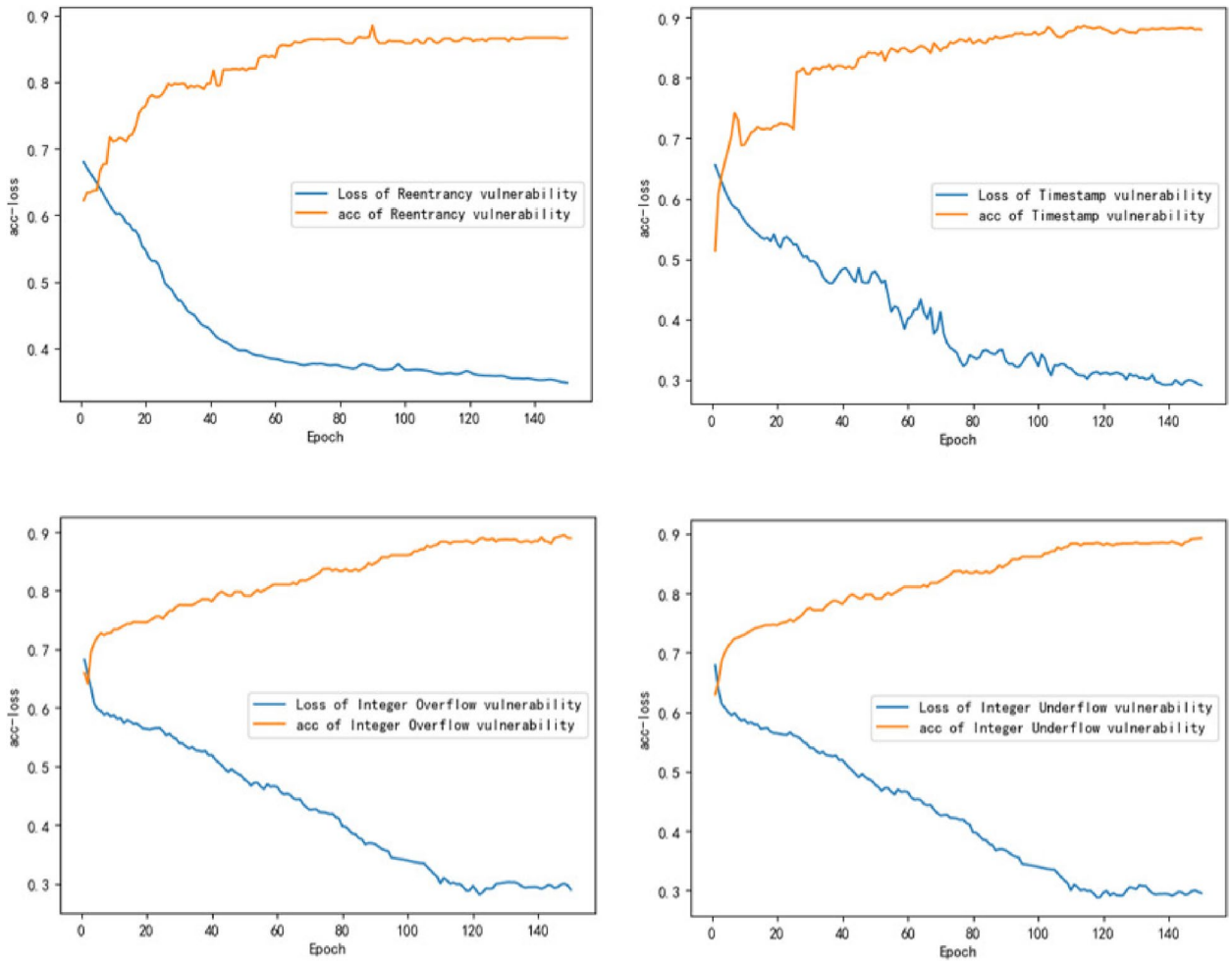

Fig. 5 Reentrancy, Timestamp, Integer Overflow and Integer Underflow vulnerabilities Detection Accuracy and Loss Curve

