Smart Contract Vulnerability Detection Based on Dual Attention Graph Convolutional Network

Yuqi Fan[1,2(✉)], Siyuan Shang[1], and Xu Ding[3]

[1] School of Computer Science and Information Engineering, Hefei University of Technology, Hefei 230601, Anhui, China
yuqi.fan@hfut.edu.cn, siyuan0102@mail.hfut.edu.cn
[2] Anhui Provincial Key Laboratory of Network and Information Security, Anhui Normal Unviersity, Wuhu 241002, Anhui, China
[3] Institute of Industry and Equipment Technology, Hefei University of Technology, Hefei 230009, Anhui, China
dingxu@hfut.edu.cn

Abstract. Smart contracts on blockchains have received increasing attention due to the decentralized, transparent, and immutable characteristics of blockchain. However, smart contracts are prone to security problems caused by critical vulnerabilities, which can lead to huge economic losses. Therefore, it is urgent to provide strong and robust security assurance for smart contracts. Most existing studies on smart contract vulnerability detection methods take heavy reliance on experts-defined rules, which are extremely time-consuming and labor-demanding. Moreover, the manually-set rules are limited to specific tasks and subject to errors. Although some studies explore the use of deep learning methods, they fail to represent both semantics and structural information. In this paper, we propose a novel model, Dual Attention Graph Convolutional Network (DA-GCN), to detect vulnerabilities in smart contracts on blockchains. Both control flow graph and opcode sequence extracted from smart contract bytecodes are fed into the feature extractor based on graph convolutional network and self-attention mechanism. Model DA-GCN then uses control flow level attention to focus on the more important nodes in the control flow graph and suppress useless information. Finally, a multi layer perceptron is used to identify whether the smart contract is vulnerable. Experimental results on the real-world smart contract data set containing two vulnerabilities of reentrancy and timestamp dependency demonstrate that our proposed model DA-GCN can effectively improve the performance of smart contract vulnerability detection.

Keywords: Smart contract · Vulnerability detection · Deep learning · Graph convolution · Dual attention

1 Introduction

Blockchain is widely used as an underlying programmable distributed infrastructure to support smart contract applications which can be seen as event-driven

H. Gao and X. Wang (Eds.): CollaborateCom 2021, LNICST 407, pp. 335–351, 2021.
https://doi.org/10.1007/978-3-030-92638-0_20

programs on the blockchain system. Once deployed on the blockchain, smart contracts are not allowed to be modified, due to the decentralized, transparent, and immutable characteristics of blockchain. However, serious vulnerability in smart contracts can be maliciously utilized, leading to huge financial losses. In June 2016, DAO, a large crowdfunding organization deployed on Ethereum, had $150 million worth of Ethereum digital currency stolen. In July 2017, Parity, a widely used multi-signature digital wallet was attacked, which directly resulted in a $30 million financial loss. Therefore, the smart contracts require strong security assurance, and there is an urgent need for effective vulnerability detection of smart contracts, such that we can remove the found vulnerabilities in the smart contracts before deploying them.

Current research on detecting smart contract vulnerability is mainly based on the traditional methods such as symbolic execution [15], fuzzy detection [7], etc. However, adopting traditional methodologies in vulnerability detection demands expert definition of logic rules, which takes heavy reliance on experience, capability of professionality, and understanding of domain knowledge. The design process of logical rules can also consume significant time. Besides, the manually set rules are often limited to specific tasks and prone to errors.

Deep learning methodologies have been used in software defect prediction or malware classification these years and achieved certain results. However, the research on smart contract vulnerability detection using deep learning has not received much attention. Some studies detect smart contracts vulnerabilities via deep learning approaches by taking either sequence-based [6,14,17] or graph-based [20] feature representations as input. Tann et al. [14] first converted the bytecode into an opcode sequence. The bytecode, represented by 32 bytes of hexadecimal numbers, is an intermediate form between high-level and machine languages. The authors then constructed a long short-term memory network (LSTM) model for vulnerability detection in smart contracts. Gogineni et al. [6] improved the work in [14] by using an encoder to predict the next instruction in the opcode sequence, such that the LSTM model continues to learn the parameters for the vulnerability classification task on the pre-trained layers of the encoder. Xing [17] proposed a feature extraction method named "slicing matrix" to divide smart contracts into a sequence of function blocks using jump instructions as the demarcation in the sequence of opcodes; the number of different opcodes within each function block is used as the input feature fed into the neural network for classification. To capture the structural information in smart contracts, Zhuang et al. [20] characterized the source code of smart contracts as a graph structure based on data and control dependencies of program statements, and then constructed a graph neural network and a temporal message propagation (TMP) framework to perform classification.

There is still room for improvement on the performance of smart contract vulnerability detection. The sequence-based methods [6,14] representing each smart contract as a one-dimensional opcode sequence cannot learn the control or dependency relations between instructions. The sequence-based method [17] defining the sequence at the scale of "function blocks" neglects the execution

order of instructions within the function blocks. Graph-based method [20] characterizing the control dependencies of program statements via a graph works at the source code level or control flow level, failing to capture the information in the opcode level. In summary, neither sequence-based nor graph-based feature representations in the existing work can fully retain the rich structural or semantic information in the smart contracts and learn high-dimensional features effectively for classification.

