Smart Contract Vulnerability Detection Based on Generative Adversarial Networks and Graph Matching Networks

Hao Li[1[0009−0001−2086−2930]], Xu Wang[1[0000−0001−9439−6437]], Guangsheng Yu[2[0000−0002−6111−1607]], Wei Ni[2[0000−0003−0780−4637]], Ren Ping Liu[1[0000−0001−7001−6305]], Nektarios Georgalas[3], and Andrew Reeves[3]

[1] University of Technology Sydney, Sydney NSW 2007, Australia
hao.li-7@student.uts.edu.au, xu.wang-1@uts.edu.au, renping.liu@uts.edu.au
[2] Data61, CSIRO, Sydney NSW 2015, Australia
saber.yu@data61.csiro.au, wei.ni@data61.csiro.au
[3] Applied Research, British Telecom, Martlesham, UK
nektarios.georgalas@bt.com, andrew.reeves@bt.com

Abstract. With Blockchain technology's tamper-proof and decentralized characteristics, smart contracts have been developed rapidly for wide application in critical areas, e.g., the Internet of Things, digital management, healthcare and finance. However, the security vulnerabilities of smart contracts have led to significant economic losses. Once deployed on the blockchain, smart contracts cannot be modified, making pre-deployment vulnerability detection crucial. We focus on Ethereum-based smart contracts and innovatively propose a detection method based on Generative Adversarial Networks (GAN) and Graph Matching Networks (GMN) to uncover vulnerabilities, which greatly impact the security of smart contracts. The GAN will be used to solve the data starvation in training, which is one of the serious obstacles in building the deep learning model. Specifically, we convert smart contracts' Solidity code into graphs that contain semantic and structural information. Based on this, we use GAN based on graph representation to augment the small-sample training set into a large-sample training set. The augmented training set will be used to train GMN, which is an extension of Graph Neural Networks. The novel GMN model uses a cross-graph attention mechanism to calculate the feature similarity between the target and vulnerable contracts. The experimental results show that our model has high accuracy and efficiency in detecting reentrancy and integer overflow.

Keywords: Blockchain · Smart contract · Graph matching network · Generative adversarial network · Graph representation · Vulnerability detection.

1 Introduction

Blockchain technology [1] is a technical solution that does not rely on a third party, but uses its distributed nodes to store, verify, transmit and communicate network data. Smart contracts are automatic rules based on blockchain

technology, which have a wide range of applications in important fields such as finance, digital management, healthcare and the Internet of Things [2]. In various research works related to smart contracts, most of them focus on smart contracts running on the Ethereum platform [3]. However, the whole process of smart contracts, from generation to execution, faces different security threats and vulnerabilities [4]. With the increasing popularity of Ethereum applications, the financial value of smart contracts also rises, leading to greater prominence of associated security issues. For example, in 2016, the Decentralized Autonomous Organization (DAO) was attacked [5]. The attacker exploited the split function vulnerability and realized multiple token withdrawals by calling an Ethereum smart contract only once, and transferred the currency of the DAO target main chain to a DAO sub-chain. As a result, this attack led to more than 3 million Ether assets being separated from the asset pool. In addition, in 2017, due to vulnerabilities in the wallet and sol multi-signature contract, 150,000 Ether in the Parity wallet were stolen [4]. In 2018, the Beauty Chain project became zero instantly due to security vulnerabilities [4]. The total value of the project was about 10 billion US dollars.

The vulnerabilities in smart contracts can be categorized into three levels: Solidity code layer, Ethereum Virtual Machine (EVM) execution layer and blockchain system layer [6]. Our research aims to analyze and detect vulnerabilities present in the code layer. In this paper, we provide a detailed analysis using reentrancy [7] and integer overflow [7] as illustrative examples in the experimental section. These two vulnerabilities are relatively common and can cause serious security problems. Reentrancy refers to the situation where the contract performs a transfer operation before updating the data, instead of following the standardized process of modifying the Storage variable first and then transferring. If EVM allows an external contract to initiate a new call to the called contract before the initial call ends, reentrancy will occur. This vulnerability can cause the contract state to change during execution. Integer overflow refers to the situation where the value of an integer in a smart contract exceeds the maximum value that its type can represent, causing the value to be truncated and become the minimum value, resulting in numerical errors. Attackers can exploit this vulnerability to perform improper operations, such as modifying data and stealing assets.

At present, the existing vulnerability detection methods have some limitations. Symbolic execution, represented by Oyente [8], can simulate the execution process of contracts by using symbolic values, finding the input values or value ranges that trigger the vulnerability, and thus digging out potential vulnerabilities in contracts. However, its efficiency and accuracy in detecting vulnerabilities in multi-layer data sets are low. Fuzzing represented by sFuzz [9] can randomly generate a large number of normal and abnormal inputs for contracts during the contract execution process, and detect vulnerabilities by monitoring the contract state and behavior. However, it cannot automate the testing of contracts without source code or interface information. Graph Neural Networks (GNN) such as the Combination of Graph Features and Expert Patterns (CGE) [10],

can extract graph structural features such as nodes, edges, and attributes from contract code. Subsequently, it combines expert knowledge patterns, such as common vulnerability patterns and security rules, with the graph features. Finally, it employs a Graph Neural Network model to learn embeddings of the graph features and expert knowledge patterns, yielding vulnerability labels and scores as outputs. However, building a detection model based on GNN requires a substantial amount of training data; otherwise, it may impact the model's generalization ability and detection efficiency.

