ECAI 2023
K. Gal et al. (Eds.)

doi:10.3233/FAIA230374

GraphSA: Smart Contract Vulnerability Detection Combining Graph Neural Networks and Static Analysis

Long He[a], Xiangfu Zhao[a;*], Yichen Wang[a], Jiahui Yang[a] and Xuelei Sun[a]

[a]School of Computer and Control Engineering, Yantai University, Yantai 264005, China
ORCiD ID: Xiangfu Zhao https://orcid.org/0000-0001-5870-5730

Abstract. Security incidents in smart contracts still occur frequently, as the underlying code is often vulnerable to attacks. However, traditional methods to detect vulnerabilities in smart contracts are limited by certain rigid rules, reducing accuracy and scalability. In this work, we propose *GraphSA*, which combines ***Graph*** neural networks (GNNs) and ***S****tatic* ***A****nalysis* for smart contract vulnerability detection. First, we present the *contract tree*, which is obtained by converting the control flow graph (CFG) of a smart contract. Each node in the tree represents a crucial operation code (opcode) block, and each edge represents the control flow (execution order) between code blocks. Then, we propose an extended SAGConv and Topkpooling graph neural network (ST-GNN) to learn the features of each node in the tree. To enhance detection accuracy, we eliminate and merge some non-crucial nodes to highlight key nodes and execution orders. Finally, we evaluate our approach on 7,962 real-world smart contracts running on Ethereum and compare it with state-of-the-art approaches on *six* types of vulnerabilities. Experimental results show that our approach achieves higher detection accuracy than others.

1 Introduction

Blockchain is a distributed ledger technology that is revolutionizing the way that we store and exchange information. It is a digital ledger of transactions maintained by a network of computers rather than a central authority. Each block in the chain contains a cryptographic hash of the previous block, a timestamp, and transaction data, ensuring that the data is secure and tamper-proof. Blockchain technology was first introduced in 2008 with the creation of Bitcoin [21], a digital currency that operates on a blockchain network. Since then, blockchain technology has evolved, and it is now being widely applied, including in finance [32], healthcare [24], supply chain management [16], and knowledge graph domain [3], etc.

Smart contract and security incidents. Smart contracts, self-executing computer programs, are created using advanced programs like Solidity [9] running on Ethereum [23]. They contain a set of rules and conditions that are automatically executed without the need for intermediaries or notaries. Smart contracts have been widely adopted on various blockchain platforms, where millions of contracts are currently deployed. This has enabled a diverse range of applications such as wallets [1], crowdfunding, decentralized gambling [6], and cross-industry finance [15]. According to the latest report from Alchemy [27], the number of smart contracts deployed on Ethereum in 2022 increased by 293% compared to those in 2021.

Applications based on smart contracts are becoming more popular in many fields, with these contracts holding over $10 billion worth of virtual coins. The high value of smart contracts has resulted in numerous security incidents. The reentrancy vulnerability of the DAO contract [33] was exploited by attackers in June 2016, resulting in the theft of 3.6 million ethers, equivalent to about $60 million. Besides, an integer overflow bug in the BEC campaign [2] resulted in the sudden disappearance of more than $900 million. Instances of security vulnerabilities in smart contracts are not isolated, and new exploits are discovered and utilized every few months. According to statistics from SlowMist Hacked [25], blockchain networks have suffered losses of more than $29 billion due to security issues in smart contracts, with losses reaching $4.3 billion in 2022. So far, there have been a total of 968 recorded security incidents on Ethereum, with 308 of them occurred in 2022, accounting for around one-third of all security incidents.

Causes of smart contract prone to errors. There are several factors that make smart contracts particularly error-prone. *First,* smart contracts are designed to automate complex business processes, and as a result, their code can be highly intricate and challenging to write and audit. *Second,* the rapid evolution of smart contract technology means that developers are often working with new and untested frameworks and tools, which can introduce new risks and vulnerabilities. *Last,* traditional programs can be modified when encountering bugs during runtime. However, due to the tamper-proofing of smart contracts, they cannot be modified once deployed. Therefore, it is necessary to use an effective and efficient tool for vulnerability detection before deploying smart contracts.

Limitations of traditional methods. Traditional methods for smart contract vulnerability detection mainly include symbolic execution [20], formal verification [4], intermediate representation [10], and fuzzy testing [31]. *Symbolic execution* is a method for analyzing software by treating program variables as symbols and systematically exploring all possible execution paths. It is a powerful technique for analyzing software behavior, but its effectiveness depends on the size and complexity of the program being analyzed. *Formal verification* is used to check the correctness of smart contracts by mathematically proving that they behave as intended. It can help effectively identify potential vulnerabilities, but the techniques require significant expertise to apply effectively. *Intermediate representation* is used to transform smart contract source code into a more amenable form for analysis. However, the method may lead to a loss of precision when analyzing the source code. *Fuzzy testing* involves randomly generating input data to simulate various scenarios used for testing the

* Corresponding Author. Email: xiangfuzhao@gmail.com.

security of smart contracts. However, fuzzing testing results depend on the quality of the input generator.

Recently, however, efforts have been made to adopt deep neural networks for the detection of smart contract vulnerabilities, resulting in increased accuracy. [26] utilizes the Long Short Term Memory (LSTM) networks to sequentially process source code. [34] models graphs to represent both syntactic and semantic structures of smart contract functions. [18] uses graph neural networks and expert knowledge for smart contract vulnerability detection. However, the above-mentioned approaches either only consider the textual features of operation code or construct semantic and control information flows at the source code level. These limitations result in a loss of precision in the detection of smart contract vulnerabilities while making the models complex to use. In addition, based on statistics, among the top 1.5 million smart contracts deployed, only 32,499 (about 2%) have their source code on Etherscan [12], where their bytecode is public to ensure transparency.