In this paper, we propose a novel smart contract vulnerability detection method based on smart contract bytecodes using a hybrid feature representation by combining control flow and opcode sequence in the Ethereum virtual machine (EVM). The main contributions of this paper are as follows. We generate the attributes of the smart contracts, including the control flow graphs (CFGs) and the EVM opcode sequences. We also propose a Dual Attention Graph Convolutional Network (DA-GCN) to detect vulnerabilities in smart contracts. Model DA-GCN extracts the features from the generated CFGs and opcode sequences using two modules: graph convolutional module and dual-attention module. Graph convolutional module extracts the features of the graph structure using Graph Convolutional Network (GCN) at the scale of nodes of a CFG. The dual-attention module uses self-attention mechanism to extract sequential and associative features between opcode instructions, and further extracts the attention coefficients from a CFG, such that the model can better focus on the important parts of smart contracts in vulnerability detection and suppress useless information. Model DA-GCN then performs classification to identify whether the smart contract is vulnerable, using a Multilayer Perceptron (MLP) and the extracted features from the previous two modules. We obtain the dataset from Ethereum smart contracts containing two vulnerabilities of reentrancy and timestamp dependency, and conduct experiments on the obtained data set. Experimental results demonstrate that the proposed model DA-GCN achieves the accuracy of 91.2% and 87.5% in the two smart contract vulnerability detection task, respectively, and DA-GCN can effectively improve the performance in terms of accuracy, precision, recall, and F1-score on smart contract vulnerability detection.

The rest of the paper is organized as follows. Related works are presented in Sect. 2. The smart contract vulnerability detection problem is described in Sect. 3. Model DA-GCN is proposed in Sect. 4. We experimentally evaluate the proposed model in Sect. 5, and finally we conclude the paper in Sect. 6.

2 Related Works

Blockchain has been applied to various fields like industry, healthcare, supply chain, etc. The creation of smart contracts is a milestone in the history of blockchain. A smart contract, which can be seen as a program running on the blockchain system, automatically performs the execution of transactions once triggered by certain events. After being deployed on the blockchain system, the smart contract cannot be modified or reversed, so there are no remedies available

to deal with code errors. However, attacks on smart contract vulnerabilities have become increasingly frequent and diverse in recent years. Therefore, vulnerability detection is one of the most fundamental tasks to ensure the reliability and security of smart contracts.

Some work on smart contract vulnerability detection relies on symbolic execution and formal verification methods such as Oyente [11] and Securify [15]. Luu et al. [11] proposed a static analysis tool for smart contract vulnerability detection. The tool takes bytecodes with the global state of Ethereum as input, constructs the CFG, and symbolically runs a contract to identify specific vulnerabilities through logical analysis. Tsankov et al. [15] proposed Securify which analyzes the smart contract in Ethereum to show whether the contract is secure in an automatic and scalable way. For each vulnerability property, Securify defines the corresponding secure or insecure smart contract behavior. To detect violation patterns, Securify symbolically analyzes the dependency graph of each contract, extracts attributes of semantic, and examines key code structures, using logical conditions and check for the presence of vulnerability properties. Feist et al. [4] proposed Slither, a multiple-vulnerability detector based on smart contract source code written in Solidity. Slither uses taint analysis to check vulnerabilities in relation to the input and data dependencies. Nikolić et al. [13] proposed a smart contract security tool which uses the runtime trace of a series of calls to capture vulnerabilities, targeting greedy, prodigal, and suicidal contracts. Jiang et al. [7] used fuzzy detection and runtime behavior monitoring to identify vulnerabilities during the execution of a contract. Liu et al. [9] proposed an analysis tool based on fuzzing method called Reguard, which finds reentrancy vulnerability in the contracts. The aforementioned works mainly apply traditional software analysis methods to detect smart contract vulnerabilities. However, vulnerability detection using these techniques demands expert definition of logic rules, which takes heavy reliance on experience, capability of professionality, and domain knowledge. The design of rules for logical statements takes significant time and is prone to errors. Moreover, the manually-set rules in most cases will only be used for specific tasks, which has significant limitations.