It should be emphasized that there are some challenges in obtaining a large number of smart contract source codes that meet the experimental requirements. First, research has found that only 1% of smart contracts are open-source [11]. Second, obtaining real vulnerability data may involve privacy protection and legal compliance issues. At the same time, due to the limitations of the Ethereum network and the number of nodes, obtaining a large number of source codes that meet the actual needs may require a lot of time and resources [12]. In addition, to ensure data quality and security, manual data screening is necessary. However, manual screening requires a lot of time and resources when dealing with large sample data sets. These situations may lead to insufficient data for experiments. It should be noted that insufficient experimental data may lead to a decline in the accuracy and generalization ability of the detection model [13].

To address these challenges, we propose a detection method based on the Generative Adversarial Network (GAN) [14] and the Graph Matching Network (GMN) [15]. Since graph representation has more capabilities to preserve the code's structural and semantic features [6], we use graphs to represent the Solidity code of smart contracts. Given that traditional GANs are not suitable for augmenting graph data, we use GraphGAN [15] in model construction, which can learn graph representation and generate synthetic graph data. Graph Neural Networks (GNN) have been proven effective in learning and classifying graph data. GMN adds a cross-graph attention mechanism [15] on top of GNN, thus having better graph similarity judgment ability. GraphGAN will be used to solve the data starvation in training, which is one of the serious obstacles in building the GMN model. Specifically, we first extract the Abstract Syntax Tree (AST) [16], Control Flow Graph (CFG) [16] and Data Flow Graph (DFG) of Solidity code, and integrate them into a contract graph that contains semantic and structural information. Using GraphGAN to augment the small sample training set with synthetic data, GMN can obtain sufficient contract graphs for similarity training. After the training is completed, we use the GMN model to detect target contracts, and judge whether there are vulnerabilities through the similarity feedback of the model. The experiments demonstrate that the method effectively reduces the data requirement and ensures high accuracy and efficiency of detection. Our main contributions include the following points:

- We innovatively propose a smart contract vulnerability detection method that combines GAN and GMN, which can effectively identify reentrancy and integer overflow vulnerabilities.

- We use the graph representation-based GAN model to generate a large number of synthetic contract graphs, achieving the goal of deep learning with a small-sample training set. This approach effectively addresses the issue of data starvation caused by the difficulty in obtaining smart contract source code.
- We conducted experiments on 300 public Ethereum smart contracts, and the results show that our method has high accuracy and efficiency.

The remaining sections of this paper are organized as follows. We conduct a survey of related work in Section 2. Section 3 outlines the methodology of the detection model. In Section 4, we conduct experiments and analyze the experimental results. Finally, we conclude this paper in Section 5.

2 Related Works

It is significant to study reasonable and effective detection schemes in the face of vulnerabilities that pose serious threats to the execution and credibility of smart contracts. In recent years, various vulnerability detection methods for smart contracts have been developed, which can be classified into fuzzing, symbolic execution, and Graph Neural Networks according to their technical characteristics. This section will analyze the above three techniques in detail.

2.1 Fuzzing

Fuzzing is a dynamic analysis technique that detects vulnerabilities or anomalies by inputting invalid, unexpected or random data into smart contracts [9]. The key of this technique lies in how to generate different types of random inputs. Currently, there are mainly two ways to deal with this problem: generation-based and mutation-based. Fuzzing has better scalability and usability than other techniques, and can mine vulnerabilities in the case of black boxes. The execution process of this technique starts with preprocessing the contract. Preprocessing includes building Control Flow Graphs and collecting function dependencies. Then, input data is generated by strategies such as random mutation and coverage-based feedback. During contract execution, relevant information such as coverage, exceptions and state changes are collected. Finally, this information is used to identify vulnerabilities in the contract.

sFuzz is a representative tool that uses fuzzing to detect security defects in Ethereum smart contracts. It generates transaction inputs based on contract bytecode and executes them in an instrumented EVM. By collecting EVM feedback, sFuzz can detect vulnerabilities such as reentrancy, integer overflow, and call stack overflow. sFuzz combines the advantages of fuzzing and symbolic execution, and can cover more code and discover vulnerabilities that are difficult to detect by black-box fuzzing, such as the vulnerabilities in DAO and Parity Wallet. However, sFuzz may have false positives and false negatives [4], and may not be able to handle some complex semantic features. It needs preprocessing or postprocessing to improve testing efficiency and accuracy.

2.2 Symbolic Execution

Symbolic execution simulates the execution process of contracts by using symbolic values, and finds the exact input values or value ranges that trigger vulnerabilities, thus finding potential vulnerabilities in contracts [8]. Smart contracts have small code sizes and relatively few paths, which are quite suitable for symbolic execution. Symbolic execution first symbolizes the inputs in the source code, transforming the input values into indeterminate symbolic vectors, and then uses a symbolic execution engine to solve all paths of the contract, updating the state in the path and exploring the constraints in the path until all executable paths in the contract are explored.

