Theory and Practice of Science and Technology
2025, VOL. 6, NO. 8, 254-258
DOI: 10.47297/taposatWSP2633-456948.20250608

Smart Contract Security Detection Based on Graph Neural Network

Ziteng Su

Tianjin University of Science and Technology, Tianjin, 300000, China

ABSTRACT
As the core component of blockchain applications, smart contracts are increasingly scrutinized for their security. Among various vulnerabilities, infinite loop flaws pose significant threats due to their hidden nature and potential for exhausting system resources. This paper proposes a static detection method based on Graph Convolutional Networks (GCNs), which transforms smart contracts into control flow and data flow graphs. Through graph-based modeling and vectorized encoding, semantic features such as loop structures and function dependencies are effectively captured. An improved GCN architecture is employed to identify potential infinite execution patterns through neighborhood aggregation and graph-level representation learning. Experimental results demonstrate that the proposed method achieves high accuracy and F1-score across real-world contract datasets, offering an effective and scalable solution for smart contract vulnerability analysis.

KEYWORDS
Smart contract; Infinite loop; Vulnerability detection; Graph Convolutional Network (GCN); Static analysis; Control flow graph; Deep learning

1 Introduction

1.1 Research background and significance

With the widespread adoption of blockchain technology in domains such as finance, the Internet of Things (IoT), and supply chain management, smart contracts—serving as one of its core components—have increasingly attracted significant attention from both researchers and industry practitioners. A smart contract is an automatically executable protocol deployed on a blockchain network, characterized by decentralization, immutability, and transparency. However, due to its immutable nature once deployed, any vulnerability may lead to irreversible economic losses. Security flaws within smart contracts have become frequent targets for malicious exploitation, particularly infinite loop vulnerabilities, which can be leveraged to consume system resources, cause service interruptions, or even launch denial-of-service (DoS) attacks, thereby posing severe threats to the stability of blockchain systems.

Consequently, achieving efficient and automated vulnerability detection prior to contract deployment has emerged as an urgent challenge. Traditional detection approaches often rely on manual auditing or rule-based static analysis tools, which are prone to false positives and false negatives when handling contracts with complex structures. In recent years, Graph Neural Networks (GNNs), owing to their strengths in modeling graph-structured data, have been extensively applied to program analysis and security detection tasks. As a representative model within the GNN family, the Graph Convolutional Network (GCN) effectively aggregates information from nodes and their neighbors in a graph, capturing control-flow and data-dependency features, and thus demonstrating considerable potential in smart contract security analysis.

This study proposes a novel method for detecting infinite loop vulnerabilities in smart contracts based on GNNs. By performing static analysis of the contract code, constructing a corresponding graph representation, and feeding it into a GCN model for training and inference, the method enables efficient identification of potential infinite loop patterns.

1.2 Current research status at home and abroad

The evolution of blockchain technology has progressed from the 1.0 era of digital currency, through the 2.0 era of smart contracts, to the 3.0 era characterized by cross-chain interoperability and multi-domain integration. As a representative technology of the 2.0 stage, smart contracts have been extensively deployed across various blockchain platforms; however, their security issues have become increasingly prominent.

At present, vulnerability detection techniques for smart contracts can be broadly categorized into three types: static analysis, dynamic execution, and symbolic execution. Among these, static analysis has emerged as the mainstream approach due to its high efficiency and independence from runtime environments. Nevertheless, conventional static analysis methods rely heavily on rule-based pattern matching, making them inadequate for covering complex logical scenarios and thus exhibiting clear limitations in detection capability.

In recent years, deep learning–based detection methods have gained traction, with Graph Neural Networks (GNNs) in particular demonstrating strong performance in modeling program control flow and data flow. Relevant studies abstract smart contracts into control flow graphs (CFGs) and data flow graphs (DFGs), construct corresponding graph structures,

and input these into Graph Convolutional Network (GCN) models for vulnerability detection. Existing work has shown that GCN models hold notable advantages in identifying highly concealed vulnerabilities. However, challenges remain, including simplistic model architectures and insufficient feature engineering. Therefore, designing graph representations that better capture the semantic characteristics of contracts, along with developing improved GCN architectures, has become a key direction for enhancing detection accuracy.