Deep learning methodologies have been showing great advantages in end-to-end automated feature learning, enabling better understanding of the intrinsic structure of complex data. There is limited research on smart contract vulnerability detection using deep learning. Tann et al. [14] extracted operand sequence from smart contract bytecodes and then used an LSTM model to detect vulnerabilities. Peng et al. [6] also used an LSTM model for learning sequence information, and used an auxiliary task for predicting operands as pre-training and fine-tuned the model on the vulnerability detection tasks, with an attention mechanism to tackle the problem of long-distance dependency in sequences. Considering that vulnerabilities in local code can show great impact on the overall code, Xing et al. [17] proposed a feature extraction method named "slice matrix". The bytecodes of smart contract are partitioned with RETURN instructions to form a series of matrices, and the number of instructions of each type within the matrix is used as features, which are fed into different classifiers such as CNN for classification. Ashizawa et al. [1] proposed Eth2Vec, which can learn the lexical

and semantics features from the bytecode of a smart contract in the EVM, and they used similarity checking and code embedding to decide whether a smart contract is vulnerable. Zhuang et al. [20] represented the source code of smart contracts in the form of directed graphs and constructed a graph neural network DR-GCN and a time-based propagation framework for vulnerability detection. Liu et al. [10] designed specific patterns for different vulnerabilities and combined features learned from the graph neural network with the expert patterns to get the final analysis of the target smart contract.

Note that the sequence-based or graph-based feature representation methods introduced above lose certain structural and semantic information in the smart contract, and thus fail to learn high-dimensional features for vulnerability detection. In this paper, we use a hybrid feature representation and propose a novel model DA-GCN for smart contract vulnerability detection, which learn the features from both control flow graphs and opcode sequences.

3 Problem Description

In this paper, we study how to detect whether a smart contract is vulnerable based on the smart contract bytecode. Specifically, we investigate two types of vulnerabilities, i.e. Reentrancy and Timestamp Dependency.

Reentrancy is a vulnerability of great concern existing in Ethereum smart contracts, which was exploited by attackers in the DAO hack with a total loss beyond 60 million dollars. An Ethereum smart contract can call another smart contract during execution, and the smart contract being called must wait until the call is finished. The recipient of the call can exploit this intermediate status to steal digital currency. Example 1 shows a real-world attack event using reentrancy.

Example 1. There is a special mechanism in smart contract system called fallback function which has no function name and takes no arguments. The fallback function will be invoked in two scenarios when the function call does not match any functions in the called smart contract or the ether (the dedicated cryptocurrency used in Ethereum) is received by the caller. There are two contracts in Fig. 1. Contract Attacker attempts to steal the ether from contract Victim, a simplified version of digital wallet with a reentrancy vulnerability, by exploiting the fallback mechanism. To be specific, Attacker executes its attack() function to invoke withdraw() function in Victim. Victim will then transfer certain amount of ether to Attacker. Once the ether is received, the fallback function in Attacker will be executed. As we can see, the balance of Attacker's account has not yet been set to 0 by Victim at the time, the Attacker can hence repeatedly invoke the withdraw() method until the ether held by Victim drops to 0.

Timestamp dependency is another well-known Ethereum smart contract vulnerability which is related to timestamp in the blockchain system. When a smart contract takes the block timestamps as a dependency condition to trigger curtain critical operation like transfer ether, some malicious miners may manipulate it

```
1 contract Victim{
2
3    mapping(address=>uint) userBalances;
4
5    function withdraw() {
6      uint amount = userBalances[msg.sender];
7      msg.sender.call.value(amountToWithdraw)();
8      userBalances[msg.sender] = 0;
9    }
10 }
```

```
1 contract Attacker{
2
3    function attack(address addr) {
4        victim(addr).withdraw();
5    }
6
7    function () payable{
8        victim(addr).withdraw();
9    }
10 }
```

Fig. 1. Example of a smart contract with the reentrancy vulnerability.

to modify the timestamp for illegal benefits. Example 2 illustrates a real-world contract which suffers from the timestamp dependency vulnerability.

Example 2. Contract theRun, shown in Fig. 2, uses a set of rules based on the current block timestamp to choose who will win the bonus. In contract theRun, the variable h defined in line 7 is the hash value of a certain block in the blockchain, which is used to decide the winner. The selection of a block is determined by the variable seed defined in line 6. Three variables decide the seed value, including block number, last payout, and timestamp. Among them, block number and last payout are determined values recorded on the blockchain, while timestamp is decided by the miner. Normally, the timestamp is set as the current time of the miner's local system. However, the miner can vary the timestamp value by roughly 900 s, while still having other miners accept the block. Therefore, by choosing the timestamp, the miner can calculate the result in advance and manipulate the outcome of timestamp-dependent contracts to get the final bonus.

```
1   contract theRun {
2      uint private Last_Payout = 0;
3      uint256 salt = block.timestamp;
4      function random returns(uint256 result){
5          uint256 y = salt * block.number/(salt%5);
6          uint256 seed = block.number/3 + (salt%300) + Last_Payout + y;
7          uint256 h = uint256 (block.blockhash(seed));
8          return uint256 (h % 100) + 1;
9      }
10  }
```

Fig. 2. Example of a smart contract with the timestamp dependency vulnerability.

4 Smart Contract Vulnerability Detection Method

We first generate the attributes of smart contracts, including the CFGs and the EVM opcode sequences. We then propose a novel model, Dual Attention Graph Convolutional Network (DA-GCN), to detect vulnerabilities in smart contracts. Model DA-GCN extracts the features from the generated CFGs and opcode sequences, and identifies whether the smart contract is vulnerable by combining the extracted features.