Oyente is one of the earliest research works that apply symbolic execution to smart contract vulnerability detection. It simulates the EVM to explore all executable paths of the smart contract, thus detecting whether there are security problems in the contract, and outputs problematic symbolic paths to users. Oyente consists of three core modules: Control Flow Graph generator, explorer and analyzer. It uses a Z3 solver to determine whether a path is executable during the entire execution process. Oyente can detect smart contract vulnerabilities such as timestamp dependency, reentrancy, integer overflow and unchecked return value [4].

2.3 Graph Neural Networks

Graph Neural Networks (GNN) are a kind of deep learning method based on graph structures [17]. The working principle of GNN is to represent the nodes and edges in the graph with vectors, then update the vectors through multilayer information propagation and aggregation, and finally obtain the vector representation of the graph. In smart contract vulnerability detection, GNN can convert the source code into a contract graph that contains semantic and structural information, and extract graph features from the graph, which are used to detect whether there are vulnerabilities in the contract.

Combination of Graph Features and Expert Patterns (CGE) [10] is a GNN that is combined with expert knowledge, and can classify and predict network vulnerabilities. CGE first normalizes the contract graph, highlighting the key nodes. Then, CGE proposes a novel temporal message propagation network, which extracts graph features from the normalized contract graph, and combines the graph features with the designed expert patterns, resulting in the final detection system. It should be emphasized that CGE explicitly takes into account the distinct roles and temporal relationships of program elements.

3 Detection Method Based on GAN and GMN

This section details a novel and effective detection method for reentrancy and integer overflow vulnerabilities in Ethereum smart contracts. The high-level idea is to use graphs to represent smart contract source codes, and innovatively use

GAN to augment the training set, which is used to train GMN to generate a detection model. More specifically, for a given smart contract, whether it is a target or used for model training, we extract the Abstract Syntax Tree (AST) [16], Control Flow Graph (CFG) [16] and Data Flow Graph (DFG) [18] from its Solidity code and integrate them into an undirected multigraph, which can preserve the semantic and structural features of the contract as much as possible. Then, we construct a GraphGAN model consisting of a discriminator and a generator. The generator is responsible for generating neighboring nodes, which are evaluated by the discriminator to determine their authenticity. A small sample training set will be used for adversarial training of the discriminator and generator. After GraphGAN is trained, the large number of synthetic graphs generated by the generator will be used to augment the training set, and the augmented training set will be embedded as graph vectors, and used to train the GMN model. Finally, the trained GMN model uses a cross-graph attention mechanism to calculate the similarity between the target and vulnerability contracts. Our method is the first method that combines GraphGAN and GMN to detect smart contract vulnerabilities, and it is superior to existing methods in terms of accuracy and efficiency.

3.1 Data Preprocessing

A common obstacle for deep learning-based vulnerability detection is that it relies on a single code representation, such as AST, CFG, and text tokens. To address the data representation obstacle, we explore and design a preprocessing method to obtain the graph representation of the Solidity code. Fig. 1 illustrates the overview of data preprocessing. The Solidity code needs to be normalized first. Useless information, such as comments, spaces, and blank lines, must be removed. By removing these redundant fragments, we can improve the efficiency of preprocessing and model training.

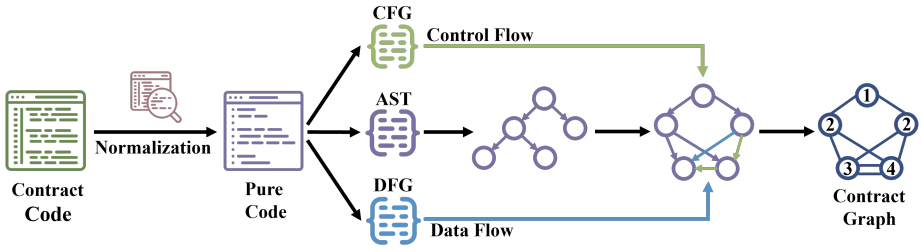

Fig. 1. Overview of data preprocessing.

To capture the syntactic structure of the code, solidity-parser-antlr [19] is used to generate the AST of the code. AST can provide the abstract syntactic structure of smart contracts, where leaf nodes represent operands and non-leaf

nodes represent operators or other syntactic constructs. In order to distinguish different types of nodes, we can mark unique identifiers for each type of node in the AST. Through these identifiers, we can construct context paths that represent semantic information of the code. These paths represent different semantic fragments of code execution.

It should be noted that AST, as a fine-grained syntactic representation, has a considerable computational overhead that cannot be ignored. If the code has a significant size, AST will be extremely complex, leading to problems such as gradient explosion or overfitting in subsequent model training. Therefore, preprocessing needs to introduce coarse-grained syntax to solve this problem. Consequently, we innovatively designed a statement-level granularity representation form - Contract Graph (CG), which consists of AST, CFG, and DFG. The CFG of contracts can be generated using SIF [16], while the DFG is generated using Peculiar [18]. Building upon the foundation of AST, CFG and DFG are introduced to enrich the semantic and structural information of CG, which is defined as

$$CG = (N, E) \tag{1}$$

where N is a set of nodes, and E is a set of edges. Each node in N represents a statement in the contract code. E contains undirected edges between each pair of nodes, and the number of undirected edges represents the execution order, control relationship or data relationship between the corresponding nodes. The construction of CG needs to follow the input format of the GraphGAN [15].