Fig. 6 Reentrancy vulnerability detection results

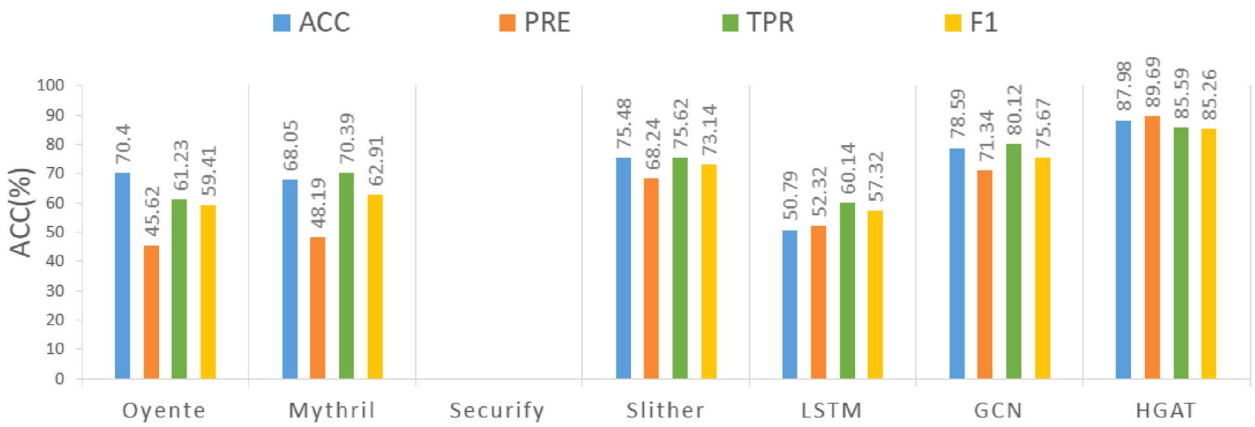

Fig. 7 Timestamp vulnerability detection results

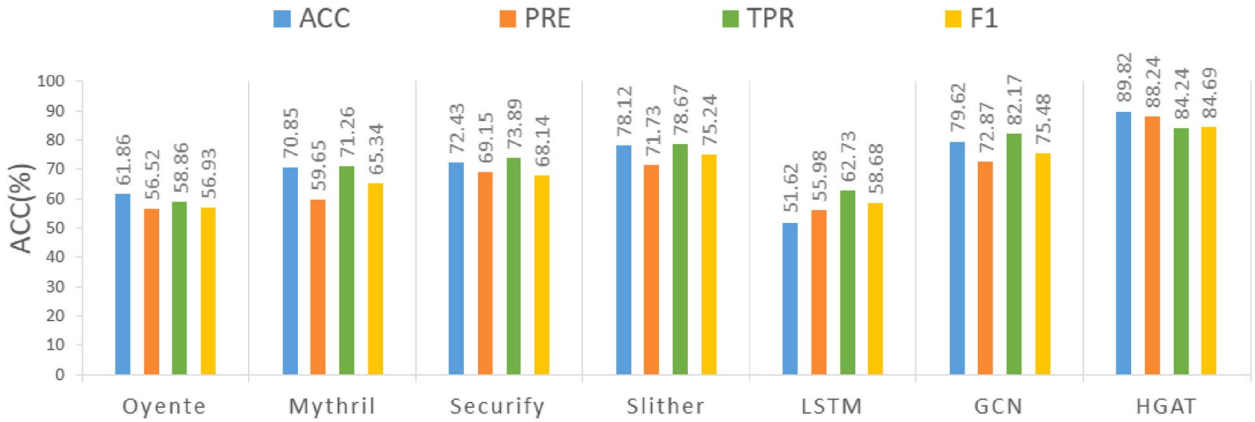

Fig. 8 Integer Overflow vulnerability detection results

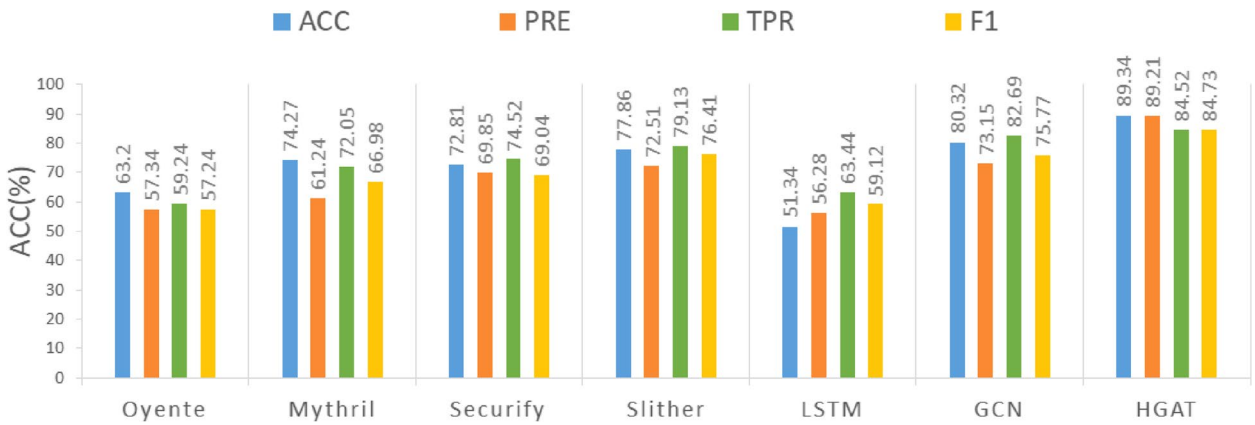

Fig. 9 Integer Underflow vulnerability detection results

performance is better than other detection methods, indicating that the method proposed in this paper performs better than other detection methods.

Timing begins before the detection tool or model inputs graph structure data in terms of detection time, and ends after the output results. Under the same hardware