2 Fundamentals of blockchain and smart contract technology

Blockchain is a data structure in which data blocks are cryptographically linked in chronological order, with a distributed consensus mechanism ensuring data immutability, transparency, and public verifiability. It is primarily composed of the data layer, network layer, consensus layer, and incentive layer, which operate in synergy to enable data sharing, trustworthy transmission, and consistency.

A smart contract, deployed on a blockchain, is a piece of computational code capable of automatically executing predefined logic. Once specific conditions are met, the contract autonomously performs the corresponding operations without human intervention, thereby improving system efficiency and reducing trust-related costs. Ethereum is the first widely adopted platform to support smart contracts. Its contract language, Solidity, allows developers to flexibly define complex business logic. Contract execution relies on a virtual machine (such as the Ethereum Virtual Machine, EVM) and consumes Gas resources to quantify computational workload. If a contract contains a non-terminating loop structure, it may lead to Gas exhaustion, causing transaction failure and potentially triggering a network-wide denial-of-service (DoS) condition. Consequently, effective identification and prevention of infinite loop vulnerabilities have become one of the critical challenges in smart contract security.

Moreover, the complexity of blockchain smart contracts is not limited to control logic; it also encompasses factors such as dependencies among state variables, event-driven mechanisms, and cross-contract invocation. These factors not only increase the technical difficulty of vulnerability detection but also impose higher demands on the expressiveness and modeling capability of detection systems.

3 Analysis of infinite loop vulnerabilities in smart contracts

Infinite loop vulnerabilities represent a relatively concealed yet highly detrimental category of security risks within smart contracts. They are primarily caused by improper configuration of loop conditions in for and while statements or by recursive calls in callback functions. Once the execution enters an infinite loop, the smart contract continuously consumes Gas resources until failure occurs.

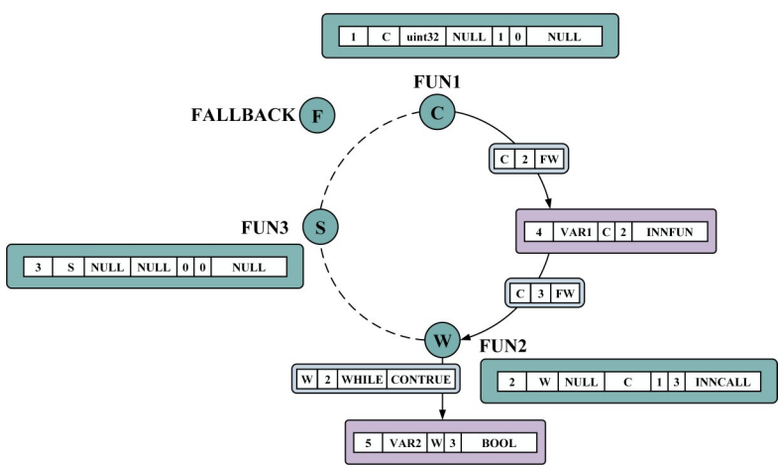

Figure 1

During smart contract development, if the termination condition of a loop is not well-defined or depends on external input without effective constraints, the program is prone to becoming trapped in an infinite loop. For instance, when a loop variable exceeds the maximum value of its type (such as uint8 or uint16) and wraps around, the absence of an exit mechanism can lead to a dead loop. Similarly, if the condition of a while loop remains perpetually true, the contract will fail to terminate. Furthermore, if a fallback function contains recursive calls, it can easily form an infinite execution path. Infinite loops may also result from abnormal recursive patterns. In certain smart contracts, state updates are implemented by invoking functions of the same contract or external contracts. Without appropriate restrictions on call depth or conditions, this can easily create a callback chain, ultimately exhausting system resources.Figure 1 illustrates the control flow and call dependency structure of an infinite loop vulnerability caused by recursive calls to a fallback function, forming a callback chain.