The control relationship can be obtained from CFG [6]. This relationship usually appears in statements with keywords such as if, $while$, for and $break$. If there is a control relationship between two nodes, then the edge between the nodes needs to be assigned directionality according to the keyword attribute. The data relationship can be obtained from DFG, which describes the data relationship between some operations. It should be emphasized that data relationship depends on the execution order of statements. In addition, variables defined in the code must be recorded in a variable set. We first determine the control path in the contract, and then determine the data path by checking the variables on the control path. We assigned corresponding priority numerical labels to the key nodes in the contract according to the top-down order of the hierarchy in the AST. Then, we determined the number of undirected edges between each pair of key nodes according to the execution order, control relationship, and data relationship. Specifically, the path from a high-priority node to a low-priority node is the positive direction, and the number of undirected edges is 1; the path from a low-priority node to a high-priority node is the reverse direction, and the number of undirected edges is 2; if there is no path between two nodes, it is the no direction, and the number of undirected edges is 0. The key nodes and edges together form the CG. Compared with the research methods that only use AST or CFG [6] to represent smart contracts, CG is more conducive to GMN extracting the key features of contracts.

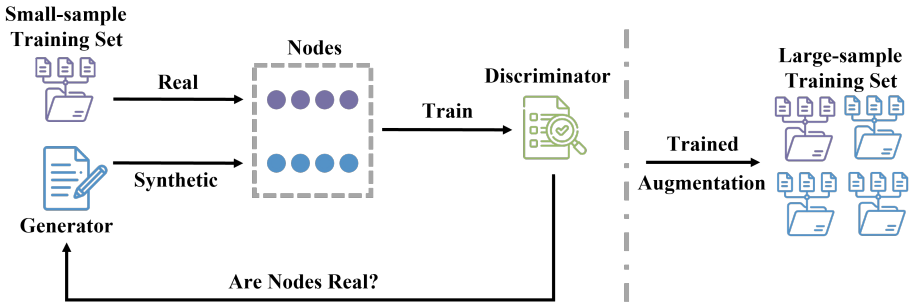

Fig. 2. Process of dataset augmentation.

3.2 Dataset Augmentation

Fig. 2 illustrates the running process of dataset augmentation. This detection method innovatively introduces the GraphGAN model to solve the data starvation problem mentioned in Section 1, which is recognized as a hindrance to building Graph Neural Network models. GraphGAN consists of a generator and a discriminator. In the training process of this model, the generator is used to generate neighbor nodes of CG. In contrast, the discriminator calculates the probability of edge existence between two nodes. More specifically, $CG = (N, E)$ is input to the model, where

$$N = \{n_1, \cdots, n_N\} \tag{2}$$

represents the set of nodes, and

$$E = \{e_{xy}\}_{x,y \in N} \tag{3}$$

represents the set of edges. For a given node n_c, we define $N(n_c)$ as the set of nodes directly connected to n_c, and $N(n_c)$ is always much smaller than N in general. We represent the potential true connectivity distribution of n_c as conditional probability $p_{true}(n|n_c)$, which reflects the connection preference distribution of n_c over all other nodes in N. $N(n_c)$ can be regarded as a set of observation samples drawn from $p_{true}(n|n_c)$.

Based on this, the generator can be defined as $G(n|n_c; \theta_G)$, which tries to approximate the potential true connectivity distribution $p_{true}(n|n_c)$, and generate or select the most likely nodes to connect with n_c. The discriminator can be defined as $D(n|n_c; \theta_D)$, which aims to distinguish the connectivity of node pairs (n, n_c), and output a single scalar, indicating the probability of edge existence between n and n_c, and predicts the number of undirected edges. θ_G and θ_D are the parameters of the GraphGAN, which control the behavior of the generator and the discriminator, respectively. They can automatically learn and adjust through the backpropagation algorithm, enabling GraphGAN to generate high-quality data for various application scenarios. In the training process, the

generator gradually learns how to generate synthetic nodes that match the real neighbor nodes, while the discriminator also gradually learns how to evaluate the edge probability between two nodes correctly [15].

The generator $G(n|n_c;\theta_G)$ and discriminator $D(n|n_c;\theta_D)$ are playing a minimax game with the value function $V(G,D)$:

$$\min_{\theta_G}\max_{\theta_D} V(G,D)=\sum_{c=1}^{N}(\mathbb{E}_{n\sim p_{true}(\cdot|n_c)}\log D+\mathbb{E}_{n\sim G(\cdot|n_c;\theta_G)}[\log(1-D)]). \quad (4)$$

where $\mathbb{E}$ is the expectation, which represents the average output value of the discriminator for all possible n under the real distribution $p_{true}(\cdot|n_c)$ or the generator's distribution $G(\cdot|n_c;\theta_G)$ given n_c. The optimal parameters of the generator and discriminator can be learned by alternately maximizing and minimizing the value function $V(G,D)$. When the generator and discriminator reach Nash equilibrium, we can consider that GraphGAN has completed training. In the case of completing training, the model will be able to generate synthetic CGs that are similar to real CGs. In this way, we can generate a large number of synthetic CGs to enrich the training set of GMN.