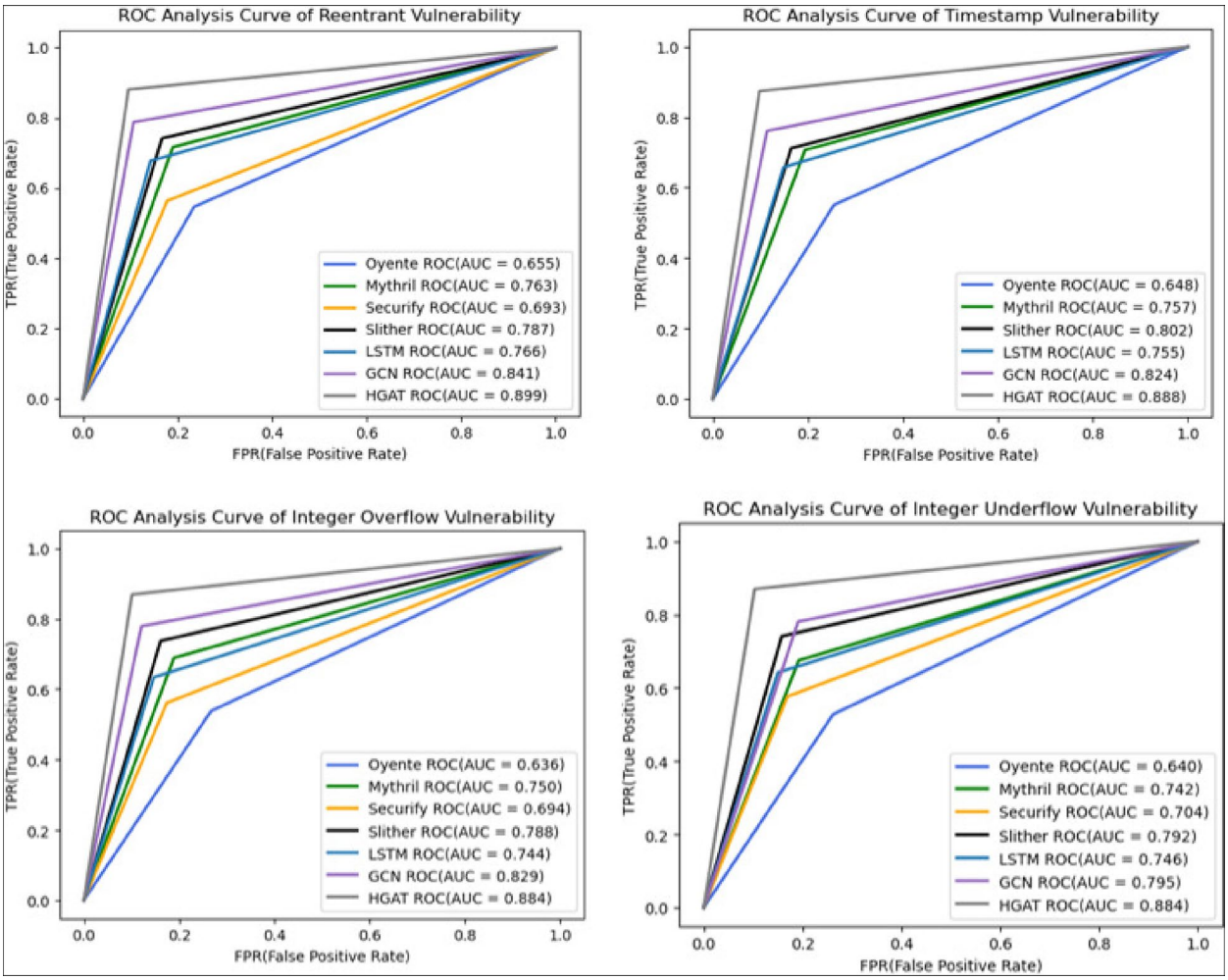

Fig. 10 ROC analysis curve on Reentrancy, Timestamp, Integer overflow and Integer Underflow vulnerabilities

Table 1 Test results based on SmartBugs Wild dataset

Vulnerability Type	ACC/%	PRE/%	TPR/%	F1 Score
Reentrancy	85.64	76.96	88.12	82.47
Timestamp	86.62	82.69	89.62	88.31
Integer Overflow	88.26	83.12	87.14	83.08
Integer Underflow	88.44	83.64	87.17	83.15

conditions and experimental environment, the average detection time obtained by statistics is shown in Table 2.

The test findings in Table 2 demonstrate the poor detection performance of four conventional detection tools. A smart contract may be found in almost a minute using the Mythril detection tool. The detection efficiency is better, at about 1s, with the deep neural network detection model. Compared to LSTM and GCN, the method proposed in this study is a little bit slower. The use of the hierarchical attention network, which has higher computational complexity than a basic neural network when an attention mechanism is introduced, may be the cause of the longer detection times.

Table 2 Average detection time (unit: second)

Methods	Oyente	Mythril	Securify	Slither	LSTM	GCN	HGAT
Average Time	6.18	54.72	13.58	5.26	0.83	0.93	1.04

Conclusion

In this paper, we proposed a method for detecting smart contracts' reentrancy vulnerability based on hierarchical graph attention network, which generated local abstract semantic graph data from the syntax information of the abstract syntax tree in the smart contract source code, and the semantic information of control flow and data flow, used hierarchical graph attention network to spread and aggregate information on the graph, and used two-layer attention mechanism to extract extract features of smart contract samples. Finally, a reentrancy vulnerability detection experiment was carried out on the open source smart contract dataset, which proved that the method had a good detection performance on the reentrancy vulnerability of smart contracts. The next step is to continue to improve the model to improve the accuracy and extend it to other types of vulnerability detection and to improve the detection efficiency without reducing the accuracy of the model.