In this work, we used traditional tools for generating the control flow graph (CFG) of a smart contract, such as *Oyente* [20], *Mythril* [8], and *Vandal* [5], to analyze over 1,500 real-world operation codes of smart contracts, and presented a fully automated and scalable method for detecting smart contract vulnerabilities with higher accuracy at the operation code level. In a CFG, nodes represent the complete semantic information of basic blocks, and edges represent the execution order. However, not all basic blocks and edges in the CFG are equally important, and most of the GNN is essentially flat during information propagation. Therefore, we designed a non-critical basic block simplification and aggregation phase to normalize the CFG and highlight key nodes and the order of execution. Then, we transformed the basic blocks, control flow dependencies, and data flow information in the CFG into a novel *contract tree*. Next, we extracted the features of the contract tree and utilized them as input for the ST-GNN network, which comprises the extended SAGConv [11] and Topkpooling [7] components. The ST-GNN network learned the features of the contract tree and employed them to detect vulnerabilities in the smart contracts. Finally, experiments on 7,962 real-world smart contracts from Ethereum concluded that the accuracy of GraphSA, which we proposed, has significantly improved in detecting *reentrancy*, *self-destruct*, *delegate-call*, *transaction-order dependency*, *timestamp dependency*, and *integer overflow* vulnerabilities.

Contributions. The key contributions are listed as follows:

- We proposed a novel concept *contract tree*, converted by CFG of a smart contract, and combined it with GNNs for vulnerability analysis.
- We proposed *two* algorithms based on CFG to better capture information from *contract trees*, highlighting key nodes and execution sequences, and maximizing the reconstruction of the static program analysis process.
- We proposed the *ST-GNN* network model for training. Experimental results showed that our method has significant advantages in detecting *six* specific types of vulnerabilities in smart contracts.

2 Background

In this section, we briefly review relevant knowledge.

2.1 Control Flow Graph

The CFG is a graphical representation for describing the control flow of code in smart contracts. It can help us better understand the behavior of smart contracts and identify vulnerabilities. Each smart contract can be represented as a CFG, composed of nodes and directed edges. Nodes represent basic blocks, which are a group of opcode instructions executed in sequence. Edges represent the flow of control, the order of execution between basic blocks. There are two execution sequences: the first is sequential execution between adjacent basic blocks, dependent only on the type of opcode. The second is jump execution between non-adjacent basic blocks, dependent on the analysis of the relationship between instruction opcodes and operands in the execution process to determine the position of specific jump blocks.

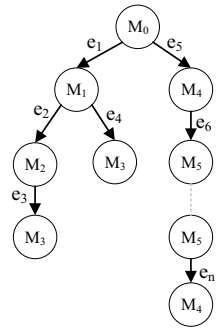

Figure 1. A simple contract tree example.

2.2 Graph Neural Networks

GNNs have gained widespread attention in recent years as a type of machine learning model used for processing graph data. Unlike traditional neural network models that take vectors or matrices as inputs, GNNs take graphs in their representation as inputs. This feature allows GNNs to perform forward and backward propagation on graphs of any size and shape, making them suitable for processing no-fixed-structure data. The core idea behind GNNs is to model the interaction between nodes as an information passing process to obtain global graph structural features. At each node, GNNs combine the features of the node and its neighboring nodes into a new feature vector, which is then passed on to the next node. This process can be repeated multiple times until a certain termination condition is met.

3 GraphSA

In this section, we firstly present a new concept *contract tree*, and based on which, we then introduce our proposed method GraphSA in detail.

3.1 Contract tree

Definition 1. *Given a smart contract c and its CFG, a* Contract Tree *of c is a tree, defined as follows:*

- *Each node is a code block and carries a unique identifier, and each edge is the control flow passing from one code block to another.*
- *The root node is the starting point of the program execution flow.*
- *Each leaf node is the termination of the program execution flow.*
- *A sequence of adjacent edges between nodes is referred to as a path, with each path representing a scenario during program execution.*

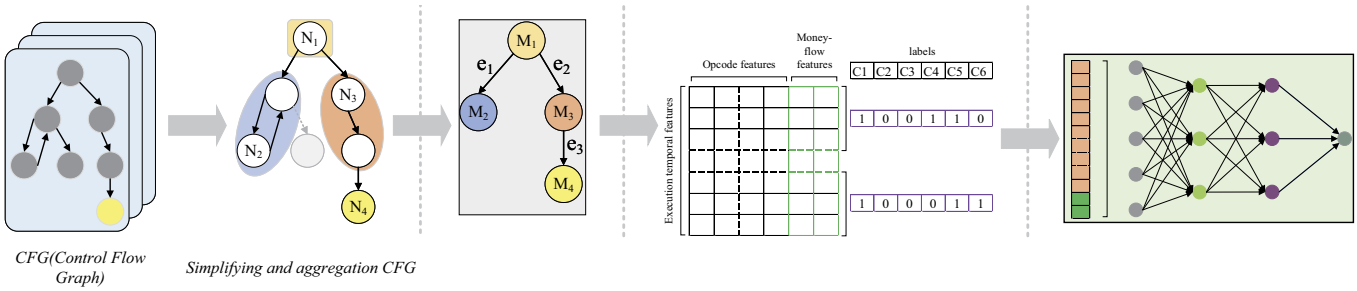

Figure 2. The overall architecture of GraphSA. (a) The simplification of the CFG phase; (b) the cotract tree building phase; (c) the feature extraction phase; (d) the vulnerability detection phase.