It should be emphasized that we augment the datasets of safe contracts, integer overflow contracts and reentrancy contracts separately by independently training three GraphGAN models. Independent training aims to reduce the similarity between these three types of synthetic contracts, thereby reducing the false positive detection rate. Each GraphGAN model is trained with the same parameters, which can achieve the effect of fast transfer learning and improve the training efficiency.

3.3 Similarity Detection

GMN is used to build a similarity detection model. Unlike traditional GNNs, GMN does not map each graph to a feature vector independently, but obtains a pair of graphs simultaneously and uses the cross-graph attention mechanism to learn embeddings [14]. By associating the nodes in the graph pair and identifying the differences, GMN can provide a good accuracy-computation trade-off, which is unavailable in traditional embedding models.

Taking the construction of the integer overflow detection model as an example, we need to label the types of safe contracts and integer overflow contracts output by the Dataset Augmentation stage. All contracts will be embedded as graph vectors by word2vec [20] and form the training set. Moreover, the contracts in the training set will be randomly combined into graph pairs. GMN model will use this dataset for training and detect whether the target contract is an integer overflow contract. Similarly, the reentrancy vulnerability detection model also needs to go through the same process for construction. The construction and similarity detection process of the GMN model is shown in Fig. 3.

GMN model consists of encoding layer, propagation layer and aggregation layer. The encoding layer will use Multi-Layer Perceptron (MLP) [21] to extract

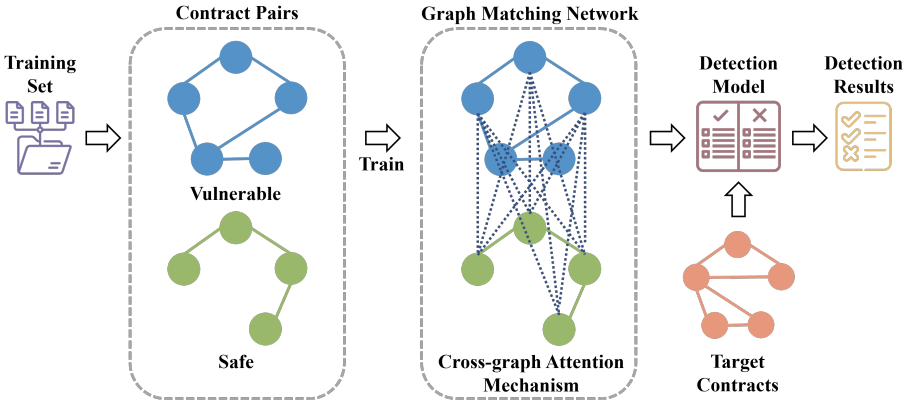

Fig. 3. Construction and workflow of detection model.

and encode the features of nodes and edges in $CG = (N, E)$:

$$\eta_i^{(0)} = MLP_{node}(\alpha_i), \forall i \in N \tag{5}$$

$$e_{ij} = MLP_{edge}(\alpha_{ij}), \forall (i,j) \in E \tag{6}$$

where α_i is the feature vector of node n_i, α_{ij} is the feature vector of edge e_{ij}, $\eta_i^{(0)}$ is the initial embedding of the i-th node, and e_{ij} is the initial embedding of the edge between the i-th and the j-th nodes.

The propagation layer introduces the attention mechanism, which can perform information interaction between two graphs, so as to better capture the structural and semantic similarity between graphs [14]. Compared with ordinary GNNs, GMNs have stronger discriminative and expressive abilities, and can distinguish some graph structures that GNNs cannot distinguish. Given two contract graph $CG_1(N_1, E_1)$ and $CG_2(N_2, E_2)$, at each iteration t, the propagation layer formula is as follows:

$$\mu_{j \to i} = f_m(\eta_i^{(t)}, \eta_j^{(t)}, e_{ij}), \forall (i,j) \in E_1 \cup E_2 \tag{7}$$

$$\nu_{j \to i} = f_c(\eta_i^{(t)}, \eta_j^{(t)}), \forall i \in N_1, j \in N_2 \; or \; i \in N_2, j \in N_1 \tag{8}$$

$$\eta_i^{(t+1)} = f_u(\eta_i^{(t)}, \sum_j \mu_{j \to i}, \sum_{j'} \nu_{j' \to i}). \tag{9}$$

where $\eta_i^{(t)}$ and $\eta_j^{(t)}$ are respective sets of node hidden vectors for CG_1 and CG_2; f_m is the message function that collects messages from the neighborhood of node n_i, f_c is used for cross-graph information computation, and f_u is the node

update function that combines these messages and the final hidden state of node embedding. The attention formula is as follows:

$$a_{j\to i} = exp(s_\eta(\eta_i^{(t)}, \eta_j^{(t)}))/\sum_{j'} exp(s_\eta(\eta_i^{(t)}, \eta_j^{\prime(t)})). \tag{10}$$

and therefore,

$$\sum_j \nu_{j\to i} = \sum_j a_{j\to i}(\eta_i^{(t)} - \eta_j^{(t)}) = \eta_i^{(t)} - \sum_j a_{j\to i}\eta_j^{(t)}. \tag{11}$$

where s_η is a vector space similarity measure, $a_{j\to i}$ are the attention weight, and $\sum_j \nu_{j\to i}$ intuitively measures the difference between $\eta_i^{(t)}$ and its nearest neighbour $\eta_j^{(t)}$ in another graph.

