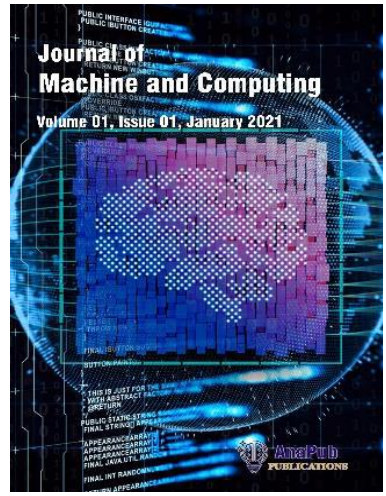
Journal of
Machine and Computing
Volume 01, Issue 01, January 2021
AnaPub
PUBLICATIONS

A Study on the Security Weakness Detection of Solidity Smart Contracts using Graph Neural Networks on Blockchain Platforms

Sunghyun Kim [1], Seunggi Jung [1], Yunsik Son [2*], Yangsun Lee [1*]

[1] Department of Computer Engineering, Seokyeong University,
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 02713, Korea

[2] Dept. of Computer Science and Engineering, Dongguk University,
3-ga, Pil-dong, Jung-gu, Seoul 04620, Korea

[2*] sonbug@dongguk.edu; [1*]yslee@skuniv.ac.kr

Abstract

Blockchain is a distributed ledger technology that allows users to record and share information safely and transparently. A smart contract is a contract decided based on a blockchain and is a program that automatically executes or executes contract terms. Smart contracts improve the transparency and reliability of transactions by utilizing the tampering prevention function of blockchain technology. Software security vulnerability refers to the fundamental cause of vulnerabilities caused by logical errors, bugs, and mistakes that can be defective in software development. To prevent software security accidents, security weaknesses must be analyzed before the program is distributed. Smart contract codes that operate on ethereum, a blockchain-based framework, can have security vulnerabilities inside the code. When the contract is completed and the block is created, the chaincode cannot be arbitrarily modified, so the security weakness must be analyzed before execution.

In this paper, we used deep learning's graph neural network (GNN) to detect security vulnerabilities in solidity codes. To analyze security vulnerabilities in solidity code, we defined eight types of security weakness items, converted the solidity code into graph data. In order to represent both the structural elements of the program, the control flow, and the data flow, the solidity code was converted into an abstract syntax tree (AST) and the graph information required for GNN learning was extracted from AST to convert the solidity code into a graph. Next, after generating several datasets for training GNN models by integrating these graph data and their properties with labels, it is possible to detect whether security vulnerabilities exist in the solidity code through GNN learning. This method performs security weakness detection more effectively than conventional rule-based methods.

Keywords: *Blockchain, Smart Contract, Security vulnerability, Solidity, Ethereum, Security Weakness Analyzer, Graph Neural Networks(GNN), Graph Convolution Network(GCN)*

1. Introduction

Blockchain is a distributed ledger technology that allows users to record and share information safely and transparently. A smart contract is a contract decided based on a blockchain and is a program that automatically executes or executes contract terms. Smart contracts improve the transparency and reliability of transactions by utilizing the tampering prevention function of blockchain technology [1-5].

Software security vulnerability refers to the fundamental cause of vulnerabilities caused by logical errors, bugs, and mistakes that can be defective in software development. To prevent software security accidents, security weaknesses must be analyzed before the program is distributed. Smart contract codes that operate

[1*] Corresponding Author, [2*] Co-corresponding Author

on ethereum, a blockchain-based framework, can have security vulnerabilities inside the code. Due to the nature of the blockchain, no one can arbitrarily modify the contract when the contract is completed and the block is created, so if you sign a chain code with weak security, it cannot be modified, which creates a security threat. Software security weakness analysis is a process of inspecting the security weaknesses inherent in the developed source code to remove security threats by finding and removing the security weaknesses inherent in the software in advance [6-11].

In this paper, we used deep learning's graph neural network (GNN) to detect security vulnerabilities in solidity codes [12-13]. To analyze security vulnerabilities in solidity code, we defined eight types of security weakness items, converted the solidity code into graph data. In order to represent both the structural elements of the program, the control flow, and the data flow, the solidity code was converted into an abstract syntax tree (AST) and the graph information required for GNN learning was extracted from AST to convert the solidity code into a graph. Next, after generating several datasets for training GNN models by integrating these graph data and their properties with labels, it is possible to detect whether security vulnerabilities exist in the solidity code through GNN learning. This method performs security weakness detection more effectively than conventional rule-based methods.

2. Related Studies

2.1. Blockchain and Smart Contracts

Blockchain is a distributed ledger technology that allows users to record and share information safely and transparently. A smart contract is a contract decided based on a blockchain and is a program that automatically executes or executes contract terms. Smart contracts improve the transparency and reliability of transactions by utilizing the tampering prevention function of blockchain technology [1-5].