An example of a contract tree. An example of a simple contract tree is shown in Fig. 1. e_i indicates the number of the node control flow. In each path, the order from smallest to largest of i represents the actual operation order of the control flow. M_i represents the node number, and nodes with the same subscript i are copies of the same code block. In Fig. 1, there are two M_3 nodes in the left branch of M_0, indicating that there are two different paths to reach M_3. In the right branch of M_0 in Fig. 1, there are multiple control flows from M_4 to M_5 that correspond to the cyclic structure in the CFG. In Fig. 1, e_1 and e_5 are conditional branches of program execution that will not be carried out concurrently in the process of program execution. Therefore, the information passing process from M_0 to M_1 and M_0 to M_4 in the contract tree can be carried out simultaneously.

The feature of contract tree. Based on Definition 1 and the example in Fig. 1, we can summarize the characteristics of a contract tree as follows:

- **Integrality** - The contract tree contains all paths of program execution, and each path just represents a situation.
- **Temporality** - Each path in the contract tree is constructed in the order of each code block in one execution.
- **Efficiency** - Nodes in each path that are unrelated to vulnerabilities are simplified and aggregated, while identical and unrelated paths are merged to improve message passing efficiency.

In this work, we will use contract trees as inputs to ST-GNN training and detection, in subsequent sections.

3.2 GraphSA

Method overview. The main idea of our proposed method for smart contract vulnerability detection is to combine ***Graph*** neural networks and ***S***tatic ***A***nalysis (GraphSA). The overall architecture of GraphSA is shown in Fig. 2, including four phases:

(1) Simplification of CFG: We first simplify the CFG paths irrelevant to vulnerability detection, and then perform information aggregation on the simplified CFG.
(2) Contract tree building: Next, we build the contract tree from the simplified and aggregated CFG.
(3) Feature extraction: Feature extraction includes extracting opcode semantics and data flow features from each node.
(4) Vulnerability detection: We use *ST-GNN* to train the features of the *contract tree* and output the detection results.

In the following sub-sections, we will provide detailed explanations of these four steps one by one.

3.2.1 Simplification of CFG

Observations. For analyzing the smart contracts in detail, we used existing software debugging tools to generate CFGs over 1,500 real-world smart contracts. We observed the presence of quantities of non-relevant paths for smart contract vulnerability detection in CFGs. Therefore, simplifying these paths will improve the accuracy and efficiency of feature extraction for vulnerability detection. In addition, to highlight the key nodes and control flows of CFGs, we further aggregated information on simplified CFGs.

Nodes construction and definition. *Major nodes* represent the basic block that is closely related to the vulnerability. It includes the basic block that directly or indirectly contains opcodes related to the vulnerability. For example, ADD, $TIMESTAMP$, etc., can lead to integer overflow and timestamp dependency vulnerabilities. Formally, we describe all key opcode blocks as major nodes, which are denoted by M_1, M_2, ..., M_n. *Secondary nodes* are used to describe basic code blocks that do not contain opcodes related to vulnerabilities. Formally, secondary nodes are defined as S_1, S_2, ..., S_n. *Leaf nodes* represent nodes with an out-degree of 0. Formally, leaf nodes are defined as L_1, L_2, ..., L_n. *Edges* include two types: the basic control flow edge and the special control flow edge. The basic control flow is a common control flow that exists in CFG. However, some special structures in CFG are closely related to vulnerabilities, such as cyclic structures, etc. We define the control flows related to these structures as special control flows.

For convenience, in subsequent descriptions, some definitions in CFG are given as follows: A *branch node* is a node with out-degree greater than 1. A *root node* is a node with in-degree of 0.

Simplifying CFGs. As shown in the left and middle sub-figures of Fig. 3, we applied Algorithm 1 to simplify the CFG. In this algorithm, we adopted a bottom-up recursive tactic to process the CFG paths, ensuring the integrity of the major nodes and special structures. The recursive termination condition of the algorithm is that all leaf nodes are major nodes, that is, the subset of major nodes consists of leaf nodes. The first line of the algorithm is the recursive termination condition, while lines 4-10 are used to check whether the current leaf node is a major node and, if not, delete the node and its associated edges. Line 12 regains the new leaf nodes in the current CFG after one round of simplification. Line 13 serves as the entry point for the recursion process.

Aggregating information. As shown in the middle subgraph of Fig. 3, the simplified CFG preserves the major nodes and their control flows. However, there are still numerous secondary nodes in the simplified CFG. Moreover, most GNNs are inherently flat when