After T rounds of updates in the graph, it is necessary to generate a vector representing the global graph structure information by aggregating the information of all nodes in the graph:

$$\eta_{CG_1} = f_a(\{\eta_i^{(T)}\}_{i\in N_1}). \tag{12}$$

$$\eta_{CG_2} = f_a(\{\eta_i^{(T)}\}_{i\in N_2}). \tag{13}$$

where f_a is the graph aggregation function.

The following formula can calculate the graph pair similarity in the aggregation layer:

$$s = f_s(\eta_{CG_1}, \eta_{CG_2}). \tag{14}$$

where s is the similarity score, f_s is the cosine similarity between η_{CG_1} and η_{CG_2}. The closer s is to -1, the more dissimilar the two graphs are; the closer s is to 1, the more similar the two graphs are ($s \in [-1, 1]$).

After training, the GMN model is used to detect the similarity between the target and vulnerable contracts. We will evaluate and analyze the effectiveness of this detection method through experiments in Section 4.

4 Experimental Result and Analysis

In this section, we will outline the experimental setup, determine the key parameters of GMN through experiments, and compare and analyze it with other methods.

4.1 Experimental Setup

Our experiments are conducted on a Windows 10 (x64) computer equipped with an Intel Core CPU (2.30GHz×8), 16GB (3200MHz) memory, and Nvidia GeForce RTX 2060.

We collect 550 open-source smart contracts compiled with Solidity from the Etherscan platform [22], which are all preprocessed into CGs and labeled with

security types. As shown in Table 1, these smart contracts were divided into two datasets for model building and performance evaluation. We used the GraphGAN model to augment the number of safe, reentrancy, and integer overflow contracts in the original training set to 500, 1000, and 1000, respectively. The augmented training set was used to build the GMN detection model.

Table 1. Summary of datasets.

Type of dataset	**Number of smart contracts**		
	Safe	Reentrancy	Integer overflow
Original training set	50	100	100
Test set	100	100	100

4.2 GMN Parameters Analysis

As we know, GMN, as a Graph Neural Network, its performance is affected by the key parameters [14]. We focus on experimenting and analyzing the Embedding Dimension and the number of Hidden Layers in GMN. According to the sample class distribution of the test set, Accuracy (ACC) is used to evaluate the effectiveness of the detection model:

$$ACC = (TP + TN)/(TP + TN + FP + FN). \tag{15}$$

where TP represents the number of true positives, TN represents the number of true negatives, FP represents the number of false positives, and FN represents the number of false negatives. The closer ACC is to 100%, the higher the detection accuracy of the model.

Control variable experiments are implemented to explore the optimal values of the Embedding Dimension and the number of Hidden Layers. When considering the assignment of dependent variables, the Embedding Dimension is set to 90, and the number of Hidden Layers is set to 5. As shown in Fig. 4, the detection accuracy of the model is highest when the Embedding Dimension is 100. It is obvious that overfitting occurs when the value exceeds 100. As shown in Fig. 5, four hidden layers can capture the connection relationships in CG well, and the model can maximize the extraction of the internal features of the contract. We will use these optimal parameters in the subsequent comparative experiments.

4.3 Comparative Analysis

The comparative experiments will use the test set. We select three representative detection tools, Oyente [8], sFuzz [9] and CGE [10], mentioned in Section 2, for comparative analysis. In addition, a special control group is introduced into the experiment to analyze the role of GraphGAN in building the detection model. The control group (Model Ω) does not use GraphGAN for data augmentation,

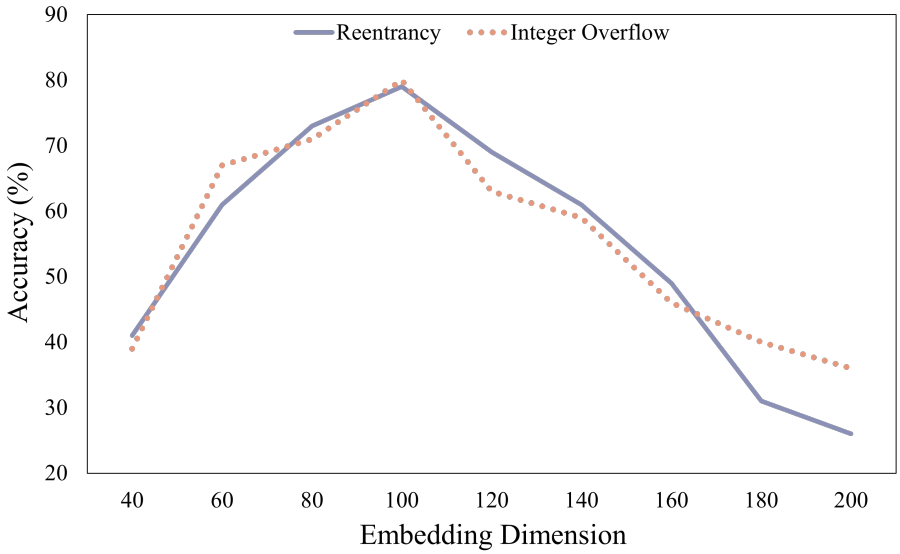

Fig. 4. Experiments on the embedding dimension.

