GNN-based Ethereum Smart Contract Multi-Label Vulnerability Detection

Yoo-Young Cheong
Department of Artificial Intelligence Application
Kwangwoon University
Seoul, South Korea
yycheong@kw.ac.kr

La Yeon Choi
Department of Artificial Intelligence Application
Kwangwoon University
Seoul, South Korea
chlfkdus123@kw.ac.kr

Jihwan Shin
Department of Artificial Intelligence Application
Kwangwoon University
Seoul, South Korea
shinjihwan1997@kw.ac.kr

Taekyung Kim
department of Big Data Analytics
KyungHee University
Seoul, South Korea
tk_kim@khu.ac.kr

Jinhyun Ahn
department of Management Information Systems
Jeju National University
Jeju, South Korea
jha@jejunu.ac.kr

Dong-Hyuk Im
School of Information Convergence
Kwangwoon University
Seoul, South Korea
dhim@kw.ac.kr

***Abstract*— Smart contracts are self-executing programs that are executed on blockchain platforms, and they have been widely used in recent years. However, malicious exploitation of the characteristics of smart contracts has become a pressing problem in blockchain security. Most of the existing methods have the drawback of detecting only a single type of vulnerability. To solve this problem, this study proposes a model for detecting multiple vulnerabilities in smart contracts. We preprocessed the data and transformed the Opcodes of smart contracts' source code into a control flow graph. We then extracted node features that are suitable to be the input of a graph neural network using Sent2Vec and performed graph classification. The proposed model was evaluated using real smart contracts, and the experimental results demonstrated that the proposed model can simultaneously detect multiple vulnerabilities with high performance.**

Keywords—blockchain, smart contract, vulnerability detection, multi-label classification

I. Introduction

Blockchain is basically a distributed and shared ledger that verifies and records transaction-related information without a central system. Ethereum [1], developed by Vitalik Buterin, is a blockchain-based open-source distributed computing platform and operating system that features smart contracts. Many studies related to the Ethereum network have recently been underway [2]. Smart contracts are self-executing programs [1] that run on a blockchain. They encode the terms of a contract using a source code and can implement arbitrary rules to manage the asset. Smart contracts simplify complex distributed applications (Dapp) by enabling automatic execution of the terms of a contract [3]. Smart contracts are deployed on a blockchain network according to the consensus protocol, and the rules of the contract produce the same results regardless of which node on the blockchain network executes them. A smart contract that has been mined and uploaded cannot be modified due to blockchain integrity. In other words, even if a programming defect has been identified, the uploaded smart contract cannot be updated. Therefore, a malicious user can exploit an unsafe smart contract, causing the contract owner to incur a significant loss [4]. For example, an Ether loss of about 60 million dollars occurred in June 2016 due to the reentrancy vulnerability problem [6] of the DAO smart contract [5]. Therefore, the vulnerability of smart contracts needs to be examined before they are deployed.

Developers typically test for vulnerabilities in smart contracts before executing them to ensure their security. Manual code analysis is inefficient because there are many smart contracts, and there are various types of vulnerabilities. Therefore, studies are currently being conducted on the detection of smart contract vulnerabilities through deep learning [7][8]. However, most of the previous studies have been based on the binary classification method, and this method can only detect whether or not a vulnerability exists. Therefore, this study utilizes the multi-label classification method to accurately and effectively detect multiple smart contract vulnerability types to strengthen the security of blockchain platforms. Our main contributions are as follows.

- Unlike existing models that perform binary classification to classify a single smart contract vulnerability, we propose a multi-label classification model that can identify multiple types of smart contract vulnerabilities simultaneously.

- We can determine the presence of vulnerabilities by learning both the semantic and structural information of the source code based on the control flow graph of the smart contract EVM bytecode.