4.1 Attributes Generation

To better express the rich semantic and structural information in smart contracts, we generate both control flow graphs and opcode sequences from smart contract bytecodes as attributes. The workflow of attributes generation process from smart contract bytecodes is shown in Fig. 3.

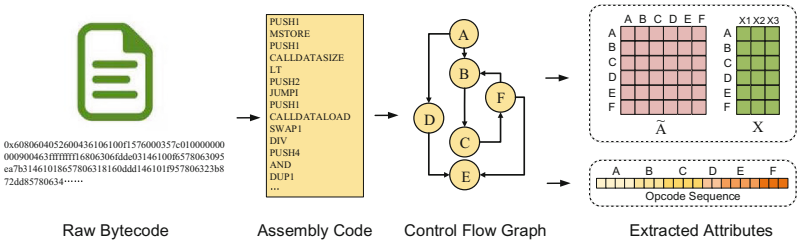

Fig. 3. Workflow of attributes generation process.

Control Flow Graph. We represent the smart contract bytecodes as CFGs. The reasons for using CFGs to characterize contracts are as follows. Firstly, CFGs use graph structures to explicitly express the logic of smart contracts. A CFG represents all the paths that can be traversed during the execution of a smart contract, which can well indicate the structural dependencies of smart contracts. We can also extract the CFGs from contract codes in different forms. Secondly, the CFG contains various information that can be used as attributes in deep learning-based smart contract vulnerability detection, e.g. n-grams, opcodes, and structural information [18].

We utilize the state-of-the-art tool evm-cfg-builder [2], which has been used in several applications including Ethersplay and Manticore [12], to extract CFGs from bytecodes. Every node in a CFG represents a basic-block consisting of a sequence of EVM opcodes. The basic blocks are independent of each other. That is, each basic block can only be entered at the first instruction and exited at the last instruction. Two neighboring basic blocks are connected by a directed edge. In this graph structure, we put the basic blocks in the order in which they appear in the EVM bytecode.

The complete EVM instruction set contains more than 150 instructions. Among them, we select totally 74 instructions, which can best reflect the logic and semantics of smart contract execution process and relate to the smart contract vulnerability detection tasks. For example, the jump instruction can indicate the switch from the current basic block to other basic blocks; the block, timestamp, call, callvalue instructions can relate to the critical part of vulnerable contracts as shown in Examples 1 and 2.

The selected instructions are categorized into 7 types: Arithmetic, Environment, Blockchain Information, Stack, System, Logic, and Cryptograph. We take the numbers of instructions in different instruction types as the attributes of each basic block, and the attributes will be fed into the neural network for feature extraction.

Opcode Sequence of Ethereum Virtual Machine. Considering that CFGs working at the basic block level cannot capture certain finer-grained information, we further extract the opcode sequences in each basic block and concatenate them together in the same order as the basic blocks appear. Different contracts may vary in the length of opcode sequences. To obtain the input of the neural network, the sequence lengths are fixed to the same value. Specifically, we pad 0 to the sequences shorter than the fixed-length and truncate the exceeding part of the sequences longer than the fixed-length. We randomly select 1000 Ethereum smart contracts and count the lengths of their opcode sequences, among them 91.5% of smart contracts are with the opcode sequence length within 2000. Therefore, we choose 2000 as the fixed-sequence length. Before feeding each fixed-length opcode sequence into the neural network, we map each opcode to a unique integer.

4.2 Construction of DA-GCN Model

The DA-GCN model shown in Fig. 4 consists of three modules: graph convolutional module, dual-attention module, and classification module. The graph convolutional module uses graph convolutional layers to extract features at the scale of basic blocks from the CFG. In the dual-attention module, we first use the self-attention mechanism to extract sequential and associative features between opcode instructions, and further extract attention coefficients from the CFG to better focus on the more important basic blocks in vulnerability detection task and suppress useless information. The final obtained high-dimensional features will be concatenated and fed into the classification module.

Graph Convolutional Module. We use graph convolutional layers at the control flow level to propagate each basic block's features to the neighbor blocks based on the structural connectivity, extract the features of the graphs, and learn the structural information in the smart contracts.

A CFG is a graph with directed edges. For a graph G with N nodes, we denote its corresponding adjacency matrix as $A \in \mathbf{R}^{N \times N}$. Since the features in a node cannot be propagated to itself using this adjacency matrix, we define an augmented adjacency matrix $\tilde{A} = A + I$. Accordingly, we denote this augmented diagonal degree matrix as $\tilde{D}$, which can be calculated from $\tilde{A}$:

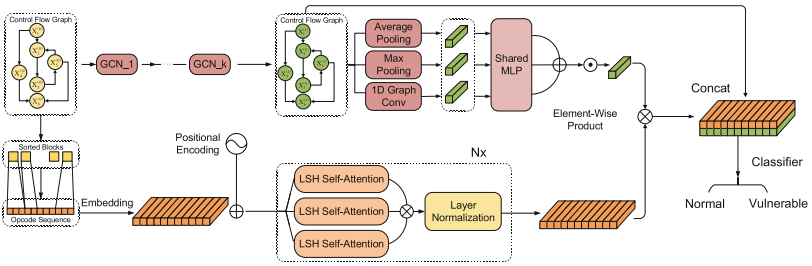

Fig. 4. Diagram of model DA-GCN.

$$\tilde{D}_{u,u} = \sum\nolimits_{v} \tilde{A}_{u,v} \tag{1}$$

where $\tilde{D}_{u,u}$ denotes the element (u, u) in augmented degree matrix $\tilde{D}$, and $\tilde{A}_{u,v}$ is the element (u, v) in augmented adjacency matrix $\tilde{A}$.

Assuming each node in graph G has c attributes, we denote the matrix of attributes for graph G as $X \in \mathbf{R}^{N \times c}$.

We use multiple staked graph convolutional layers in the graph convolutional module. The propagation formula of the graph convolutional layer [19] can be defined as follows:

$$Z^{t+1} = f(\tilde{D}^{-1}\tilde{A}Z^{t}W^{t}) \tag{2}$$

where f denotes the activation function. $Z^{t} \in \mathbf{R}^{N \times c_t}$ is the output of the t-th graph convolutional layer, and $Z^{0} = X$. c_t is the number of output channels of the t-th graph convolutional layer, and $W^{t} \in \mathbf{R}^{c_t \times c_{t+1}}$ denotes the parameters mapping from c_t channels to c_{t+1} channels.

Dual Attention Module. In this module, we calculate the attention coefficients at the opcode level and control flow level, respectively.

Extraction of Opcode Level Attention. We first apply word embedding to the opcode sequence so that the instructions mapped as different integers are transformed into fixed-size real value vectors. The parameters of word embedding are randomly initialized and will then be updated during the training process.

To exploit the ordering information of the opcode sequence, the position of each instruction is encoded in the sequence. The position encoding is added to the embedding vector and defined as:

$$\begin{cases} PE_{(i,2p)} = \sin(i/10000^{2p/d_{\text{embedding}}}) \\ PE_{(i,2p+1)} = \cos(i/10000^{2p/d_{\text{embedding}}}) \end{cases} \tag{3}$$

where i denotes the i-th position in an opcode sequence. $2p$ and $2p+1$ represent the $2p$-th and $(2p+1)$-th dimensions in the embedding vector of an opcode, respectively. $d_{\text{embedding}}$ is the dimension of the embedding vector of an opcode. $PE_{(i,2p)}$ and $PE_{(i,2p+1)}$ are the encoding values of the $2p$-th and $(2p+1)$-th

dimensions in the embedding vector of the i-th position in an opcode sequence, respectively. That is, every dimension in position encoding matches a sinusoidal signal whose wavelength grows geometrically from 2π to $20{,}000\pi$.

Long-distance dependency problem occurs when attention is used to process sequence-structured data. Transformer [16], a self-attention mechanism, uses the scaled dot-product attention to solve the long-distance dependency problem and also improves the computational speed of the model. However, Transformer will consume huge memory resources when dealing with long sequences. Locally Sensitive Hashing (LSH) self-attention mechanism [8] can address the memory-demanding problem of Transformer. LSH is a hashing scheme that can find nearest neighbors quickly in a high-dimensional space, where nearby vectors get the same hash value with high probability and distant ones do not. Therefore, we adopt LSH self-attention mechanism to extract the opcode level attention.

In order to use LSH self-attention, we write the attention equation in Transformer as:

$$\alpha_i = \sum_{j \in P_i} \frac{\exp(q_i \cdot k_j - z(i, P_i))v_j}{\sqrt{d}} \tag{4}$$

where i and j denote the i-th position and j-th position in an opcode sequence, respectively. α_i is the attention coefficient of position i. q_i, k_i, and v_i are query, key, and value vectors at position i, respectively. d is the dimension of each vector of q_i, k_i, and v_i. P_i denotes all the queries which the query at position i can attend to, and z denotes softmax which is used as the partition function.

We filter the keys into a hash bucket using a hash function $H(\bullet)$ from the queries which the query at position i can attend to:

$$P_i = \{j : H(q_i) = H(k_j)\} \tag{5}$$

The queries and keys at different positions are grouped into different hash buckets via Eq. (5). Note that the number of keys and queries may not be balanced in different buckets. Therefore, we set $k_j = \frac{q_j}{\|q_j\|}$.

We then sort the queries by the buckets which include the queries, and divide all the queries into several parts, each containing the same number of queries. We call each part a *chunk*. Obviously, the queries in the same bucket may fall into different chunks. Therefore, we repeat N^H rounds of hashing, each round with a different hash function H^r, where r denotes the r-th round of hash calculation:

$$P_i = \bigcup_{r=1}^{N^H} P_i^r \qquad \text{where } P_i^r = \{j : H^r(q_i) = H^r(q_j)\} \tag{6}$$

Extraction of Control Flow Level Attention. Different from the self-attention mechanism between different opcodes, the control flow level attention, shown in Fig. 5, focuses on basic blocks that are more important for the vulnerability classification by assigning them greater weights.