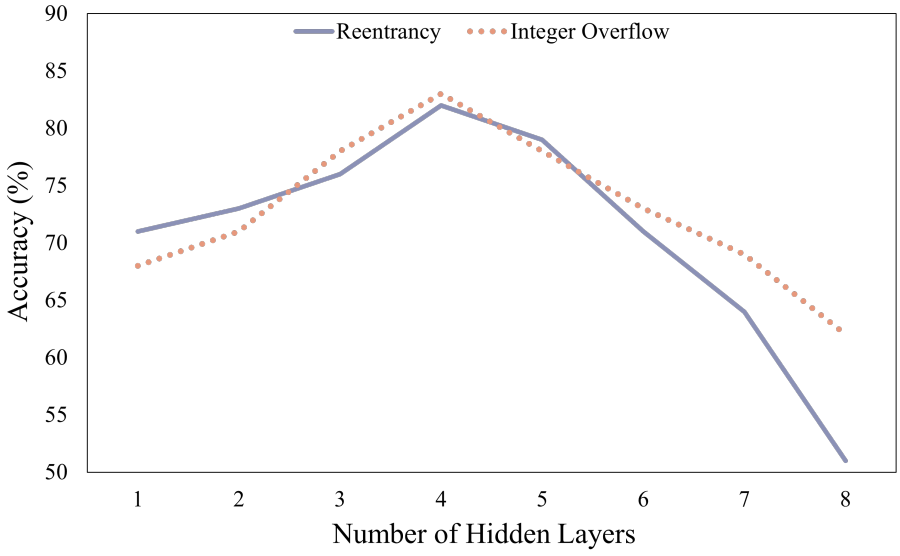

Fig. 5. Experiments on the number of hidden layers.

but directly trains the GMN model with the original training set. As shown in Fig. 6, Model Ω's detection accuracy for reentrancy and integer overflow is much lower than our model. Due to the small sample size of the training set, Model Ω produces overfitting, which prevents the model from more comprehensively identifying the rich vulnerability features in the test set.

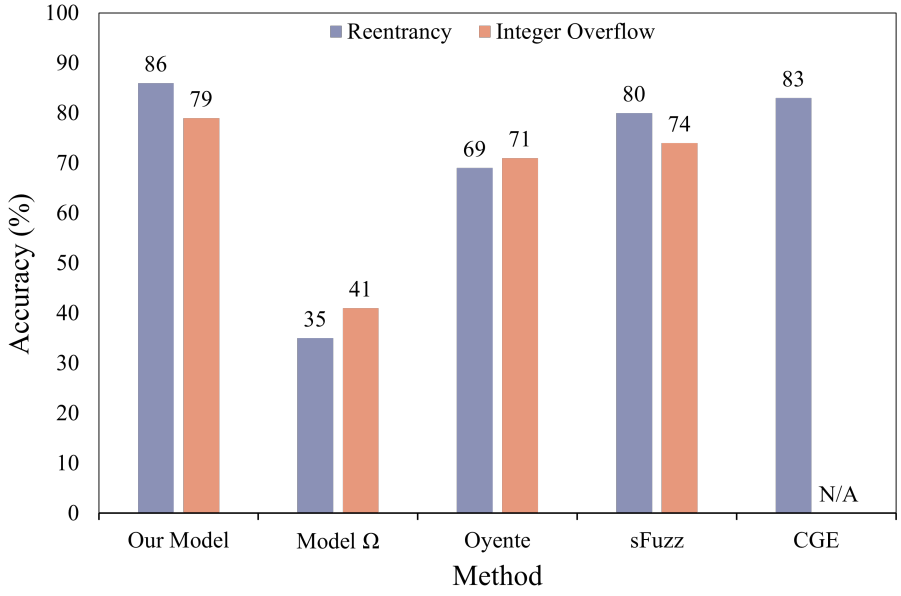

Fig. 6. Methods accuracy experiment.

In the comparative experiments, our model's detection accuracy for reentrancy and integer overflow was 86% and 79%, respectively, higher than the other three tools. Moreover, our model also has some advantages in detection efficiency. As shown in Fig. 7, we conducted experiments by calculating each contract's average detection time to measure the method's efficiency. In terms of detecting integer overflow, our model's average detection time was 5.8s, second only to sFuzz's 4.8s. Our model's average detection time for reentrancy was only 4.1s, shorter than the detection time of the other three tools. In summary, our model can achieve high detection accuracy while ensuring detection efficiency.