II. Background

A. *Solidity Language and Compiling Solidity*

Solidity is an object-oriented high-level language for implementing smart contracts [9]. Fig. 1 reports an example of a smart contract to implement a bank written in the solidity language. The *balance* on the source code is the internal state of the smart contract, and the function deposit allows the user to deposit the arbitrary amount of currency into the virtual account. In addition, through function withdraw, the user to get back a certain amount of Ether previously deposited. Solidity provides various basic elements that can interoperate with the blockchain environment, and there are various elements that are not in the examples.

```
pragma solidity ^0.8.0;

contract SimpleBank {
   mapping(address => uint256) private balances;
   function deposit() public payable {
      require(msg.value > 0, "Deposit amount must be greater
than 0");
      balances[msg.sender] += msg.value;
   }
   function withdraw(uint256 amount) public {
      require(balances[msg.sender] >= amount, "Insufficient
balance");
      balances[msg.sender] -= amount;
      payable(msg.sender).transfer(amount);
   }
   function getBalance() public view returns (uint256) {
      return balances[msg.sender];
   }
}
```

Fig. 1. Example of Solidity code

To execute smart contracts written in Solidity on the Ethereum blockchain, the source code needs to be compiled into an Ethereum bytecode that can be executed on the Ethereum virtual machine (EVM). As shown in Fig. 2, the Ethereum bytecode is represented as a valid hexadecimal sequence, and it can be parsed into Opcodes, which are minimum instructions that can be executed by the EVM [10][11]. The complete list of opcodes with their semantics is defined in Ethereum's yellow paper [11]. Every opcode pushes or pops a certain number of elements from/to the stack, and it can either access memory, get information about the execution environment or interact with other blockchain smart contracts.

```
6060604052600060095564174281070060a55620186a0600c556201
86a0600d556000600e556000600f55738216a5958 ... c877c00029
```

EVM Bytecode example

```
PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x04 CALLDATASIZE
LT PUSH2 0x0175 JUMPI PUSH1 0x00 CALLDATALOAD ...
DUP2 ADD DUP3 MUL SWAP2 POP DUP1 PUSH1 0x01 ADD SWAP1
POP PUSH2 0x3057 JUMP JUMPDEST DUP4 SWAP5 POP POP POP
POP POP SWAP5 SWAP4 POP POP POP POP JUMP STOP
```

Opcodes example

Fig. 2. Example of EVM Bytecode and Opcodes

B. *Smart contract vulnerabilities*

Smart contracts are considered vulnerable when an attacker can exploit them. Six well-known vulnerabilities [12] are detected in this study.

Integer Overflow and Underflow In Solidity, integer types have a maximum value and a minimum value. Overflow and underflow problems occur when the integer value exceeds the maximum integer value (overflow) or is less than the minimum value (underflow).

Transaction-Ordering Dependence (TOD) A blockchain block contains the transaction set, so the blockchain state is updated each time. At this point, there may be a mismatch between the state of the smart contract the user intends to call and the actual state of the smart contract when it is executed. Only the miner can determine the order of transactions, that is, the update order. TOD is a vulnerability that occurs as a result of changing the order of transactions.

Timestamp Dependence This vulnerability occurs when a contract uses the block timestamp as a condition for calling the contract to execute an important operation. When mining a block, the miner sets the timestamp for the block. At this point, the miner can change this value by about 900 seconds. Therefore, the miner can exploit this vulnerability if the contract is dependent on the timestamp.

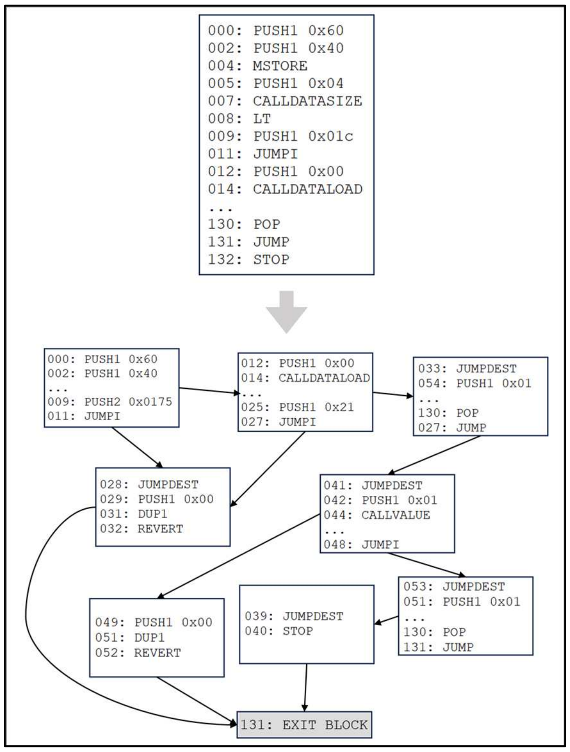
Mishandled Exception (Call stack's depth attack) In Ethereum, a smart contract can call another smart contract via an instruction or call a function. Here, if the call stack depth exceeds the threshold value, the contract is terminated, and false is returned after reverting the contract's state. Therefore, the caller contract must handle exceptions by checking whether the call was properly executed.