Smart contracts security vulnerability analysis is an analysis technique that diagnoses whether the security vulnerability, which is the basis cause of security vulnerability, exists inside the program, and proactively detects and removes potential vulnerabilities such as program defects and errors in advance to proactively eliminate the possibility of security threats such as hacking [6-11]. The smart contract vulnerability analysis method is divided into static analysis through the existing rule-based method and dynamic analysis through flow graph [3-5].

Security weakness analysis for smart contracts is an analysis technique that diagnoses whether the security weakness which is the basis cause of security vulnerability exists inside the program, and it is a method that proactively eliminates the possibility of causing security threats such as hacking by detecting and removing potential vulnerabilities such as defects and errors in program in advance. Security weakness analysis method is divided into static analysis and dynamic analysis. Static analysis is usually done by code review and is performed during the implementation phase of the security development life cycle. Dynamic analysis, unlike static analysis, does not have access to source code, and is a method of finding security weaknesses in a running application program, such as vulnerability scanning and penetration testing [6-11, 15-17].

2.2. Solidity

Solidity was first proposed by Gavin Wood in august 2014 and developed by the solidity team led by Christian Reitwiessner of the ethereum project. Solidity is a smart contract development language provided by ethereum and is used to write or implements smart contracts for various blockchain platforms. It mainly provides data types and functions needed to exchange ether (ETH). This language is a statically typed language, so the types of variables are determined at compile time. Solidity was designed to target the ethereum virtual machine (EVM), a virtual machine shared by nodes of the ethereum blockchain network and the engine that operates the entire ethereum.

Solidity is designed to develop smart contracts that run on the EVM and are compiled into bytecode that can run on the EVM. Through solidity, developers can implement applications by including self-executing business logic in a smart contract. Matters recorded in the smart contract cannot be denied and are performed forcefully. In addition, Ethereum is a platform that allows multiple distributed applications to be used as a new blockchain network [18-21].

Since ethereum supports the complete turing language, it can accommodate various applications implemented using the language mainly used by developers. However, due to the nature of the blockchain, it cannot be arbitrarily modified when the contract of the chain code is completed, so there is a problem that if a chain code with a security weakness is executed on ethereum, it can develop into a security weakness.

2.3. Graph Neural Networks(GNNs)

GNN is a type of artificial neural network for processing data that can be expressed as a graph, and is a powerful deep learning model designed to operate on graph-structured data. Unlike traditional neural networks, which process data in grid-like structures such as images or sequences, GNNs can process complex non-Euclidean structures in graphs. This feature makes GNNs particularly suitable for tasks involving relationships and interactions between entities, such as smart contract analysis [12-14].

Smart contract codes can be naturally expressed as a graph with CFG(control flow graph) and DFG(data flow graph) [22] representing execution and data flow. GNN can effectively analyze and detect security vulnerabilities by converting smart contracts into a graph representation. Through this processing, GNN captures complex relationships and dependencies within code, which are important for identifying security problems in smart contracts, and can efficiently process large graphs, making it suitable for analyzing complex smart contracts. Additionally, GNNs can improve detection of new vulnerabilities by generalizing training data to new, unseen data. This approach can detect a wider range of security vulnerabilities more effectively than traditional methods. Figure 1 shows the GNN model structure.

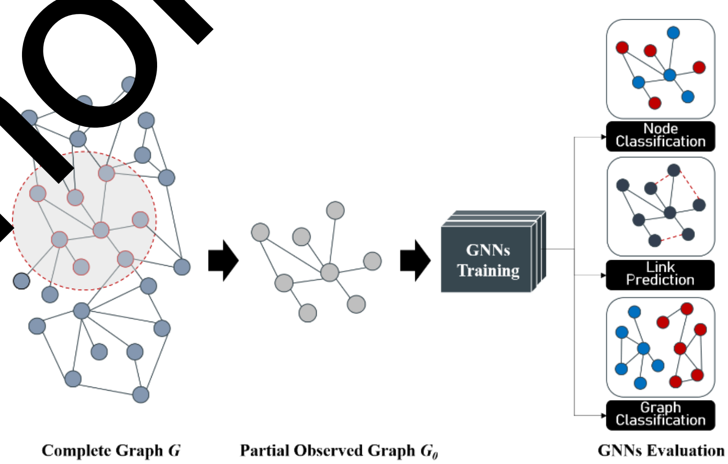

Fig. 1 GNN Model Structure

3. Solidity Smart Contract Security Weakness Analyzer

The solidity smart contract security weakness analyzer diagnoses security weaknesses by converting the source program of solidity, one of the languages that write smart contracts, into a syntax tree. Figure 2 is a structural diagram of the solidity smart contract security weakness analyzer.