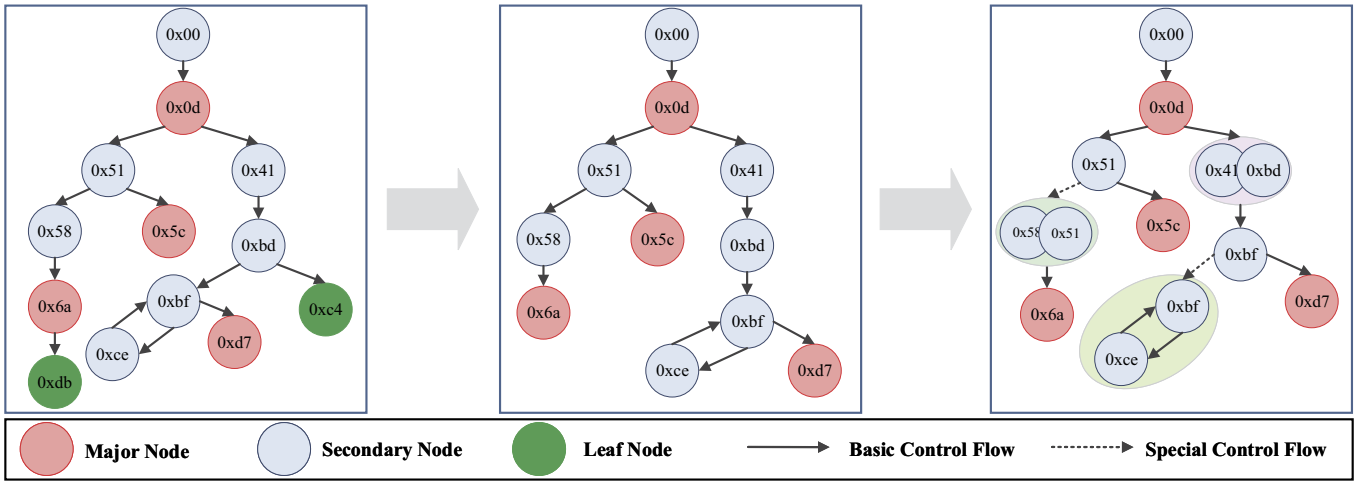

Figure 3. The detailed process of CFG preprocessing includes the complete CFG (left), the simplified CFG (middle), and the aggregated subgraph (right).

Algorithm 1 *SCFG*($\mathcal{E}$,$\mathcal{L}$): Simplifying CFG.

Input: $\mathcal{E}$: the set of edges that make up the CFG; $\mathcal{L}$: the set of leaf nodes in the CFG.
Output: *SE*($\mathcal{E}$,$\mathcal{L}$) - the set of edges that make up the simplified CFG.

if $\mathcal{L}$ consists of only $node(major)$ **then**
  return $\mathcal{E}$;
else
  for each $node_2 \in \mathcal{L}$ **do**
    if $node_2 \neq node(major)$ **then**
      for each $edge < node_1, node_2 > \in \mathcal{E}$ **do**
        $\mathcal{E} \leftarrow \mathcal{E} \setminus \{edge(node_1, node_2)\}$;
      end for
    end if
  end for
end if
$\mathcal{L}$ = *get_leaf_nodes*($\mathcal{E}$);
$SCFG(\mathcal{E}, \mathcal{L})$;

propagating information. To highlight major nodes and reserve each important control flow of program execution paths, we use information aggregation as follows: As illustrated in the right of Fig. 3, information aggregation operations are applied to S_i of three structures. *First*, the nodes of consecutively connected S_i, e.g., $0x41$ and $0xbd$, are aggregated directly. *Second*, the S_i of related to branch nodes, e.g., $0x51$ and $0x58$, reserve branch nodes and aggregate sequential S_i nodes. *Last*, S_i is related to a special structure, such as $0xbf$ and $0xce$, where $0xbf$ is a branch node and a node from a loop structure, so S_i is aggregated while retaining branch nodes.

3.2.2 Contract Tree Building

In this section, we generate contract trees using simplified and aggregated CFGs by Algorithm 1. As shown in Fig. 4, there are two key steps to building a contract tree. First, we obtain the execution order of all paths in CFG (Fig. 4-b), where each path represents a situation in program execution. Then, we merge the same paths that are unrelated during program execution to form a *contract tree* (Fig. 4-c).

The detailed process of building a contract tree is shown in Algorithm 2. In line 1, we obtain all possible paths from the root node to each leaf node. In line 2, we calculate the number of incoming edges for each node on all edges. In line 3, we visit the nodes in the first path and count the number of times each node is visited. Next, lines

Algorithm 2 *BCT*($\mathcal{E}$,$\mathcal{L}$): Building contract tree.

Input: $\mathcal{E}$: The edges set of simplified and aggregated CFG; $\mathcal{L}$: The leaf nodes set of simplified and aggregated CFG.
Output: $CT(E)$ - The edges composing the *contract tree*.

1: $P \leftarrow$ all paths from root to each leaf node in $\mathcal{L}$.
2: $V \leftarrow$ number of incoming edges to each node in $\mathcal{E}$.
3: Visit first path, counting visited count of each node.
4: **for** p in P starting from second path **do**
5: **for** n in p **do**
6: **if** n meets conditions for visitation **then**
7: $V[n] \leftarrow V[n] - 1$;
8: **else**
9: Remove n from p.
10: **end if**
11: **end for**
12: **end for**
13: $E \leftarrow$ Merge remaining paths in P.
14: **return** E;

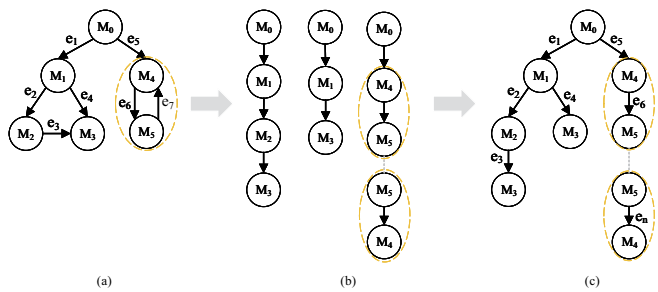

Figure 4. The process of building a contract tree: (a) simplified and aggregated CFG; (b) all paths of the contract tree; (c) *contract tree*.