Reentrancy Vulnerability When a smart contract calls another contract, the execution waits until the call is completed. If the smart contract has a reentrancy vulnerability, an attacker can exploit this vulnerability to force repetitive execution of a code like a recursive function call.

C. *Control Flow Graph*

The control flow graph (CFG) is a directional graph of G=(V, E). Each vertex represents a basic block, which is a sequence of program instructions with the entry point (the first instruction executed) and the exit point (the last instruction executed). The directional edge represents the control flow path [13]. If an EVM bytecode is transformed into a CFG, the call relationships between the basic blocks can be defined according to the jump instructions. EtherSolve[10] parses an EVM bytecode into an

Identify applicable funding agency here. If none, delete this text box.



```
000: PUSH1 0x60
002: PUSH1 0x40
004: MSTORE
005: PUSH1 0x04
007: CALLDATASIZE
008: LT
009: PUSH1 0x01c
011: JUMPI
012: PUSH1 0x00
014: CALLDATALOAD
...
130: POP
131: JUMP
132: STOP
```

```
000: PUSH1 0x60
002: PUSH1 0x40
...
009: PUSH2 0x0175
011: JUMPI
```

```
012: PUSH1 0x00
014: CALLDATALOAD
...
025: PUSH1 0x21
027: JUMPI
```

```
033: JUMPDEST
054: PUSH1 0x01
...
130: POP
027: JUMP
```

```
028: JUMPDEST
029: PUSH1 0x00
031: DUP1
032: REVERT
```

```
041: JUMPDEST
042: PUSH1 0x01
044: CALLVALUE
...
048: JUMPI
```

```
049: PUSH1 0x00
051: DUP1
052: REVERT
```

```
039: JUMPDEST
040: STOP
```

```
053: JUMPDEST
051: PUSH1 0x01
...
130: POP
131: JUMP
```

```
131: EXIT BLOCK
```

Fig. 3. Control Flow Graph

Opcode sequence and aims to extract an accurate CFG by resolving jump targets based on symbolic execution. EtherSolve was used in this research to extract the CFG of a smart contract EVM bytecode.

D. Multi-label Classification

Multi-label classification has the characteristic where a single instance has one or more labels. Many deep learning multi-label classification models have been proposed in various areas, such as image classification [14, 15] and text classification [16]. Among recent studies on smart contract vulnerability detection, ContractWard [17] trained six classifiers after extracting N-Gram features from smart contract Opcodes and classified six types of vulnerabilities. Although this method enables multi-label classification, it has high algorithm complexity because it trains multiple classifiers for different vulnerabilities. Moreover, this method cannot capture simultaneously occurring label dependencies. In this study, we utilize multi-label classification to accurately and effectively detect various vulnerabilities in smart contracts simultaneously by training our model with the CFG generated from Opcodes and six labels.

III. Methods

This study aims to more accurately detect the presence of multiple vulnerabilities in smart contracts. As shown in Fig. 4, the proposed vulnerability detection model consists of three main steps: graph extraction, sentence embedding, and graph classification.

A. Graph Extraction

After parsing the source code into Opcodes, the CFG is extracted using EtherSolve. Opcodes that change the program's control flow are divided into basic blocks, which become the nodes in the graph. The JUMP, JUMPI, STOP, RETURN, INVALID, and SELFDESTRUCT Opcodes mark the end of a basic block, while the JUMPDEST Opcode marks the beginning of a new block. Once the code is partitioned into basic blocks, edges are designed, and the destination of the jump immediately preceding the PUSH opcode becomes the value of the PUSH Opcode. If the PUSH Opcode does not exist, the state of the stack is updated by executing only the Opcodes that partially interact with the jump address to find the destination from the symbolic stack.