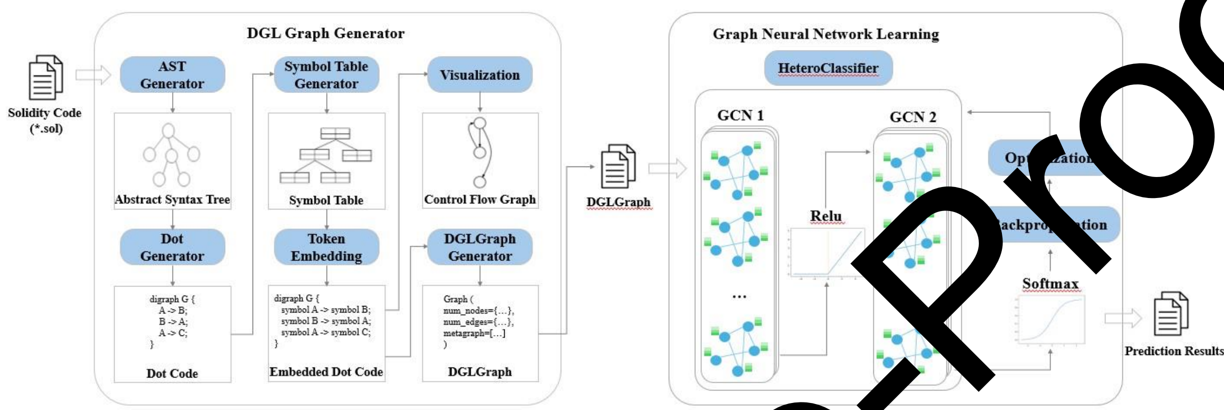

Fig. 2 Structure of the Solidity Smart Contract Security Weakness Analyzer

The solidity smart contract security weakness analyzer consists of a generation unit and a learning unit. The generation unit receives the solidity code as input to generate a Deep Graph Library (DGL) graph, and the learning unit receives the DGL graph as input to perform learning to generate a security weakness analysis model and detect the security weakness of the smart contract.

The DGL graph generation unit consists of an Abstract Syntax Tree(AST) generator that converts the solidity code into an AST, a dot generator that generates a dot code by extracting only necessary information from the generated AST [23], a symbol table generator that generates a symbol information table for token embedding, and a DGL graph generator that converts the Embedded dot code generated by receiving the symbol information table and the Dot code as input into a DGL graph. In this process, an intuitive understanding of the graph generated by visualizing and representing the Embedded Dot code using a visualization tool can be provided.

The learning unit's model for solidity smart contract security weak point detection consists of two layers of the Graph Convolution Network (GCN), which performs graph classification learning using the DGL graph-type dataset generated by the DGL graph generator, and each layer aggregates neighborhood information to calculate a new node representation.

3.1. Defining the Security Weakness of Solidity Code

In order to analyze the security weaknesses of the solidity code, the security weaknesses of the solidity code are first defined. Table 1 is a list of the items of the solidity code security weaknesses proposed in this paper. The causes of security weaknesses were defined into eight items as follows in terms of the reliability of code execution and data processing.

Table 1. Defined Security Weakness Items

Solidity Security Weakness Item List
- unchecked external call
- dangerous delegate call

- timestamp dependency
- Integer overflow
- reentrancy
- block number dependency
- ether strict equality
- ether frozen

3.2 AST(Abstract Syntax Tree) Generator

The AST generator receives the solidity code as input and generates the AST using the parse method of the solidity parser library. Figure 3 shows the example solidity code to be used for the analysis of security weaknesses and the AST generated by the AST generator receiving the solidity code as input.

TestCoin.sol and AST

```
pragma solidity ^0.4.15;

contract TestCoin is EIP20Interface {
...
function transferFrom(address _from, address _to,
                    uint256 _value) public returns (bool success) {
      uint256 allowance = allowed[_from][msg.sender];
      require(balances[_from] >= _value && allowance >= _value);
      balances[_to] += _value;
      balances[_from] -= _value;
      if (allowance < MAX_UINT256) {
         allowed[_from][msg.sender] -= _value;
      }
      emit Transfer(_from, _to, _value);
      return true;
 }
...
```


```
{
   "type":"FunctionDefinition",
   "name":"transferFrom",
   "parameters":{
      "type":"ParameterList",
      "parameters":[
         {
            "type":"Parameter",
            "typeName":{
               "type":"ElementaryTypeName",
               "name":"address"
            },
            "name":"_from",
            "storageLocation":"None",
            "isStateVar":false,
            "isIndexed":false
         },
         {
            "type":"Parameter",
            "typeName":{
               "type":"ElementaryTypeName",
               "name":"address"
            },
            "name":"_to",
            "storageLocation":"None",
            "isStateVar":false,
            "isIndexed":false
         },
         {
            "type":"Parameter",
            "typeName":{
               "type":"ElementaryTypeName",
               "name":"uint256"
```


Fig. 3 Solidity Code and AST

3.3. Dot Generator

The dot generator receives the AST generated by the AST generator as an input to generate the dot code. The dot code is a language used to draw graphs in Graphviz, a visualization tool, and only necessary information was reflected when generating Deep Graph Library(DGL) graphs, and unnecessary information in the AST was removed. Figure 4 is an example of the dot code generated through the dot code generator.