Acknowledgements
The authors would like to thank the teachers and students of the Blockchain and Information Security Laboratory, School of Software Engineering, Chongqing University of Posts and Telecommunications for their help and valuable opinions.

Authors' contributions
Chuang Ma found the target problem and proposed the solution. Shuaiwu Liu completed most of the writing of this manuscript. Guangxia Xu helped in revising the paper and gave many useful suggestions. All authors have read and approved the manuscript.

Authors' information
Chuang Ma is currently a lecturer in the School of Software Engineering of Chongqing University of Posts and Telecommunications. He obtained a master's degree in network and information security from Jilin University in 2011 and a doctor's degree in computer system structure from Jilin University in 2016. Mainly engaged in the research and development of complex networks, big data analysis and processing, and artificial intelligence. Shuaiwu Liu is now studying for a master's degree in the School of Software Engineering of Chongqing University of Posts and Telecommunications, and received a bachelor's degree in software engineering from Sichuan University of Light Chemical Industry in 2016. His main research interests are blockchain, smart contract, and network security. Guangxia Xu is currently a professor at Cyberspace Institute of Advanced Technology of Guangzhou University in Guangdong, China. She received the M.S. and Ph.D. degrees in computer science from the Chongqing University, Chongqing, China in 2006 and 2011, respectively. Her research interests include information security and network management, Big Data analytics for network security, and Blockchain technology. She is a committee member at the Blockchain of CCF, IEEE Senior Member and ACM member.