B. Sentence Embedding

Each vertex of the CFG corresponds to a sequence of program instructions. The Opcode sequences, which are basic blocks, were treated and processed as single sentences in this

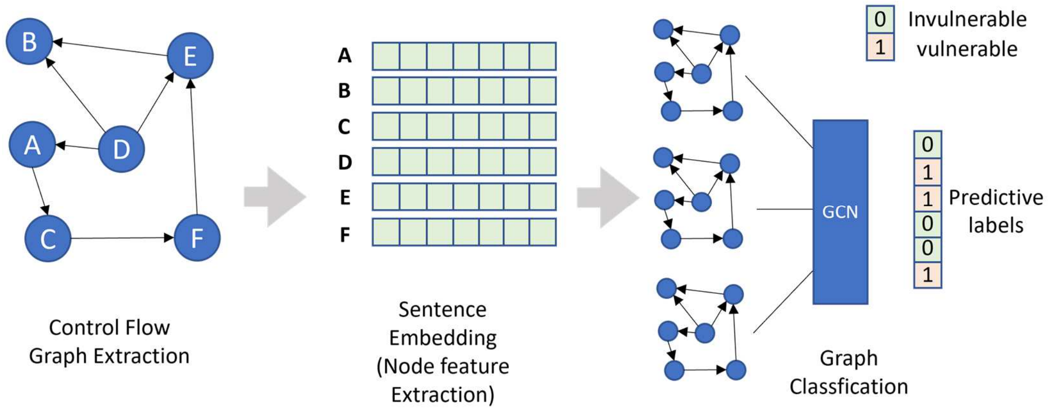


Fig. 4. The model structure for Ethereum smart contract vulnerability detection

study. Each Opcode sequence is transformed into a fixed-length vector by applying Sent2Vec [18], which is a sentence embedding technique. Sent2Vec is an unsupervised learning model for general sentence embedding that has extended the CBOW model of Word2Vec to sentence-level embedding. In our study, we obtain the node features of the CFG by transforming the Opcode sequences into 128-dimensional vector representations using the Sent2Vec model.

C. Graph Classification

Graph classification predicts the labels of the graph, and it requires fusion learning of global information, including the structural information of the graph and the attributes of nodes and edges. We utilize graph neural networks (GNNs) to train the graph structure. After generating fixed-size node features of the graph using sentence embedding, we train a GNN model with these features. First, we extract the global information through the convolutional and pooling layers. After pooling, the global information of the graph is aggregated to perform classification.

IV. Experiments

In this section, Ethereum smart contracts with source codes verified by Etherscan.io [19] are collected to evaluate the performance of our proposed model. We aimed to verify whether the proposed method can effectively and simultaneously detect six vulnerabilities in smart contract source codes.

A. Dataset

The experiment was conducted using a smart contract dataset verified by Etherscan.io. We randomly collected 4,000 contracts from publicly available open-source smart contracts that were compiled and deployed on the Ethereum network and have transaction information. After obtaining the bytecode and relevant information, such as address and compiler version, from the open-source smart contracts, we processed them to remove duplicates. We excluded duplicate smart contracts because there are cases where existing smart contracts are reused and deployed multiple times.

B. Comparison with the Existing Machine Learning-based Methods

We used the XGBoost classifier, Decision Tree classifier, Random Forest Classifier, and SVM classifier as comparison models for the same test set. Each model was trained with six classifiers for the six vulnerabilities. The six binary classification results for each model were consolidated to produce the final results. Labels were assigned for all smart contracts using Oyente [12]. Each contract consists of six labels in a format similar to [0 0 1 0 0 0]. In this study, we assumed that the labels generated by Oyente are reliable. Accuracy and macro-F1 were selected as the performance evaluation metrics for the experiment. Macro-F1 is a measurement that is used to evaluate multi-label classification, and the weight of each category is identical.

TABLE I. Vulnerability Detection Results

Method	Accuracy	Macro-F1
Decision Tree	0.81	0.79
SVM	0.87	0.85
XGBoost	0.87	0.85
Random Forest	0.88	0.86
Our approach	**0.90**	**0.89**

Table I shows that the accuracy and macro-F1 score of our proposed model are greater than that of the existing classifiers for the same data set. Hence, it demonstrates that our proposed model has better performance than the existing classifiers.