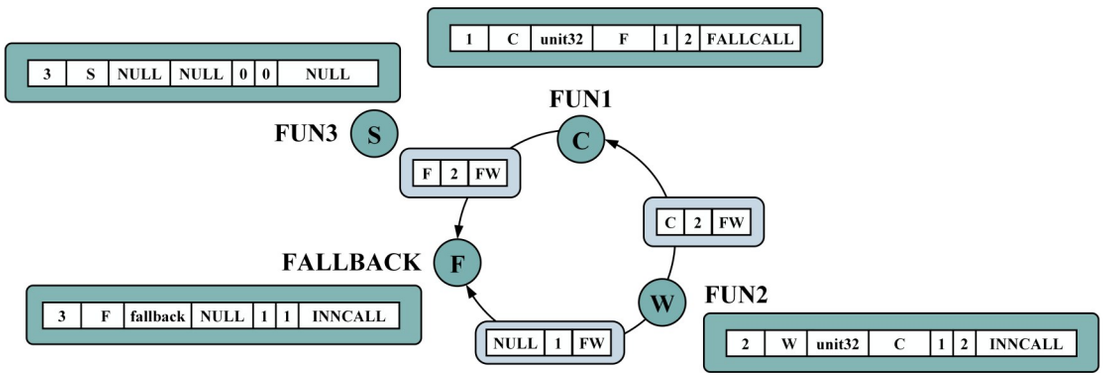

Figure 2

Figure 2 illustrates the control flow and data dependency graph of an infinite while-loop vulnerability with a constant true condition.

Traditional detection methods, such as rule-based static analysis tools, often struggle to identify these deep control-structure issues. In contrast, transforming a smart contract into a graph representation enables effective capture of loop control paths, function dependencies, and data-flow jump patterns through the relationships among nodes and edges. This approach provides strong support for automated vulnerability detection.

To further understand vulnerability behavior, infinite loop samples can be subjected to pattern induction, summarizing typical infinite loop structure templates. Examples include: loop counters not being updated, exit conditions that are perpetually true, and abnormal nested recursion. Labeling these patterns in the training dataset enhances the model's target-specific learning capabilities.

4 Detection method

This study proposes a smart contract infinite loop vulnerability detection approach that integrates static graph modeling with graph neural network (GNN) learning. The overall workflow can be divided into three key stages: graph structure generation, feature encoding, and model training & inference.

4.1 Graph structure generationIn the graph structure generation stage, the system employs static analysis tools to perform syntactic parsing and semantic abstraction of the contract source code, transforming it into a Control Flow Graph (CFG) and a Data Flow Graph (DFG). In the CFG, nodes represent function calls, loop statements, or control conditions, while edges denote control jump relationships between basic blocks. In the DFG, nodes reflect variable declarations, assignments, and operational behaviors, whereas edges depict data dependency paths between variables. To enhance representational capacity, semantic annotations are added for key control structures (e.g., for / while loops), variable operation patterns, and function call paths.

4.2 Graph vector encodingIn the graph vector encoding stage, the system extracts features for each graph node and its associated edges, converting them into fixed-length vector representations. Node features include syntactic type (e.g., FUN, VAR, IF), operator type, function scope, and positional context. Edge attributes encompass edge type (e.g., CALL, FOR, ASSIGN), directionality, and contextual weights. All features are concatenated into fixed-dimensional tensors via one-hot encoding, ensuring semantic consistency and model interpretability. This process is implemented through the graph2vec module, ultimately producing structured graph tensor data for downstream model usage.

4.3 Model training and inferenceThe model training and inference stage adopts a Graph Convolutional Network (GCN) architecture for graph representation learning. The core principle of GCN can be summarized as a neighborhood aggregation – based local perception mechanism: during each propagation layer, a node updates its representation by performing a weighted aggregation of its own feature vector and those of its neighbors. The mathematical formulation of a standard two-layer GCN can be expressed as follows:

$$R = D^{-\frac{1}{2}}(A + I)D^{-\frac{1}{2}}$$

$$H^{(l+1)} = \sigma(RH^{(l)}W^{(l)})$$

In this context, A refers to the processed adjacency matrix, which incorporates self-loops and has undergone normalization. H(0) denotes the initial node feature matrix at the first layer, while H(0) represents the trainable weight parameters of the current layer. σ denotes the activation function, such as the ReLU function. This operation, essentially a neighborhood aggregation based on the spectral domain approximation, achieves a balance between computational efficiency and representational capability.

In the proposed system, the primary responsibility is to perform deep feature learning on the contract graph structure and to conduct vulnerability classification. Based on the characteristic requirements of infinite loop vulnerabilities in smart contracts, we implemented and compared four representative graph neural network models: the original GCN, an

improved GCN, a multi-graph convolution network, and a graph attention network.