Funding
This work is supported by the National Natural Science Foundation of China (Grant No. 62272120, 62106030); the Technology Innovation and Application Development Projects of Chongqing (Grant No. cstc2021jscx-gksbX0032, cstc2021jscx-gksbX0029); the Research Program of Basic Research and Frontier Technology of Chongqing (Grant No. cstc2021jcyj-msxmX0530); the Key R & D plan of Hainan Province (Grant No. ZDYF2021GXJS006).

Availability of data and materials
Not applicable.

Declarations

Ethics approval and consent to participate
The work is a novel work and has not been published elsewhere nor is it currently under review for publication elsewhere.

Consent for publication
Informed consent was obtained from all individual participants included in the study.

Competing interests
The authors declare no competing interests.

Received: 8 November 2022 Accepted: 18 May 2023
Published online: 22 June 2023

References

1. Liu Y, Xu G (2021) Fixed degree of decentralization dpos consensus mechanism in blockchain based on adjacency vote and the average fuzziness of vague value. Comput Netw 199:108432
2. Xu G, Liu Y, Khan PW (2020) Improvement of the dpos consensus mechanism in blockchain based on vague sets. IEEE Trans Ind Inform 16(6):4252–4259. https://doi.org/10.1109/TII.2019.2955719
3. Scekic O, Nastic S, Dustdar S (2019) Blockchain-supported smart city platform for social value co-creation and exchange. IEEE Internet Comput 23(1):19–28. https://doi.org/10.1109/MIC.2018.2881518
4. Du J, Cheng W, Lu G, Cao H, Chu X, Zhang Z, Wang J (2021) Resource pricing and allocation in mec enabled blockchain systems: An a3c deep reinforcement learning approach. IEEE Trans Netw Sci Eng 9(1):33–44
5. Feng J, Zhang W, Pei Q, Wu J, Lin X (2022) Heterogeneous computation and resource allocation for wireless powered federated edge learning systems. IEEE Trans Commun 70(5):3220–3233
6. Feng J, Liu L, Pei Q, Li K (2021) Min-max cost optimization for efficient hierarchical federated learning in wireless edge networks. IEEE Trans Parallel Distrib Syst 33(11):2687–2700
7. Mao S, Liu L, Zhang N, Dong M, Zhao J, Wu J, Leung VC (2022) Reconfigurable intelligent surface-assisted secure mobile edge computing networks. IEEE Trans Veh Technol 71(6):6647–60
8. Xu G, Dong J, Ma C, Liu J, Cliff UGO (2022) A certificateless signcryption mechanism based on blockchain for edge computing. IEEE Internet Things J
9. He D, Deng Z, Zhang Y, Chan S, Cheng Y, Guizani N (2020) Smart contract vulnerability analysis and security audit. IEEE Netw 34(5):276–282. https://doi.org/10.1109/MNET.001.1900656
10. Wang X, He J, Xie Z, Zhao G, Cheung SC (2020) Contractguard: Defend Ethereum smart contracts with embedded intrusion detection. IEEE Trans Serv Comput 13(2):314–328. https://doi.org/10.1109/TSC.2019.2949561
11. Xing C, Chen Z, Chen L, Guo X, Zheng Z, Li J (2020) A new scheme of vulnerability analysis in smart contract with machine learning. Wirel Netw 1–10
12. FU M, WU L, HONG Z, Wenbo F (2019) Research on vulnerability mining technique for smart contracts. J Comput Appl 39(7):1959
13. Wood G et al (2014) Ethereum: A secure decentralised generalised transaction ledger. Ethereum Proj Yellow Pap 151(2014):1–32
14. Dika A, Nowostawski M (2018) Security vulnerabilities in Ethereum smart contracts. In: 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Halifax, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), pp 955–962
15. D'Silva V, Kroening D, Weissenbacher G (2008) A survey of automated techniques for formal software verification. IEEE Trans Comput Aided Des Integr Circ Syst 27(7):1165–1178. https://doi.org/10.1109/TCAD.2008.923410
16. Luu L, Chu DH, Olickel H, Saxena P, Hobor A (2016) Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA, CCS '16, p 254–269. https://doi.org/10.1145/2976749.2978309