V. Conclusion

In this study, we proposed a method for detecting multi-label vulnerabilities by effectively obtaining the semantic and control flow information from smart contract source codes. We compared the performance of our proposed multi-label vulnerability classification detection model with that of several machine learning models, and the results demonstrate that our model outperforms the other models. Our proposed approach is effective in simultaneously identifying six vulnerabilities in smart contracts. In the future, we plan to conduct research on detecting vulnerabilities that have not been labeled, besides the six vulnerabilities.

Acknowledgment

The work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2021-0-00231, Development of Approximate DBMS Query Technology to Facilitate Fast Query Processing for Exploratory Data Analysis, 50%). This work was also supported by the National Research Foundation of Korea(NRF) granted by the Korea government(MSIT) (No.NRF-2021R1F1A1054739, 50%).

References

[1] C. Dannen, Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners, 1st ed., Apress, 2017.

[2] Y. Y. Cheong, G. T. Kim, and D. H. Im. "Ethereum Phishing Scam Detection based on Graph Embedding and Semi-Supervised Learning." KIPS Transactions on Computer and Communication Systems, vol. 12, no. 5, pp. 165-170, 2023.

[3] A. M. Antonopoulos and G. Wood, Mastering Ethereum: Building Smart Contracts and Dapps. Sebastopol, CA, USA: O'Reilly Media, Inc., 2018.

[4] Z. A. Khan and A. Siami Namin, "Ethereum Smart Contracts: Vulnerabilities and their Classifications," 2020 IEEE International Conference on Big Data (Big Data), Atlanta, GA, USA, 2020, pp. 1-10.

[5] The DAOsmart contract,Website, 2016. [Online]. Available: http:// etherscan.io/address/ 0xbb9bc244d798123fde783fcc1c72d3bb8c189413

[6] D. Z. Morris. Blockchain-based venture Capital Fund Hacked for $60 Million. 2016. [Online]. Available: https://fortune.com/2016/06/18/ blockchain-vc-fund-hacked/

[7] 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," in IEEE Transactions on Knowledge and Data Engineering, vol. 35, no. 2, pp. 1296-1310, 1 Feb. 2023

[8] P. Qian, Z. Liu, Q. He, R. Zimmermann and X. Wang, "Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models," in IEEE Access, vol. 8, pp. 19685-19695, 2020.

[9] Ethereum. Solidity documentation. Website. [Accessed: 2023-09-21]. [Online]. Available: https://solidity.readthedocs.io/

[10] F. Contro, M. Crosara, M. Ceccato and M. D. Preda, "EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode," 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), Madrid, Spain, 2021, pp. 127-137

[11] G.Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Project Yellow Paper, vol. 151, no. 2014, pp. 1–32, 2014.

[12] L. Luu, D. H Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16), New York, NY, USA, 2016, pp. 254–269.

[13] A. Viet Phan, M. Le Nguyen and L. Thu Bui, "Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction," 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI), Boston, MA, USA, 2017, pp. 45-52.

[14] D. Huynh, E. Elhamifar. "Interactive multi-label cnn learning with partial labels," IEEE/CVF Conference on Computer Vision and Pattern Recognition(CVPR), Virtual, 2020, pp. 9423-9432.

[15] Z. M. Chen. X. S. Wei. P. Wang, and Y. Guo. "Multi-label image recognition with graph convolutional networks," IEEE/CVF conference on computer vision and pattern recognition(CVPR), Long Beach, CA, USA, 2019, pp. 5177-5186.

[16] H. Alhuzali. S. Ananiadou. "SpanEmo: Casting multi-label emotion classification as span-prediction," 2021. [Online]. Available: https://arxiv.org/abs/2101.10038.

[17] W. Wang, J. Song, G. Xu, Y. Li, H. Wang and C. Su, "ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts," in IEEE Transactions on Network Science and Engineering, vol. 8, no. 2, pp. 1133-1144, 1 April-June 2021

[18] M. Pagliardini. P. Gupta, and M. Jaggi. "Unsupervised learning of sentence embeddings using compositional n-gram features," 2017. [Online]. Available: https://arxiv.org/abs/1703.02507.

[19] Etherscan.io. Etherscan. [accessed: 2023-09-21]. [Online]. Available: https://etherscan.io/