Within this system, the original GCN serves as the most fundamental graph convolutional neural network implementation. Its primary role is to aggregate deep features from the statically extracted contract graph structure and to perform preliminary classification. The corresponding implementation is located in models/gcn_origin. py. The core logic consists of only two graph convolution layers: the first layer maps the original discrete node features into a latent representation space, while the second layer projects the latent representation into the final class space. The forward propagation process, defined in forward(x, adj), sequentially invokes these two convolution layers, inserting ReLU activation and Dropout operations between them to enhance non-linear expressiveness and prevent overfitting.

Furthermore, the GCN_ORIGIN model employs a global max-pooling strategy: after the second convolution layer, the outputs of all nodes are aggregated to form a fixed-length graph-level representation, which is subsequently used by the classifier or the loss function for final prediction.

From a mathematical perspective, each graph convolution operation in the original GCN can be regarded as a normalized aggregation over the adjacency matrix,formulated as: $\hat{A} = D^{-\frac{1}{2}}(A + I)D^{-\frac{1}{2}}$Given the node feature matrix XXX, the aggregation is followed by a linear transformation yielding：

$H^{(l+1)} = \sigma(RH^{(l)}W^{(l)})$。 In this project, the ReLU function is selected as the activation function, followed immediately by a Dropout layer after the first graph convolution layer. The output of the second layer is directly subjected to global max pooling to obtain the graph-level representation hGh_GhG . This design retains the simplicity and computational efficiency of the graph convolution layers while enabling the capture of local structural dependencies between nodes to a certain extent. However, due to the fixed depth of two layers, the GCN_ORIGIN model has inherent limitations in mining deep topological information from graphs. It can effectively learn only from one-hop or two-hop neighboring nodes, making it less capable of representing complex multi-level invocation chains and deeply nested loop structures in smart contracts.

During the system integration phase, the GCN_ORIGIN model is encapsulated within the main control script InfiniteLoopDetector.py. When the command-line argument --model gcn_origin is specified, the system invokes this class to perform training and testing. In the training process, the program first uses DataReader and GraphData to convert node_features_onehot and adj_list into PyTorch tensors, which are then batch-loaded into a DataLoader. In each iteration, the graph-level feature set $X_{graph}$and adjacency matrix A_{adj} are fed into the GCN_ORIGIN.forward method, where the cross-entropy loss is computed and backpropagation is performed.

In the testing phase, the graph representation obtained through global max pooling is used for classification prediction, with performance evaluated based on accuracy, recall, precision, and F1-score metrics. Experimental results indicate that while GCN_ORIGIN achieves approximately 57.8% accuracy on the small-scale smart contract dataset, its ability to identify deep invocation chains and cyclical dependencies remains limited. This shortcoming provides a clear optimization direction for subsequent model improvements.

5 System design and implementation

5.1 System architecture

The proposed system comprises four core modules: graph structure extraction, graph vector conversion, GCN-based model training and inference, and result output and evaluation.

In the graph structure extraction stage, the system employs the AutoExtractGraph tool to perform function-level decomposition and structured abstraction of the source code, identifying loop structures, function calls, conditional branches, and variable dependencies. All structural information is stored in a standardized text format to facilitate subsequent processing and conversion.

The graph vector conversion module encodes the extracted structures into a unified graph vector representation for use in training and inference of the graph neural network model. The encoding scheme incorporates semantic labels of nodes, invocation types, variable operation categories, and edge jump markers. These features are converted into fixed-dimensional vector representations via the vec2onehot module, ensuring consistency and compatibility with the model ' s input requirements.

The GCN model adopts a dual-layer graph convolution architecture, combined with ReLU activation and Dropout mechanisms to enhance generalization capability. In each layer, the GCN aggregates features from neighboring nodes to iteratively learn node representations, which are subsequently processed by a fully connected layer to output the predicted label indicating whether an infinite loop vulnerability is present.

The system is implemented using the PyTorch framework, with a modular and extensible design that accommodates smart contract detection tasks of varying scales and types. It supports batch loading and parallel inference to improve processing efficiency.