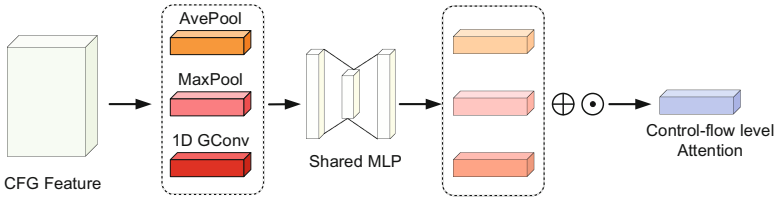

Fig. 5. Diagram of control flow level attention.

We take the CFG feature map obtained by performing m graph convolutional layers on the control flow graph as the input. We apply average-pooling and max-pooling on the channels of each node to better describe the relationship of c_m channel attributes. Moreover, to better aggregate the attributes of each node and its neighboring node, we apply a one-dimensional graph convolutional layer as follows:

$$V = f(\tilde{D}^{-1}\tilde{A}Z^m W^m) \tag{7}$$

where V is the output of the one-dimensional graph convolutional layer. $Z^m \in \mathbf{R}^{N\times c_m}$ denotes the output of the m-th graph convolutional layer, c_m represents the number of output channels of the m-th graph convolutional layer, and $W^m \in \mathbf{R}^{c_m\times 1}$ is a one-dimensional vector with c_m elements.

The features extracted from the average-pooling, max-pooling, and one-dimensional graph convolutional layer are then fed into a MLP with a hidden layer to produce the attention coefficients on the control flow level, which can be calculated as follows:

$$\begin{cases} \mathrm{ATT}_{ave} = \sigma(W_1(W_0(\mathrm{AvgPool}(Z^m)) + b_0) + b_1) \\ \mathrm{ATT}_{max} = \sigma(W_1(W_0(\mathrm{MaxPool}(Z^m)) + b_0) + b_1) \\ \mathrm{ATT}_{gcn} = \sigma(W_1(W_0(\tilde{D}^{-1}\tilde{A}Z^m W^m) + b_0) + b_1) \\ \mathrm{ATT}_{block} = \sigma(\mathrm{ATT}_{ave} + \mathrm{ATT}_{max} + \mathrm{ATT}_{gcn}) \end{cases} \tag{8}$$

where $W_0 \in \mathbf{R}^{d_{\text{hidden}}\times c_m}$, $b_0 \in \mathbf{R}^{d_{\text{hidden}}\times 1}$, $W_1 \in \mathbf{R}^{c_m\times d_{\text{hidden}}}$, and $b_1 \in \mathbf{R}^{c_m\times 1}$. σ is the sigmoid function, and d_{hidden} denotes the dimensions of the hidden layer in MLP.

Classification Module. In this module, we concatenate the features extracted from the graph convolutional module and dual attention module, and then perform classification via convolutional layer, max pooling, fully connected layer, and softmax function to identify whether the smart contract is vulnerable.

5 Experiment

5.1 Experimental Settings

Dataset and Benchmarks. We integrate the real-world vulnerability smart contracts from three sources, including the dataset in SolidiFi [5], the dataset

used in [20], and Ethereum smart contracts verified by SmartBugs [3]. In our dataset consisting of 2216 labeled smart contracts, there are 496 smart contracts with reentrancy vulnerability, 768 smart contracts with timestamp dependency vulnerability, and the remaining 952 contracts are free from reentrancy or timestamp dependency vulnerability. We further divide our dataset into two parts targeting the smart contract vulnerabilities of reentrancy and timestamp dependency, respectively.

In the experiments, we use cross-validation to evaluate our model. We first randomly divide the dataset into 10 folds, each containing 10% of the samples in the dataset. 9 of the 10 folds are used for training, and the remaining one is used as the test set. One fold in the training set is marked as the validation set. We use the validation set to validate the model after each epoch, and the model achieving the highest accuracy during the training process is saved and evaluated on the test set to obtain the final result.

The numbers of samples in training, validation, test, and total datasets on reentrancy and timestamp dependency are shown in Table 1 and Table 2, respectively.

Table 1. Dataset of reentrancy vulnerability.

	Training set	Validation set	Test set	Total
Reentrancy	396	50	50	496
Normal	760	91	91	952

Table 2. Dataset of timestamp dependency vulnerability.

	Training set	Validation set	Test set	Total
Timestamp	614	77	77	768
Normal	760	91	91	952

We compare the proposed model DA-GCN with the state-of-art smart contract detection vulnerability detection model TMP [20] and the state-of-art deep learning models dealing with sequences or graph structures, including BiLSTM-Attention (BLSTM-ATT), Transformer [16], and DGCNN [19]. BLSTM-ATT and Transformer learn the features from opcode sequences of smart contracts, while DGCNN learns from the control flow information of smart contracts.