5 Conclusion

This paper proposes a smart contract vulnerability detection method based on GAN and GMN for the first time. The method utilizes the structural characteristics of graph data and the cross-graph attention mechanism of the GMN model to

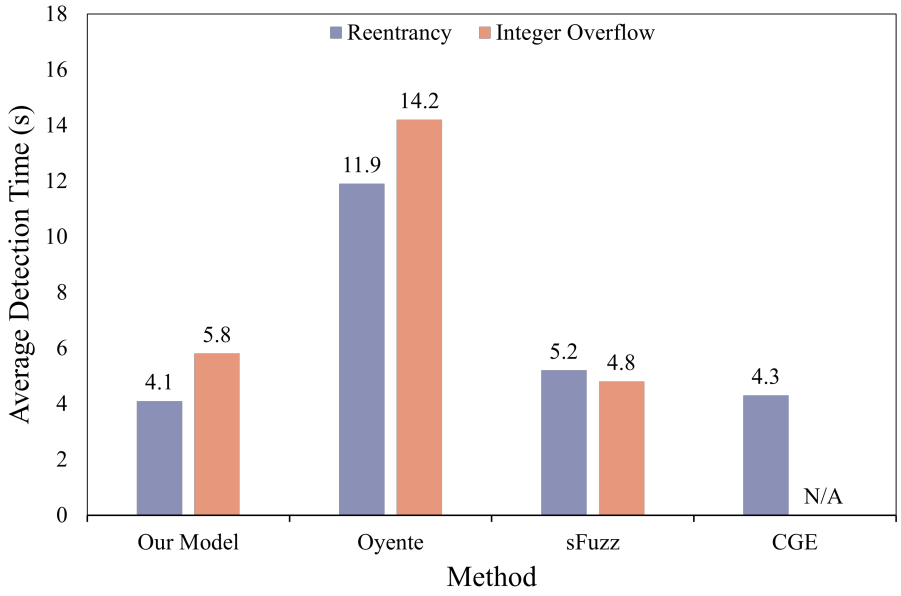

Fig. 7. Methods efficiency experiment.

identify smart contracts with reentrancy and integer overflow vulnerabilities. By using GraphGAN, we can generate a large number of synthetic contract graphs, which can maintain similar structural and semantic features to real contracts. The data starvation problem in deep learning is solved by GraphGAN, which makes it possible for GMN to train with small sample data. The detection model achieves high detection accuracy with the cross-graph attention mechanism of GMN. The experimental results demonstrate the feasibility and effectiveness of the proposed smart contract detection method. This paper not only presents a novel detection method for the field of smart contract security but also offers new insights into the fusion of GAN and GMN. In future work, we can explore and expand the model to detect more types of vulnerabilities.

References

1. Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37):2–1, 2014.
2. Xu Wang, Xuan Zha, Wei Ni, Ren Ping Liu, Y Jay Guo, Xinxin Niu, and Kangfeng Zheng. Survey on blockchain for internet of things. *Computer Communications*, 136:10–29, 2019.
3. Xu Wang, Xuan Zha, Guangsheng Yu, Wei Ni, Ren Ping Liu, Y Jay Guo, Xinxin Niu, and Kangfeng Zheng. Attack and defence of ethereum remote apis. In *2018 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6. IEEE, 2018.

4. Daojing He, Zhi Deng, Yuxing Zhang, Sammy Chan, Yao Cheng, and Nadra Guizani. Smart contract vulnerability analysis and security audit. *IEEE Network*, 34(5):276–282, 2020.
5. Cade Metz. The biggest crowdfunding project ever—the dao—is kind of a mess, 2016.
6. Jianjun Huang, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu, and Yanjun Wu. Hunting vulnerable smart contracts via graph embedding based bytecode matching. *IEEE Transactions on Information Forensics and Security*, 16:2144–2156, 2021.
7. Daojing He, Rui Wu, Xinji Li, Sammy Chan, and Mohsen Guizani. Detection of vulnerabilities of blockchain smart contracts. *IEEE Internet of Things Journal*, 2023.
8. Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.
9. Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.
10. Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering*, 2021.
11. Michael Fröwis and Rainer Böhme. In code we trust? measuring the control flow immutability of all smart contracts deployed on ethereum. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2017 International Workshops, DPM 2017 and CBT 2017, Oslo, Norway, September 14-15, 2017, Proceedings*, pages 357–372. Springer, 2017.
12. Zixu Zhang, Xu Wang, Guangsheng Yu, Wei Ni, Ren Ping Liu, Nektarios Georgalas, and Andrew Reeves. A community detection-based blockchain sharding scheme. In *Blockchain–ICBC 2022: 5th International Conference, Held as part of the Services Conference Federation, SCF 2022, Honolulu, HI, USA, December 10–14, 2022, Proceedings*, pages 78–91. Springer, 2022.
13. Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. Generative adversarial networks: An overview. *IEEE signal processing magazine*, 35(1):53–65, 2018.
14. Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *International conference on machine learning*, pages 3835–3845. PMLR, 2019.
15. Hongwei Wang, Jialin Wang, Jia Wang, Miao Zhao, Weinan Zhang, Fuzheng Zhang, Wenjie Li, Xing Xie, and Minyi Guo. Learning graph representation with generative adversarial nets. *IEEE Transactions on Knowledge and Data Engineering*, 33(8):3090–3103, 2019.
16. Chao Peng, Sefa Akca, and Ajitha Rajan. Sif: A framework for solidity contract instrumentation and analysis. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 466–473. IEEE, 2019.
17. Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

18. Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 378–389. IEEE, 2021.
19. Federico Bond. solidity-parser-antlr: A solidity parser built on top of a robust antlr4 grammar. *https://github.com/federicobond/solidity-parser-antlr*, 2019.
20. Kenneth Ward Church. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017.
21. Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.
22. Etherscan. Etherscan—the ethereum blockchain explorer. *https://etherscan. io/*, 2021.