17. Amani S, Bégel M, Bortin M, Staples M (2018) Towards verifying Ethereum smart contract bytecode in Isabelle/hol. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, Association for Computing Machinery, New York, NY, USA, CPP 2018, p 66–77. https://doi.org/10.1145/3167084
18. Bhargavan K, Delignat-Lavaud A, Fournet C, Gollamudi A, Gonthier G, Kobeissi N, Kulatova N, Rastogi A, Sibut-Pinote T, Swamy N, Zanella-Béguelin S (2016) Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, Association for Computing Machinery, New York, NY, USA, PLAS '16, p 91–96. https://doi.org/10.1145/2993600.2993611
19. Mueller B, Honig J, Parasaram N (2018) Consensys/mythril. https://github.com/ConsenSys/mythril. Accessed 5 Sept 2022
20. Tsankov P, Dan A, Cohen DD, Gervais A, Buenzli F, Vechev M (2018) Securify: Practical security analysis of smart contracts. arXiv:1806.01143
21. Tikhomirov S, Voskresenskaya E, Ivanitskiy I, Takhaviev R, Marchenko E, Alexandrov Y (2018) Smartcheck: Static analysis of Ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, Moscow, pp 9–16
22. Grishchenko I, Maffei M, Schneidewind C (2018) Foundations and tools for the static analysis of Ethereum smart contracts. In: International Conference on Computer Aided Verification, Springer, pp 51–78
23. Di Angelo M, Salzer G (2019) A survey of tools for analyzing Ethereum smart contracts. In: 2019 IEEE international conference on decentralized applications and infrastructures (DAPPCON), IEEE, pp 69–78
24. He J, Balunović M, Ambroladze N, Tsankov P, Vechev M (2019) Learning to fuzz from symbolic execution with application to smart contracts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA, CCS '19, p 531–548. https://doi.org/10.1145/3319535.3363230
25. Wang W, Song J, Xu G, Li Y, Wang H, Su C (2020) Contractward: Automated vulnerability detection models for Ethereum smart contracts. IEEE Trans Netw Sci Eng 8(2):1133–1144
26. Huang J, Han S, You W, Shi W, Liang B, Wu J, Wu Y (2021) Hunting vulnerable smart contracts via graph embedding based bytecode matching. IEEE Trans Inf Forensic Secur 16:2144–2156. https://doi.org/10.1109/TIFS.2021.3050051
27. Gori M, Monfardini G, Scarselli F (2005) A new model for learning in graph domains. In: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005, vol 2. pp 729–734. https://doi.org/10.1109/IJCNN.2005.1555942
28. Denton EL, Zaremba W, Bruna J, LeCun Y, Fergus R (2014) Exploiting linear structure within convolutional networks for efficient evaluation. In: Advances in Neural Information Processing Systems, Cambridge, vol 27
29. Veličković P, Cucurull G, Casanova A, Romero A, Lió P, Bengio Y (2017) Graph attention networks. arXiv:1710.10903
30. Ma R, Jian Z, Chen G, Ma K, Chen Y (2019) Rejection: A AST-based reentrancy vulnerability detection method. In: Chinese Conference on Trusted Computing and Information Security, Springer, pp 58–71
31. Xu X, Liu C, Feng Q, Yin H, Song L, Song D (2017) Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. https://doi.org/10.1145/3133956.3134018
32. Zhuang Y, Liu Z, Qian P, Liu Q, Wang X, He Q (2020) Smart contract vulnerability detection using graph neural network. In: Yokohama, IJCAI, pp 3283–3290
33. Zhou Y, Liu S, Siow J, Du X, Liu Y (2019) Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv Neural Inf Process Syst 32. https://doi.org/10.48550/arXiv.1909.03496
34. Liu Z, Qian P, Wang X, Zhuang Y, Qiu L, Wang X (2021) Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Trans Knowl Data Eng 35(2):1296–310
35. Ferreira JF, Cruz P, Durieux T, Abreu R (2020) Smartbugs: A framework to analyze solidity smart contracts. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), Australia, Virtual Event, pp 1349–1352

Publisher's Note