TestCoin.dot

```
pragma solidity ^0.4.15;

digraph G {
node[shape=box, style=rounded, fontname="Sans"]
...
9 [label = Function];
9 -> 10;
10 [label = Block];
10 -> 11;
```

```
11 [label = "Expression
allowance = allowed [ _from ]  [ msg . sender ]
require ( balances [ _from ]  >= _value && allowance >= _value )
balances [ _to ]  += _value
balances [ _from ]  -= _value"];
11 -> 12;
12 [label = "Condition
allowance < MAX_UINT256", shape = diamond];
12 -> 14 [label = "true", fontcolor="blue"];
12 -> 13 [label = "false", fontcolor="red"];
14 [label = Block];
14 -> 15;
15 [label = "Expression
allowed [ _from ]  [ msg . sender ]  -= _value"];
15 -> 13;
13 [label = IfEnd];
13 -> 16;
16 [label = "return
True"];
16 -> 17;
17 [label = FunctionEnd];
...
```

Fig. 4 Dot Code generated by the Dot Code Generator

3.4. Symbol Table Generator

The symbol table generator generates a symbol information table for token embedding of the dot code. The symbol table is generated at the time of execution of the dot code generator, and is used to generate the Embedded dot code by symbolically changing the user-defined function name, variable name, and state variable name. Figure 5 shows the symbol table structure.

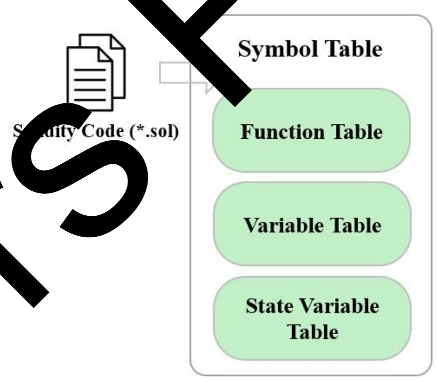

Fig. 5 Symbol Table Structure

3.5. Embedded Dot Code Generation

The embedded dot code is generated based on the dot code and the values stored in the symbol table. The embedded dot code makes the DGL graph symbolic, so that general rules and patterns can be learned without relying on specific data during training. Figure 6 is an example of the embedded dot code generated

Embedded Dot Code

```
pragma solidity ^0.4.15;

digraph G {
node[shape=box, style=rounded, fontname="Sans"]
...
```

```
9 [label = Function];
9 -> 10;
10 [label = Block];
10 -> 11;
11 [label = "Expression
variable4 = state_variable2 [ variable5 ]  [ variable0 . sender ]
require ( state_variable1 [ variable5 ] >= variable2 && variable4 >= variable2 )
state_variable1 [ variable3 ]  += variable2
state_variable1 [ variable5 ]  -= variable2"];
11 -> 12;
12 [label = "Condition
variable4 < state_variable0", shape = diamond];
12 -> 14 [label = "true", fontcolor="blue"];
12 -> 13 [label = "false", fontcolor="red"];
14 [label = Block];
14 -> 15;
15 [label = "Expression
state_variable2 [ variable5 ]  [ variable0 . sender ]  -= variable2"];
15 -> 13;
13 [label = IfEnd];
13 -> 16;
16 [label = "return
True"];
16 -> 17;
17 [label = FunctionEnd];
...
```

Fig. 6 Embedded Dot Code

3.6. Visualization of Embedded Dot Code

Visualization is performed through the Graphviz library to visualize the embedded dot code and facilitates understanding of the structure of the code and data flow. Figure 8 is an example of visualizing the embedded dot code in Figure 7.