5.2 Experimental results and performance evaluation

The experimental dataset was derived from annotated smart contract samples on the VNT blockchain, comprising 216 contracts, approximately 40% of which contain infinite loop vulnerabilities. Model training was conducted using 5-fold cross-validation. Evaluation metrics included accuracy, recall, and F1-score.

Experimental results show that the GCN model achieved 89.7% accuracy and an F1-score of 87.3% on the test set, demonstrating a clear advantage over traditional static analysis methods in identifying contracts with complex control structures and highly concealed vulnerabilities.

In terms of resource consumption, the system achieved an average inference time of approximately 0.12 seconds per contract in a single-machine environment, with parallel processing capability suitable for deployment in medium- to large-scale blockchain platforms. Further performance analysis indicates that the system maintains stable convergence speed and inference efficiency across varying dataset sizes, validating its feasibility for practical deployment and large-scale adoption.

Enhancement opportunities remain in areas such as edge attribute augmentation and node label optimization. In extended experiments, the effects of varying GCN depth and activation functions were also examined. Results indicate that increasing the number of GCN layers can improve representational capacity to some extent, albeit at the cost of increased training time. The choice of activation function had relatively minor impact on detection performance, with ReLU exhibiting favorable stability and training efficiency.

To further compare the performance of different models, Table 1 summarizes the results of four representative graph neural network models—GCN_ORIGIN, GCN_MODIFY, MGCN, and GAT—on the smart contract vulnerability detection task.

Table 1 Comparison of four models in infinite loop vulnerability detection

MODEL	Accuracy(%)	Recall (%)	Precision(%)	F1(%)
GCN_ORIGIN	57.8±1.2	74.3±1.0	59.5±1.4	66.0±1.3
GCN_MODIFY	93.3±0.8	92.6± 0.9	92.2±0.7	92.4±0.8
MGCN	94.1±0.7	94.8 ± 0.6	93.5±0.5	94.1±0.6
GAT	93.7±0.9	93.2±1.1	93.4±0.8	93.3±0.9

6 Conclusion and future work

This paper presents a method for detecting infinite loop vulnerabilities in smart contracts based on Graph Convolutional Networks (GCNs). By constructing graph structures through static analysis, combined with graph vector encoding and GCN modeling, the proposed approach achieves effective identification of potential vulnerabilities within complex control structures.

Experimental results demonstrate that the method offers significant advantages in both accuracy and efficiency, making it suitable for large-scale smart contract vulnerability detection tasks. Future research could incorporate multi-perspective graph structure modeling, heterogeneous graph neural networks, and graph attention mechanisms to further enhance the robustness and generalization capability of the model.

Moreover, the system framework exhibits strong extensibility, enabling future adaptation to the detection of other types of vulnerabilities, thereby promoting the evolution of smart contract security analysis towards greater automation and intelligence. Potential future enhancements include support for cross-language contracts, improving the system's versatility in multi-platform environments, and integration with blockchain development toolchains to enable automatic interaction between vulnerability detection and contract development. Such integration could lower the development threshold while improving the overall security assurance level.

References

[1] Kolvart M, Poola M, Rull A. Smart contracts[J]. The Future of Law and etechnologies, 2016: 133-147.

[2] Khan S N, Loukil F, Ghedira-Guegan C, et al. Blockchain smart contracts: Applications, challenges, and future trends[J]. Peer-to-peer Networking and Applications, 2021, 14: 2901-2925.

[3] Chess B, McGraw G. Static analysis for security[J]. IEEE security & privacy, 2004, 2(6): 76-79.

[4] Ayewah N, Pugh W, Hovemeyer D, et al. Using static analysis to find bugs[J]. IEEE software, 2008, 25(5): 22-29.

[5] Subramanian V. Deep Learning with PyTorch: A practical approach to building neural network models using PyTorch[M]. Packt Publishing Ltd, 2018.

[6] Ghorbani M, Baghshah M S, Rabiee H R. MGCN: semi-supervised classification in multi-layer graphs with graph convolutional networks[C]// Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining. 2019: 208-211.

[7] Van Rossum G, Drake F L. An introduction to Python[M]. Bristol: Network Theory Ltd., 2003.

[8] Parmar N, Vaswani A, Uszkoreit J, et al. Image transformer[C]//International conference on machine learning. PMLR, 2018: 4055-4064.