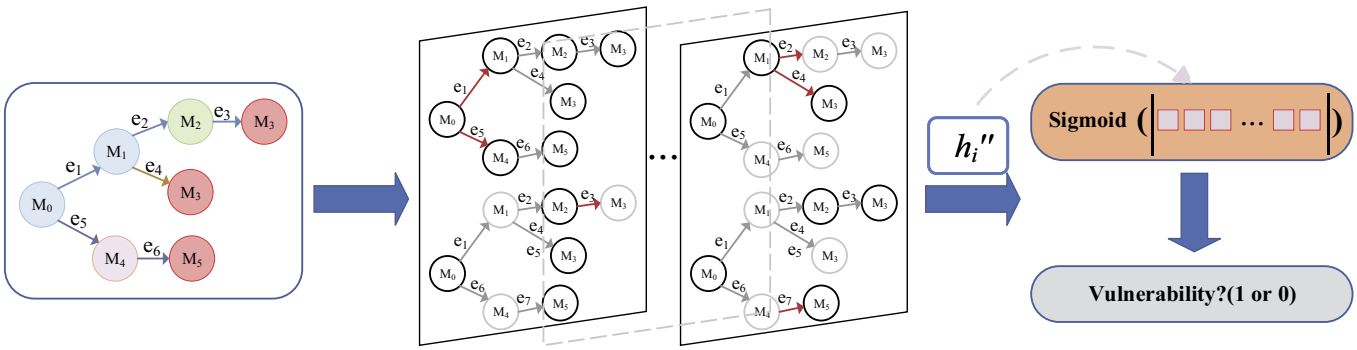

Figure 5. Overall structure of the vulnerability detection model. (a) Input in the form of a contract tree; (b) contract tree feature convolution and pooling; (c) output of vulnerability detection results.

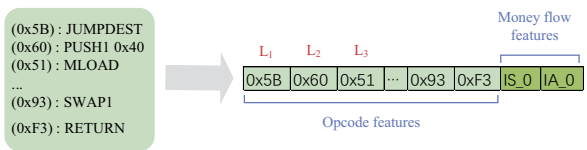

Figure 6. The process of extracting semantic and data flow information features from nodes.

4-12 traverse each path starting from the second path. For each node in the path, if it meets the conditions for visitation (line 6), its visit count is decremented by 1 (line 7). If the node does not meet the conditions, it is removed from the path (line 8). The conditions for a node to be visited are as follows: *First*, the node has not been visited before. *Second*, the node has an out-degree greater than the number of times it has been visited. *Last*, if the node is a leaf node and has an in-degree greater than the number of times it has been visited, it can also be visited. Finally, line 13 merges the remaining paths to form the *contract tree*.

3.2.3 Feature Extraction

In order to capture the semantic and data flow information of operation codes in each node, which is crucial to preserving the essential information of the contract tree, we employ feature extraction. In this section, we will provide a detailed description of the process of semantic and data flow information extraction.

As shown in Fig. 6, we first extract the semantic information from the opcode. We use the decimal value of the bytecode corresponding to each opcode as the opcode feature, and map it to an index position in the feature vector based on the execution order of the opcode in the node. Each value in the feature vector represents an opcode instruction contained in the current node, and the index position of the opcode feature vector corresponds to the execution order of the opcode. However, since the number of operation codes in each node may differ, we set the maximum dimension of the feature vector and use $0x00$ to represent unfilled operation code feature positions. Furthermore, we consider special structures when building features between nodes. Through our investigation of traditional static analysis tools, such as Oyente [20], we found that the maximum depth of path exploration is 50. Therefore, we set the maximum number of loops for special loop structures to 50 during the analysis process.

To further enrich the node feature, we parse each instruction in the order of execution and retain the relevant money flow information that is most significant to the vulnerability. The money flow represents the balance of the sender and receiver, for instance, Oyente [20] uses the money flow to describe the money state in each execution path. As illustrated in Fig. 6, IS_t and IA_t are used to simulate the balance of the sender and receiver after the execution of the t-th code block. Unlike traditional static analysis methods, we fix the amount of each transfer and set the initial values of IS_t and IA_t based on the execution frequency of the corresponding special structure nodes in the contract tree, such as the loop structure related to reentrancy vulnerability.

Once the feature extraction process is completed, we obtain semantic and money flow features for the nodes. We utilize these features as node features to generate datasets for the contract tree and feed the constructed datasets into a deep learning model for training.

3.2.4 Vulnerability Detection

Shown in Fig. 5 are the overall steps for vulnerability detection by GraphSA. Here, we proposed the *ST-GNN* network model based on the characteristics of static analysis of smart contracts. The model combines and extends the *SAGConv* and *TopKpooling*. Combining SAGConv with Topkpooling has numerous benefits for GNNs. This combination improves large-scale graph processing, local information learning, and computational efficiency while preserving the clarity and interpretability of the model. Our network model takes the contract tree T of the smart contract as input and outputs the label $\widehat{y} \in \{0, 1\}$, indicating whether the contract has a certain type of vulnerability.

Contract tree feature convolution and pooling. In the *ST-GNN* network model, the feature of each node is updated through convolution layers. As shown in Equation 1, the new feature $x_i{}'$ of node x_i is obtained by the feature of node x_i itself and the mean of the features of its neighbor nodes, where W_1 and W_2 represent two trainable weight matrixes.

$$x_i{}' = W_1 x_i + W_2 \cdot mean_{j \in N(i)} x_j \tag{1}$$

Furthermore, Equations 2-5 illustrate the scoring mechanism, which calculates the score of each node and selects the nodes with high scores to reconstruct their adjacency matrix. During this process, the convolved node representation is aggregated, and the most

important nodes are selected for the next layer of operation. Through multiple rounds of convolution and pooling, the model can gradually extract node features.

$$s_i = W_3^T max_{j \in N(i)}(x_j^{'}) \tag{2}$$

$$a_i^{'} = \frac{exp(s_i)}{\sum\limits_{j \in N(i)} exp(s_j)} \tag{3}$$

$$h_i^{'} = \sum_{j \in N(i)} a_i^{'} x_j^{'} \tag{4}$$

$$A^{'} = \sigma(A^T D^{-1} A D) \tag{5}$$

where s_i is the score of node i, W_3 is a trainable weight matrix, $a_i^{'}$ is the weight for node i, $h_i^{'}$ is the aggregated feature vector of node i, A is the adjacency matrix of the original graph, σ is the sigmoid function, and D is the diagonal matrix of node degrees.