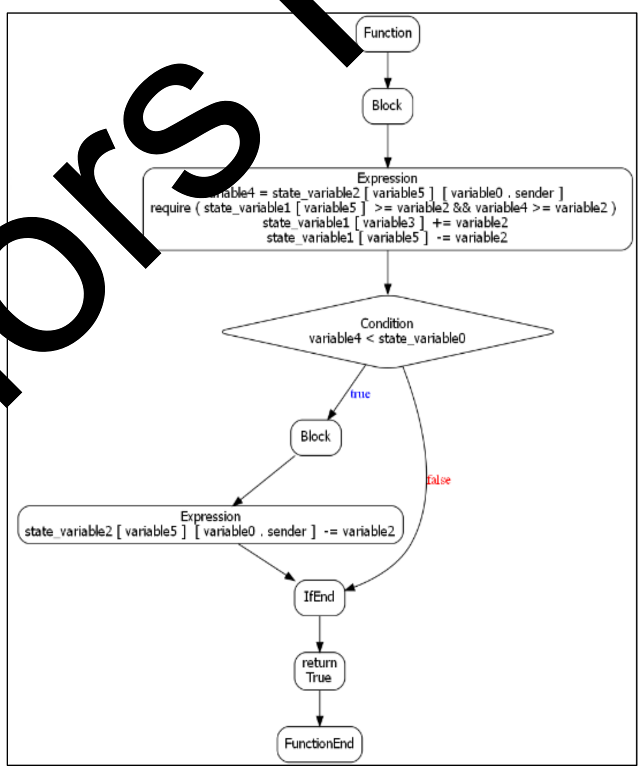

Fig. 7 Visualization of Embedded Dot Code

3.7. DGL(Deep Graph Library) Graph Generator

The DGL graph generator receives a dot code as an input to generate a DGL graph. A DGL graph is a heterogeneous graph containing various types of nodes and edges. There are 13 node types, including '**Block**', '**Return**', '**Break**', '**Expression**', '**Throw**', '**Condition**', '**IfEnd**', '**WhenEnd**', '**LoopVariable**', '**LoopExpression**', '**ForEnd**', '**Function**', and '**FunctionEnd**', and there are three edge types consisting of **Normal**, **True**, and **False**. Figure 8 shows the DGL graph generated by the DGL graph generator by receiving the embedded dot code in Figure 8 as an input.

TestCoin.sol's DGL Graph

```
Graph(num_nodes={'Block': 29, 'Condition': 13, 'Expression': 25, 'ForEnd': 2,
          'Function': 28, 'FunctionEnd': 31, 'IfEnd': 14, 'LoopExpression': 2,
          'LoopVariable': 2, 'WhileEnd': 2, 'break': 2, 'return': 30, 'throw': 2},
    num_edges={('Block', 'normal', 'Block'): 1, ('Block', 'normal', 'Expression'): 5,
          ('Block', 'normal', 'return'): 2, ('Condition', 'false', 'IfEnd'): 1,
          ('Condition', 'true', 'Block'): 1, ('Expression', 'normal', 'Condition'): 1,
          ('Expression', 'normal', 'FunctionEnd'): 1, ('Expression', 'normal', 'IfEnd'): [illegible],
          ('Expression', 'normal', 'return'): 2, ('ForEnd', 'normal', 'ForEnd'): 1,
          ('Function', 'normal', 'Block'): 5, ('IfEnd', 'normal', 'return'): 1,
          ('LoopExpression', 'normal', 'LoopExpression'): 1,
          ('LoopVariable', 'normal', 'LoopVariable'): 1, ('WhileEnd', 'normal', 'WhileEnd'): 1,
          ('break', 'normal', 'break'): 1, ('return', 'normal', 'FunctionEnd'): 5,
          ('throw', 'normal', 'throw'): 1},
    metagraph=[('Block', 'Block', 'normal'), ('Block', 'Expression', 'normal'),
          ('Block', 'return', 'normal'), ('Expression', 'Condition', 'normal'),
          ('Expression', 'FunctionEnd', 'normal'), ('Expression', 'IfEnd', 'normal'),
          ('Expression', 'return', 'normal'), ('Function', 'FunctionEnd', 'normal'),
          ('Condition', 'IfEnd', 'false'), ('Condition', 'Block', 'true'), ('IfEnd', 'return', 'normal'),
          ('ForEnd', 'ForEnd', 'normal'), ('Function', 'Block', 'normal'),
          ('LoopExpression', 'LoopExpression', 'normal'),
          ('LoopVariable', 'LoopVariable', 'normal'), ('WhileEnd', 'WhileEnd', 'normal'),
          ('break', 'break', 'normal'), ('throw', 'throw', 'normal')])...
```

Fig. 8 DGL graph generated by the DGL graph generator

3.8. Graph Neural Network(GNN) Learning

To analyze the security weakness of the solidity code, a heterogeneous graph classification model is learned through GNN learning of deep learning using the data set of the DGL graph generated through the DGL graph generator. Figure 9 shows the structure of the learning part of the solidity smart contract security weakness analyzer

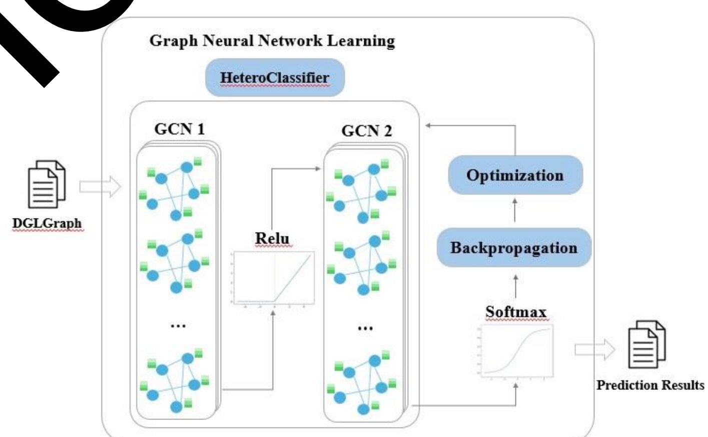

Fig. 9 Structure of Learning Part of the Security Weakness Analyzer

The GNN learning model consists of a Graph Convolution Network (GCN) consisting of two layers. graph classification learning is performed using the DGL graph-type dataset generated by the DGL graph generator, and each convolutional layer updates node features, applies the **ReLu** function in the net propagation process, and predicts class probabilities using the **Softmax** function after the second convolution layer.