For each method, 30 10-fold cross-validations are run and the average of the 30 results is taken as the final result.

Performance Metrics. We evaluate our proposed model DA-GCN in terms of 4 performance metrics: accuracy, recall, precision, and F1-score, assuming TP, FP, FN, and TN are the numbers of true positive samples, false positive samples, false negative samples, and true negative samples, respectively.

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

$$\text{accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{9}$$

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

$$\text{precision} = \frac{TP}{TP + FP} \tag{10}$$

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

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

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

$$\text{F1} - \text{score} = \frac{2 \times \text{precison} \times \text{recall}}{\text{precison} + \text{recall}} \tag{12}$$

5.2 Experimental Results

Table 3 lists the accuracy, recall, precision, and F1-score performance of DA-GCN, BLSTM-ATT, Transformer, DGCNN, and TMP on reentrancy and timestamp dependency vulnerability detection tasks, respectively. The experimental results demonstrate that model DA-GCN achieves the best performance in terms of all the four evaluation metrics.

We first compare model DA-GCN with BLSTM-ATT, Transformer, and DGCNN. The classic LSTM model suffers from gradient disappearance and gradient explosion when propagating over long distances. Although BLSTM-ATT compensates the problem to a certain extent by using a bidirectional model and the attention mechanism, BLSTM-ATT still fails to achieve the ideal results when dealing with longer sequences. The experimental results show that Transformer can better capture the long-range association of sequences and achieve better results than BLSTM-ATT due to the distance between two arbitrary elements being 1, when applying self-attention mechanism in Transformer.

The control flow graph of a smart contract reflects the invocation relationship of basic blocks and the logical instruction structure of smart contracts, while the EVM opcode sequence focuses on the execution mode and the order

Table 3. Performance in terms of accuracy, recall, precision and F1-score on each vulnerability detection task.

Methods	Reentrancy				Methods	Timestamp dependency			
	Acc (%)	Recall (%)	Precision (%)	F1 (%)		Acc (%)	Recall (%)	Precision (%)	F1 (%)
BLSTM-ATT	83.92	71.25	74.41	72.80	BLSTM-ATT	78.69	72.09	80.52	76.07
Transformer	86.03	76.00	80.36	77.55	Transformer	83.72	78.96	82.41	80.13
DGCNN	88.89	82.86	74.36	78.38	DGCNN	84.25	76.29	76.88	76.44
TMP	89.87	80.80	87.46	83.46	TMP	85.33	80.27	85.50	82.81
DA-GCN	**91.15**	**82.00**	**89.84**	**85.43**	**DA-GCN**	**87.54**	**82.85**	**87.15**	**84.83**

of instructions in smart contracts at the opcode level. Both opcode sequences and control flows play an important role in smart contract vulnerability detection. Separate analyses of control flow graphs and opcode sequences may lose certain semantic and structural information. The proposed model DA-GCN uses a hybrid model to jointly learn the control flow graph and opcode sequence features. To be specific, the instructions are assembled in the form of basic blocks, resulting in opcodes within the same block and between different blocks have different associations. Model DA-GCN further implements attention at the scale of control flow level, which enables the network to focus on some basic blocks that are more useful for vulnerability detection and suppress information that is not useful for the classification tasks.

The experimental results in Table 3 also show that model DA-GCN that integrates the control flow level and opcode level features improves the accuracy by 3.25% and 3.82% on the two different detection tasks, respectively, compared to the best results achieved by DGCNN and Transformer, which only consider the features from either of the two levels.

We then compare model DA-GCN with TMP which is the state-of-the-art method used for smart contract vulnerability detection. As shown in Table 3, DA-GCN improves the accuracy by 1.28% and 2.21% on the two vulnerability detection tasks, respectively.

We also evaluate the effectiveness of each model using the receiver operating characteristic (ROC) curve. ROC is a plot regarding True Positive Rate (TPR) versus False Positive Rate (FPR) at different thresholds. To quantitatively measure the ROC curve and evaluate the performance of the classifier, we further adopt the area under the ROC curve (AUC) to measure the capability of the classifier to discriminate positive and negative samples. The range of AUC values is [0.5, 1]. The ROC curves and AUC values of different models targeting reentrancy and timestamp dependency vulnerabilities are shown in Fig. 6. It can be seen that the proposed model DA-GCN performs the best among the different models and achieves the AUC of 0.94 and 0.92 on the two vulnerability detection tasks, respectively.