Readout phase. After the multiple rounds of convolution and pooling in *T*, *ST-GNN* get a label for *T* by reading out the final features of all nodes. Let $h_i^{''}$ be the final pooled node feature of all nodes, we may generate the prediction label $\widehat{y}$ by

$$\widehat{y} = \sigma(\sum_{i=1}^{|V|} W h_i^{''} + b) \tag{6}$$

where W and b are the weight matrix and bias vector of the fully connected layer, $|V|$ denotes the number of nodes obtained after multiple rounds of convolution and pooling, and σ is the sigmoid function.

The *ST-GNN* is trained to detect vulnerabilities in contracts. The network is trained using numerous *contract trees*, along with their corresponding ground truth labels. Then, the trained model is employed to absorb a contract tree and yields a vulnerability detection label. It is worth noting that we have created automated tools capable of converting source code into contract trees, hence the entire process is fully automated.

4 Experiments

4.1 Datasets and Experimental Settings

Datasets. In this experiment, we chose a dataset consisting of 7,962 real-world smart contracts in versions 0.4-0.8 of Ethereum. To ensure the reliability of the dataset, we made appropriate adjustments to the distribution of six vulnerabilities in it. Choosing this particular dataset was based on several factors. *First*, smart contract bytecode is often unreadable, and different testing tools have varying levels of support for different versions of smart contracts. *Second*, it is difficult to classify and manually review smart contracts, adding to the complexity of the selection process. *Finally*, the uneven distribution of vulnerabilities in the dataset requires appropriate adjustments to ensure accurate results. During the analysis and review of the smart contracts, we found that over 80% of the contracts had complex calling relationships, and almost all had special contract structures. Therefore, the aggregation and simplification mentioned in Section 3 are worthy of this dataset.

Experimental settings. Most smart contract vulnerability detection methods using neural networks are non-open source, while running projects that provide source code locally is challenging. Therefore, we compared our method with three existing smart contract detection tools and two neural network-based methods. In this way, we indirectly compare our approach with others not involved in this experiment. For the dataset, we randomly divided it into 60% of them as the training set, 20% of them as the validation set, and the other 20% as the testing set several times and reported the averaged result. We evaluated the results using *accuracy*, *precision*, *recall*, and *F1-score* metrics. As *Oyente* is unable to detect smart contracts with versions above 0.4.26, we included all smart contracts with versions below 0.4.26 in the dataset, and adjusted the vulnerability distribution accordingly.

4.2 Comparison with Existing Methods

In this section, we first compare GraphSA with three state-of-the-art methods: *Mythril* [8], *Oyente* [20], and *Smartcheck* [28]. Then, we further compare GraphSA with two methods based on alternative neural networks: Vanilla Graph Neural Networks (*Vanillia-GNN*) [19] and Graph Convolutional Networks (*GCN*) [14].

4.2.1 Comparison with State-of-the-Art Existing Methods

We compared GraphSA with existing no-deep-learning methods, namely *Smartcheck* [28], *Oyente* [20], and *Mythril* [8] on detecting six types of vulnerabilities. The performance of them is presented in the top of Tables 1 and 2, where metrics *accuracy*, *recall*, *precision*, and *F1-score* are engaged.

Based on the statistical analysis presented in Tables 1 and 2, we can conclude as follows. *First*, the state-of-the-art tools exhibit unsatisfactory accuracy in detecting these six types of vulnerabilities, with the highest accuracy rate among them being only 71.69%. *Second*, GraphSA achieved the highest accuracy rate of **85.52%**, which is **13.83%** higher than the others. *Moreover*, on average, our method achieved a detection accuracy rate of **over 80%**. However, there exists a significant difference in the detection accuracy rate of these six vulnerabilities among the existing tools, with the lowest accuracy rate being only 40.39%. To further compare our method with theirs, we visualized the experimental data, as shown in Fig. 7.

We conducted further research on the existing tools for smart contract vulnerability detection to explore the reasons behind these observations. *Smartcheck* primarily relies on strict and simple logic rules to detect vulnerabilities, resulting in lower accuracy and F1-scores. *Oyente* uses data flow analysis to improve accuracy, but the underlying patterns for detecting vulnerabilities are not very accurate. In contrast to other methods, *Mythril* integrates complex techniques such as symbolic execution, taint analysis, and manual auditing to detect vulnerabilities. As a result, Mythril performs better in terms of vulnerability detection accuracy.

4.2.2 Comparison with Neural Network-Based Methods

In the following, we further compared GraphSA with other approaches based on neural network models, namely *Vanilla-RNN* and *GCN*. Experimental results are presented in the lower half of Tables 1 and 2. Based on the statistical results, we can conclude that the accuracy of alternative neural network models is better than that of state-of-the-art tools not based on neural network models but still falls short of our method's accuracy. The highest accuracy achieved was 77.85%, which still showed a certain gap compared to our method. To provide a more intuitive and clear comparison between GraphSA and others, we visualized the experimental data in Fig. 7. The visualization reveals that our method outperforms both *Vanilla-RNN* and *GCN* in terms of accuracy, highlighting the effectiveness of our approach.

Table 1. Performance comparison of five methods for *Reentrancy*, *Self-Destruct*, and *Delegate-call* vulnerabilities using *accuracy*, *recall*, *precision*, and *F1-score* metrics, including state-of-the-art detection methods, neural network-based alternatives, and our method *ST-GNN*.'-' denotes not applicable.