During training, the cross entropy loss between the predicted result obtained through the **Softmax** function and the actual label is calculated. The slope is calculated by backpropagating the model through the calculation result. After that, the slope calculated using the **Adam** optimization algorithm is applied to the weight of the model and updated.

The GNN learning model calculates a new node representation by aggregating neighboring information of each node through the above process, and based on this, the existence of security weaknesses in the input graph is represented by **Prediction** and **Active Labels**. Figure 10 shows the prediction result example of the model for the input graph.

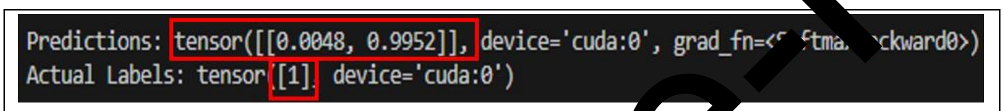

Fig. 10 Prediction Result Example of Security Weakness of GNN Learning Model

Predictions is a result of predicting the existence of a security weakness in the code after the model who has completed training receives the solidity code, and the value of index 0 indicates that there will be no security weakness, and the value of index 1 indicates that there will be a vulnerability to that security weakness. Since the model predicts probabilistically, if the value of index 0 is larger, it is predicted to be higher that there is no security weakness, and if the value of index 1 is larger, it is predicted that there is a high probability that there will be a security weakness.

Actual Labels indicates whether there is a security weakness in the corresponding code, and if it is 0, it indicates a code without a security weakness, and if it is 1, it indicates a code with a security weakness. Therefore, there are security weaknesses in the program used as an example, and a security weakness analyzer through graph-based deep learning (GAN) detects the security weaknesses present in the smart contract program

There are 8 models for each security weakness, and Figure 11 shows the prediction results example of the model obtained by inputting DGL graph into 8 models learned according to each security weakness. If the value of index 0 of **Prediction** is larger, undetected is output, and if the value of index 1 is larger, detected is output.

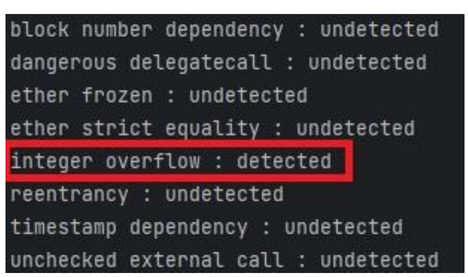

Fig. 11 Security Weakness Prediction Results Example for 8 Models

4. Experimental Results and Analysis

In order to detect the security weakness of the smart contract written with the solidity code on the ethereum platform where the solidity smart contract runs, an experiment was conducted to detect the security weaknesses by analyzing various vulnerability patterns of the solidity code. Figure 12 shows the results of detecting security weaknesses for the **integer overflow** in the solidity code used in [illegible] experiment.

IntegerOverflow.dot

```
pragma solidity ^0.4.15;

contract TestCoin is EIP20Interface {
    uint256 constant private MAX_UINT256 = 2**256 - 1;
    mapping (address => uint256) public balances;
    mapping (address => mapping (address => uint256)) public allowed;
    string public name;
    uint8 public decimals;
    string public symbol;

    function TestCoin( ) public {
        balances[msg.sender] = 10*10**26;
        totalSupply = 10*10**26;
        name = "LHJT";
        decimals = 18;
        symbol = "LHJT";
    }

    function transfer(address _to, uint256 _val[illegible] retu[illegible](bool success) {
        require(balances[msg.sender]>= _valu[illegible]
        balances[msg.sender] -= _value;
        balances[_to] += _value;
        emit Transfer(msg.sender, _to, _value)[illegible]
        return true;
    }

    function transferFrom(address [illegible]from, address _to[illegible]uint256 _value) public returns (bool success) {
        uint256 allowance = allo[illegible]rom][msg.sender];
        require(balances[_from[illegible]= _valu[illegible]& allowance >= _value);
        balances[_to] += _valu[illegible]
        balances[_from] -[illegible]value;
        if (allowance <[illegible]X_UINT256) [illegible]lowed[_from][msg.sender] -= _value; }
        emit Transfer(_fr[illegible]_to, _value);
        return true;
    }
}
```

```
[illegible]ediction[illegible] tensor([[1.6042e-06, 1.0000e+00]] grad_fn=<SoftmaxBackward0>)
Ac[illegible]els: tensor([1])
```

```
block number dependency : undetected
dangerous delegatecall : undetected
ether frozen : undetected
ether strict equality : undetected
integer overflow : detected
reentrancy : undetected
timestamp dependency : undetected
unchecked external call : undetected
```

Fig. 12 Integer Overflow Detection Result of Solidity Code

Security weakness detection for the security weakness detection item, **Integer overflow**, was performed with IntegerOverflow.sol, which has a security weakness. In the solidity code of Figure 12, **balances[_to] +=_value**; the part of transmitting tokens to the other party's account is written in the transfer, transferFrom function. In this case, be careful of exposure to security vulnerabilities for **integer overflow** because no exception is handled to **integer overflow** using SafeMath. The detection results of security weaknesses warn that among the eight security weaknesses, there are security weaknesses for **integer overflow**.