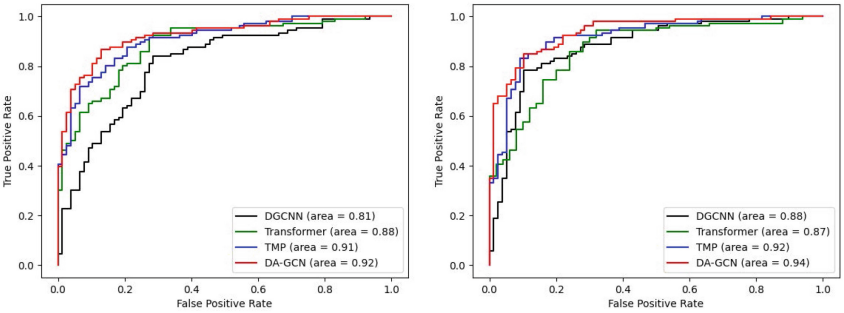

Fig. 6. (a) ROC curves for different methods on reentrancy vulnerability detection task. (b) ROC curves for different methods on timestamp dependency vulnerability detection task.

6 Conclusion

In this paper, we tackled the problem of smart contract vulnerability identification based on smart contract bytecodes. We generated the attributes of smart contracts, including the control flow graphs and the EVM opcode sequences. We also proposed a Dual Attention Graph Convolutional Network (DA-GCN) to detect vulnerabilities in smart contracts. Model DA-GCN extracts the features from the generated CFGs and opcode sequences using two modules: graph convolutional module and dual-attention module. Graph convolutional module extracts the features of the graph structure using graph convolutional network at the scale of basic blocks of a control flow graph. The dual-attention module uses self-attention mechanism to extract the sequential and associative features between opcode instructions, and further extracts the attention coefficients from a control flow graph, such that the model can better focus on the important basic blocks of smart contracts in vulnerability detection and suppress useless information. Model DA-GCN then performs classification to identify whether the smart contract is vulnerable, by combining the extracted features from the previous two modules and using a Multilayer Perceptron. We obtained the dataset from real-world Ethereum smart contracts containing two vulnerabilities: reentrancy and timestamp dependency. Experimental results demonstrated that the proposed model DA-GCN achieved an accuracy of 91.2% and 87.5% in the two smart contract vulnerability detection tasks, respectively, and DA-GCN could effectively improve the performance in terms of accuracy, precision, recall, and F1-score on smart contract vulnerability detection.

Acknowledgment. This work was partly supported by the open project of State Key Laboratory of Complex Electromagnetic Environment Effects on Electronics and Information System in China (CEMEE2018Z0102B), the Fundamental Research Funds for the Central Universities (PA2021GDSK0095), and the open fund of Intelligent Interconnected Systems Laboratory of Anhui Province (PA2021AKSK0114), Hefei University of Technology.

References

1. Ashizawa, N., Yanai, N., Cruz, J.P., Okamura, S.: Eth2Vec: learning contract-wide code representations for vulnerability detection on Ethereum smart contracts. In: Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure, pp. 47–59 (2021)
2. evm-cfg builder (2017). https://github.com/crytic/evm_cfg_builder
3. Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 530–541 (2020)
4. Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 8–15. IEEE (2019)
5. Ghaleb, A., Pattabiraman, K.: How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 415–427 (2020)
6. Gogineni, A.K., Swayamjyoti, S., Sahoo, D., Sahu, K.K., Kishore, R.: Multi-class classification of vulnerabilities in smart contracts using AWD-LSTM, with pre-trained encoder inspired from natural language processing. IOP SciNotes **1**(3), 035002 (2020)
7. Jiang, B., Liu, Y., Chan, W.: ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 259–269. IEEE (2018)
8. Kitaev, N., Kaiser, Ł., Levskaya, A.: Reformer: the efficient Transformer. In: Proceedings of the International Conference on Learning Representations (2020)
9. Liu, C., Liu, H., Cao, Z., Chen, Z., Chen, B., Roscoe, B.: ReGuard: finding reentrancy bugs in smart contracts. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pp. 65–68. IEEE (2018)
10. Liu, Z., Qian, P., Wang, X., Zhuang, Y., Qiu, L., Wang, X.: Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Trans. Knowl. Data Eng. 1–1 (2021)
11. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254–269 (2016)
12. Mossberg, M., et al.: Manticore: a user-friendly symbolic execution framework for binaries and smart contracts. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1186–1189. IEEE (2019)
13. Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 653–663 (2018)
14. Tann, W.J.W., Han, X.J., Gupta, S.S., Ong, Y.S.: Towards safer smart contracts: a sequence learning approach to detecting security threats. arXiv preprint arXiv:1811.06632 (2018)
15. Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67–82 (2018)

16. Vaswani, A., et al.: Attention is all you need. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, pp. 6000–6010 (2017)
17. Xing, C., Chen, Z., Chen, L., Guo, X., Zheng, Z., Li, J.: A new scheme of vulnerability analysis in smart contract with machine learning. Wirel. Netw., 1–10 (2020, in press)
18. Yan, J., Yan, G., Jin, D.: Classifying malware represented as control flow graphs using deep graph convolutional neural network. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 52–63. IEEE (2019)
19. Zhang, M., Cui, Z., Neumann, M., Chen, Y.: An end-to-end deep learning architecture for graph classification. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32 (2018)
20. Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., He, Q.: Smart contract vulnerability detection using graph neural networks. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, pp. 3283–3290 (2020)