Method	Reentrancy				Self-destruct				Delegate-call			
	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)
Smartcheck	52.97	30.10	25.00	28.10	40.39	32.16	36.11	40.59	-	-	-	-
Oyente	61.26	54.71	38.16	44.96	-	-	-	-	-	-	-	-
Mythril	60.54	71.69	39.58	51.02	71.69	55.61	50.84	51.23	59.25	46.15	60.28	50.48
Vanilla-RNN	49.64	58.78	49.82	50.71	53.06	51.89	43.25	41.02	45.18	43.49	43.10	42.69
GCN	77.85	78.79	70.02	74.15	74.20	74.56	70.86	67.32	63.52	63.04	59.26	60.49
GraphSA	**85.52**	**81.25**	**77.33**	**76.56**	**84.81**	**83.75**	**78.82**	**78.95**	**82.16**	**80.81**	**74.62**	**70.25**

Table 2. Performance comparison of five methods for *Transaction-order dependency*, *Timestamp-dependency*, and *Integer overflow* vulnerabilities using *accuracy*, *recall*, *precision*, and *F1-score* metrics, including state-of-the-art detection methods, neural network-based alternatives, and our method *ST-GNN*.'-' denotes not applicable.

Method	Transaction-order dependency				Timestamp-dependency				Integer overflow			
	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)
Smartcheck	-	-	-	-	51.32	37.25	39.16	38.18	58.48	54.20	55.68	54.96
Oyente	53.68	67.82	51.56	58.64	59.45	38.44	45.16	41.53	66.85	57.23	58.06	59.64
Mythril	-	-	-	-	61.08	41.72	50.00	45.49	77.25	58.49	56.20	53.26
Vanilla-RNN	49.57	47.86	43.10	42.69	49.77	44.59	51.59	45.62	52.10	50.26	48.52	49.76
GCN	72.35	70.02	63.04	59.96	74.21	75.97	68.35	71.96	68.34	67.82	64.39	66.37
GraphSA	**83.33**	**81.64**	**76.49**	**71.80**	**85.71**	**82.50**	**77.09**	**70.07**	**80.00**	**79.86**	**76.52**	**71.80**

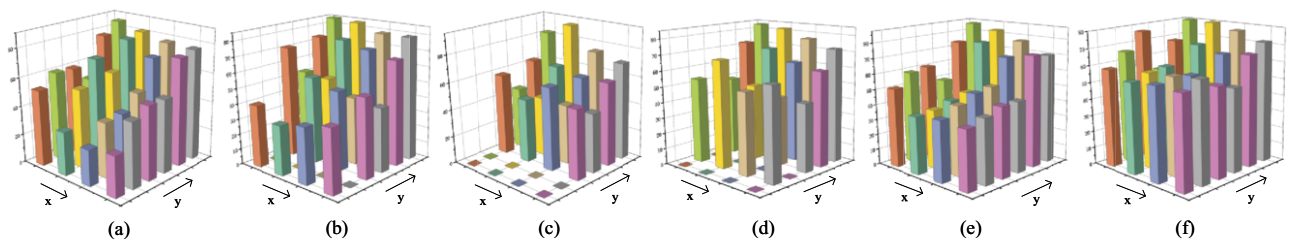

Figure 7. Visual comparison results. (a)-(f) respectively show the comparison results of six smart contract vulnerability detection methods in six aspects: *reentrancy*, *self-destruct*, *delegate-call*, *transaction-order dependence*, and *integer overflow*. In each subplot, the x-axis from left to right represents *accuracy*, *recall*, *precision*, and *F1-score*, and the y-axis from front to back respectively represents *Smartcheck*, *Oyente*, *Mythril*, *Vanillia-RNN*, *GCN*, and *ST-GNN*.

In addition, in other experiments, the *F1-score* of *Vanilla-RNN* and *GCN*, was also lower than that of GraphSA.

5 Related Work

The security of smart contracts is one of the most critical issues in blockchain. Many research teams have delved into the security problems of Solidity smart contracts. In terms of symbolic execution, mature tools have been developed, such as *Oyente* [20], *Maian* [22], and *Securify* [29], which can help researchers discover potential vulnerabilities in obscure paths within contracts. However, when dealing with complex and large smart contracts, there is a high risk of false negative rates. Because current tools require extensive computing resources and time, and may fail to execute due to "path explosion." As a result, researchers need to fine-tune the tools and verify the results of their detection. In dynamic analysis, recent work by [17] explores dynamical imitation attacks for retreency vulnerability. They divided the type of retreency vulnerability into three sections and constructed a relative attacker contract. However, the test data for input was generated with fuzzing test tools, which may have led to incomplete analysis. Deep learning technology has been applied in various ways to detect vulnerabilities, such as vulnerability detection based on natural language processing (e.g., *ContractWard* [30]), non-Euclidean graph-based vulnerability detection (e.g., *CCGraph* [35]), and image-based vulnerability detection (e.g., *R2-D2* [13]). Nonetheless, deep learning techniques still face several challenges in detecting smart contract vulnerabilities. For example, smart contract data has a highly intricate structure, and the model training process demands a substantial amount of computing resources and sample data. Additionally, researchers need to address the complexity and interpretability issues of deep learning models.

6 Conclusion

In this paper, we present a fully automated method for detecting vulnerabilities in smart contracts that combines deep learning and static analysis. In contrast to existing approaches, we aim to preserve the semantics and data flow characteristics of contracts to the greatest extent, and we validated the feasibility of using a novel contract tree for vulnerability detection. Experiments demonstrate that our approach outperforms state-of-the-art methods.

Acknowledgements

We would like to thank the referees for their comments, which helped improve this paper considerably.

This work was supported by the National Natural Science Foundation of China (Grant Nos. 61972360 and 62072392).

References

[1] A. Bahga and V.K. Madisetti, 'Blockchain platform for industrial internet of things', *Journal of Software Engineering and Applications*, **9**, 533–546, (2016).

[2] BeautyChain. The BEC Contract. https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d/.