Figure 13 shows the results of detecting security weaknesses for the **timestamp dependency** in the solidity code used in the experiment.

TimestampDependency.sol

```
pragma solidity ^0.4.15;
contract Freezable_Token is StandardToken {
    function releaseOnce() public {
        bytes32 headKey = toKey(msg.sender, 0);
        uint64 head = chains[headKey];
        require(head != 0);
        require(uint64(block.timestamp) > head);
        bytes32 currentKey = toKey(msg.sender, head);
        uint64 next = chains[currentKey];
        uint amount = freezings[currentKey];
        delete freezings[currentKey];
        balances[msg.sender] = balances[msg.sender].add(amount);
        freezingBalance[msg.sender] = freezingBalance[msg.sender].sub(amount);
        if (next == 0) { delete chains[headKey]; }
        else {
            chains[headKey] = next;
            delete chains[currentKey];
        }
        emit Released(msg.sender, amount);
    }

    function releaseAll() public returns (uint tokens) {
        uint release;
        uint balance;
        (release, balance) = getFreezing(msg.sender, 0);
        while (release != 0 && block.timestamp > release) {
            releaseOnce();
            tokens += balance;
            (release, balance) = getFreezing(msg.sender, 0);
        }
    }
}
```

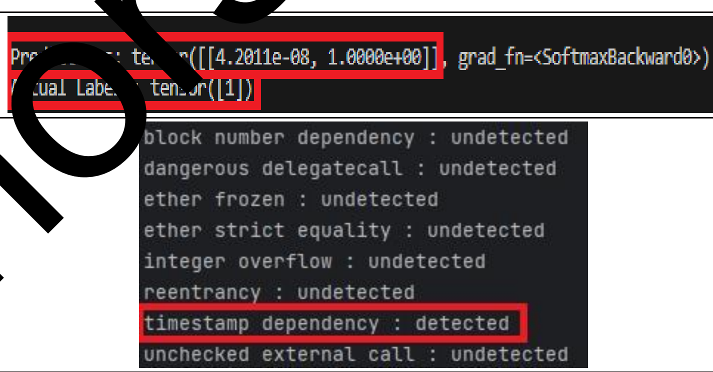

Fig. 13 Timestamp Dependency Detection Result of Solidity Code

The security weakness detection for the timestamp dependency, a security weakness detection item, was performed with TimestampDependency.sol that has a timestamp dependency security weakness. In the solidity code of Figure 13, the **releaseOnce** and **releaseAll** functions use **block.timestamp** to check specific conditions. However, **block.timestamp** can be manipulated by miners within a certain range to intentionally advance or delay the execution point of a specific event, which can lead to unexpected behavior of the system. The detection results of security weaknesses warn that among the eight security weaknesses, there are security weaknesses for timestamp dependency.

5. Conclusions and Further Researches

Blockchain is a distributed ledger technology that allows users to record and share information safely and transparently. A smart contract is a contract decided based on a blockchain and is a program that automatically executes or executes contract terms. Smart contracts improve the transparency and reliability of transactions by utilizing the tampering prevention function of blockchain technology. Software security vulnerability refers to the fundamental cause of vulnerabilities caused by logical errors, bugs, and mistakes that can be defective in software development. To prevent software security accidents, security weaknesses must be analyzed before the program is distributed. Smart contract codes that operate on ethereum, a blockchain-based framework, can have security vulnerabilities inside the code. When the contract is completed and the block is created, the chaincode cannot be arbitrarily modified, so the security weakness must be analyzed before execution. In addition, most security vulnerability analysis methods for smart contracts are currently specialized in detecting specific vulnerabilities using rule-based methods, which is prone to many false positives when detecting security vulnerabilities.

In order to solve this problem, this paper studied an analysis method through the deep learning's graph neural network (GNN) to detect security vulnerabilities in solidity codes. To analyze security vulnerabilities in solidity code, we defined eight types of security weakness items (unchecked external call, dangerous delegate call, timestamp dependency, Integer overflow, reentrancy, block number dependency, ether strict equality, and ether frozen), converted the solidity code into graph data. In order to represent both the structural elements of the program, the control flow, and the data flow, the solidity code was converted into an abstract syntax tree (AST) and the graph information required for GNN learning was extracted from AST to convert the solidity code into a graph. Next, after generating several datasets for training GNN models by integrating these graph data and their properties with labels, it is possible to detect whether security vulnerabilities exist in the solidity code through GNN learning. This proposed method performs security weakness detection more effectively than conventional rule-based methods.