[3] L. Bellomarini, M. Nissl, and E. Sallinger, 'Blockchains as knowledge graphs-blockchains for knowledge graphs (vision paper).', in *KR4L@ ECAI*, pp. 43–51. IOS Press, (2020).

[4] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, and N. Swamy, 'Formal verification of smart contracts: Short paper', in *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pp. 91–96. ACM, (2016).

[5] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, 'Vandal: A scalable security analysis framework for smart contracts', *arXiv preprint arXiv:1809.03981*, 1–28, (2018).

[6] V. Buterin, 'A next-generation smart contract and decentralized application platform', *white paper*, **3**, 1–2, (2014).

[7] C. Cangea, P. Veličković, N. Jovanović, T. Kipf, and P. Liò, 'Towards sparse hierarchical graph classifiers', *arXiv preprint arXiv:1811.01287*, 1–6, (2018).

[8] ConsenSys. Mythril. https://github.com/ConsenSys/mythril.

[9] C. Dannen, *Introducing Ethereum and solidity*, Springer, Brooklyn, USA, 2017.

[10] J. Feist, G. Grieco, and A. Groce, '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).

[11] W. Hamilton, Z. Ying, and J. Leskovec, 'Inductive representation learning on large graphs', *Advances in neural information processing systems*, **30**, 17–36, (2017).

[12] K. Harith. Etherscan-Verified contract list. https://etherscan.io/.

[13] H.T. Hsien-De and H.Y. Kao, 'R2-d2: Color-inspired convolutional neural network (cnn)-based android malware detections', in *2018 IEEE international conference on big data (big data)*, pp. 2633–2642. IEEE, (2018).

[14] T.N. Kipf and M. Welling, 'Semi-supervised classification with graph convolutional networks', *arXiv preprint arXiv:1609.02907*, 1–14, (2016).

[15] J. Kokina, R. Mancha, and D. Pachamanova, 'Blockchain: Emergent industry adoption and implications for accounting', *Journal of Emerging Technologies in Accounting*, **14**, 91–100, (2017).

[16] B. Kurdi, H. Alzoubi, I. Akour, and M. Alshurideh, 'The effect of blockchain and smart inventory system on supply chain performance: Empirical evidence from retail industry', *Uncertain Supply Chain Management*, **10**, 1111–1116, (2022).

[17] B. Li, Z. Pan, and T. Hu, 'Redefender: detecting reentrancy vulnerabilities in smart contracts automatically', *IEEE Transactions on Reliability*, **71**, 984–999, (2022).

[18] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, 'Combining graph neural networks with expert knowledge for smart contract vulnerability detection', *IEEE Transactions on Knowledge and Data Engineering*, **35**, 1296–1310, (2021).

[19] Z. Liu and J. Zhou, 'Vanilla graph neural networks', in *Introduction to Graph Neural Networks*, pp. 19–22. Springer, (2020).

[20] L. Luu, D.H. Chu, H. Olickel, P. Saxena, and A. Hobor, 'Making smart contracts smarter', in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 254–269. ACM, (2016).

[21] S. Nakamoto, 'Bitcoin: A peer-to-peer electronic cash system', *Decentralized business review*, 1–21, (2008).

[22] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, 'Finding the greedy, prodigal, and suicidal contracts at scale', in *Proceedings of the 34th annual computer security applications conference*, pp. 653–663. IEEE, (2018).

[23] K. Obscuren. Ethereum. https://github.com/ethereum/go-ethereum.

[24] S. Singh, S. Rathore, O. Alfarraj, A. Tolba, and B. Yoon, 'A framework for privacy-preservation of iot healthcare data using federated learning and blockchain technology', *Future Generation Computer Systems*, **129**, 380–388, (2022).

[25] SlowMist. Slowmist. https://hacked.slowmist.io/en/.

[26] W.J. Tann, X.J. Han, S.S. Gupta, and Y. Ong, 'Towards safer smart contracts: A sequence learning approach to detecting security threats', *arXiv preprint arXiv:1811.06632*, **1181**, 32–42, (2018).

[27] S. Temple. Alchemy. https://www.alchemy.com/.

[28] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, 'Smartcheck: Static analysis of ethereum smart contracts', in *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, pp. 9–16. ACM/IEEE, (2018).

[29] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, 'Securify: Practical security analysis of smart contracts', in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 67–82, (2018).

[30] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, 'Contractward: Automated vulnerability detection models for ethereum smart contracts', *IEEE Transactions on Network Science and Engineering*, **8**, 1133–1144, (2020).

[31] X. Wang, D. Hou, C. Tang, and S. Lv, 'A fuzzy testing method for gas-related vulnerability detection in smart contracts', in *Advances in Natural Computation, Fuzzy Systems and Knowledge Discovery: Proceedings of the ICNC-FSKD 2021 17*, pp. 407–418. Springer, (2022).

[32] Y. Wu, Z. Wang, X. Zhou, and J. Yao, 'The optimisation research of blockchain application in the financial institution-dominated supply chain finance system', *International Journal of Production Research*, 1–21, (2022).

[33] X. Zhao, Z. Chen, X. Chen, Y. Wang, and C. Tang, 'The dao attack paradoxes in propositional logic', in *2017 4th international conference on systems and informatics (ICSAI)*, pp. 1743–1746. IEEE, (2017).

[34] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, 'Smart contract vulnerability detection using graph neural network.', in *IJCAI*, pp. 3283–3290. Morgan Kaufmann, (2020).

[35] Y. Zou, B. Ban, and Y. Xue, 'Ccgraph: a pdggbased code clone detector with approximate graph matching', *202035th IEEE/ACM InternationalConferenceon Automated Software Engineering (ASE)*, **20**, 931–942, (2020).