In the future, it is believed that more data should be collected and learning about these additional vulnerabilities should be performed in order to allow the proposed system to detect a wider range of security vulnerabilities. In addition, it is expected that higher performance can be achieved by training the model using more advanced and specialized GNN models tailored to the dataset.

ACKNOWLEDGEMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) (No.2022R1F1A1063340))

References

[1] I.C. Lin, T.C. Liao, "A Survey of Blockchain Security Issues and Challenges", International Journal of Network Security, Vol. 19, No. 5, pp. 653–659, 2017.
[2] Zheng, Z.; Xie, S.; Dai, H. N.; Chen, X.; Wang, H. Blockchain Challenges and Opportunities: A Survey., International Journal of Web and Grid Services, 14(4), pp. 352-375, 2018.
[3] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F. Y. Wang, "An Overview of Smart Contract: Architecture, Applications, and Future Trends," IEEE Intell. Veh. Symp. Proc., Vol. 2018-June, pp. 108–113, Oct. 2018.

[4] S. Y. Lin, L. Zhang, J. Li, L. li Ji, and Y. Sun, “A Survey of Application Research based on Blockchain Smart Contract,” Wirel. Networks, Vol. 28, No. 2, pp. 635–690, 2022.

[5] S. N. Khan, F. Loukil, C. Ghedira-Guegan, E. Benkhelifa, and A. Bani-Hani, “Blockchain Smart Contracts: Applications, Challenges, and Future Trends,” Peer-to-Peer Network Applications, Vol. 14, No. 5, pp. 2901–2925, 2021.

[6] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H. N. Lee, “Systematic Review of Security Vulnerabilities Ethereum Blockchain Smart Contract,” IEEE Access, vol. 10, pp. 6605–6621, 2022.

[7] Fagan, Michael E “Design and Code Inspections to Reduce Errors in Program Development”, IBM Sy ems Journal, Vol. 38, No. 2.3, pp. 258-287, 1999.

[8] Y. Son, Y. Lee, S. Oh, “A Software Weakness Analysis Methods for the Secured Software , The sian International Journal of Life Sciences, Vol. 12, pp. 423-434, 2015.

[9] Y. Son, Y. Lee, "A Smart Contract Weakness and Security Hole Analyzer Using Virtual Mach e ba d Dyn ic Monitor," Journal of Logistics, Informatics and Service Science, Success Culture Pr Vo No. 1, p 6-52, Mar 2022.

[10] Y. Lee, "A Study on Intermediate Code Generation for Security Weak ss Anal is of art Contract Chaincode," Journal of Logistics, Informatics and Service Science, Success ture P ss, Vol. 9, No. 1, pp. 53-67, Jan 2022.

[11] S. Kim, Y. Son, Y. Lee, "A Study on Chaincode Security Weakness Detector in Hy rledger Fabric Blockchain Framework for IT Development," Journal of Green Engineering, Alpha Publis rs, . 10, No. 10, pp. 7820-7844, Oct 2020.

[12] Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; Monfardin G. T Graph Neural Network Model, IEEE Transactions on Neural Network, 20(1), pp. 61-80, 2008.

[13] L. Wu, P. Cui, J. Pei, L. Zhao, Graph Neural Networks : unda ons, Fr tiers, and Applications, Springer, 2022.

[14] D. Zheng, M. Wang, Q. Gan, Z. Zhang, and G aryp Auth s, Learning Graph Neural Networks with Deep Graph Library, WWW '20: Companion Proc ings of e Web nference 2020, ACM, pp. 305 - 306, 2020.

[15] S. Kim, R. Y. C. Kim, and Y. B. Park, “Soft re V erability Detection Methodology Combined with Static and Dynamic Analysis,” Wirel. Pers. Commun. 893, Vol. 89, No. 3, pp. 777–793, Dec. 2015.

[16] B. Chess, G. McGraw, "Static Analysis for Security IEEE Security & Privacy, 2(6), (**2004**), pp.76-79.

[17] A. Petukhov, et al., "Detecting Security Vulnerabili s in Web Applications Using Dynamic Analysis with Penetration Testing." online Proceedi of the Application Security Conference, (**2008**).

[18] Solidity Documentation, Ethereum 2022

[19] Solidity Documentation. https: readt edocs.io/en/v0.4.21/contracts.html

[20] S. Peyrott, An Introduction t thereum and mart Contracts, Auth0, 2017.

[21] https://www.ethereum.org/

[22] Deep Graph Library (https www.dgl.ai/

[23] Y. Son, J. Jung, Y. L , "Desig and mplementation of the Secure Compiler and Virtual Machine for Developing Secure IoT Services, Future G eration Computer Systems, Vol. 76, pp. 350-357, 2014.