Proposing of Imaging Graph Neural Network with Defined Security Pattern for Improving Smart Contract Vulnerability Detection

Pham Trong Linh[1], Ta Minh Thanh[2]
[1,2]Le Quy Don Technical University, Ha Noi, Vietnam
Correspondence: Pham Trong Linh (tronglinhmta0611@gmail.com)
Communication: received 13 Feb 2023, revised 12 April 2023, accepted 15 May 2023
Digital Object Identifier: 10.32913/mic-ict-research.v2023.n2.1198

Abstract: A smart contract is a special set of protocols based on blockchain technology to implement the terms or agreements between the parties in the contract. Lots of smart contracts are built and deployed everyday. However, to ensure the safety of smart contracts is still a big challenge. Smart contracts are built to carry out transactions directly related to cryptocurrencies, therefore, the loss of security of smart contracts leads to huge financial losses. Common methods being used to check and to verify smart contract security are heavily dependent on hard rules defined by experts, leading to low detection accuracy and non-scalable, which can be bypassed by experienced attackers. In this paper, we propose to use the combination of Imaging Graph Neural Network With Defined Pattern to detect vulnerabilities in smart contracts. We construct a contract graph that shows the relationship between the main components in a smart contract. Then we extract graph features from normalized graphs, and combine graph features with defined security patterns to create combined features. Finally, we implemented normalization to gray scale image and feed it to the Convolutional Neural Network (CNN) to learn for vulnerability detection. Results show significantly improved accuracy compared to previous methods or other models. Specifically, 96,42%, 90,12%, 79% for reentrancy, timestamp dependence and infinite loop.

Keywords: *deep learning, stock price prediction, LSTM, BiLSTM, CNN.*

I. Introduction

1. Overview

Smart contracts (SC) are known as the programs that run on the blockchain [1]. It likes a digital contract that is bounded by a significant set of rules. Such rules are pre-defined by a set of computer code which must be copied and enforced in all nodes in the network. One of the most prominent platforms for SC definition and execution is Ethereum[1]. Ethereum is a blockchain-based decentralized network platform for deploying the smart contracts using the Turing-complete language. A smart contract when deployed to the blockchain is immutable, so if the vulnerabilities are not carefully checked before being deployed to the system, its consequences are extremely serious. One of the more famous recent attacks is the 3^{rd} of august 2022 event, thousands of Solana wallets were drained of nearly $8 million in total. The hackers' exploit was thought to have occurred due to complications in importing accounts (Solana wallets). Therefore, smart contracts (the program deployed on blockchain) need to be carefully checked for security before deploying to the blockchain network.

Common methods being used to detect smart contract vulnerabilities are Symbolic Execution Technologies [2], classical static analysis [3]. These methods have the weakness of being heavily dependent on predefined rules, leading to low vulnerability detection performance and being easily bypassed by experienced attackers. In addition, these methods consume a lot of time and computation resources.

Recently, studies on the application of deep learning neural networks to the detection of smart contract security vulnerabilities have yielded highly accurate results. In this paper, we propose a fully automated and scalable approach that can detect vulnerabilities at the function level. Specifically, we build a contract graph that shows the relationships between the components of a smart contract program in chronological order. Afterwards, we extract graph features from normalized graphs. Then, we extract the security pattern features from smart contracts code using the defined security patterns, then combine graph features with security pattern features to extract the special combined features

[1]https://ethereum.org/

for each pattern. Finally, we implement normalization to grayscale image and feed it to the Convolutional Neural Network (CNN) to learn for vulnerability detection.

2. Our contributions

In proposed approach, we focus on discovering the most suitable combined features for each pattern via graph neural network, then employ the imaging features to improve the efficiency of smart contract vulnerability detection. Our contributions can be listed as follows.

(1) Propose a extraction method that can generate the contract graph from source code of smart contract components of a smart contract in chronological order.

(2) Find out the appropriate algorithm to transform the combined features to gray scale image then feed such images to CNN. Our method can visualize the deference of each vulnerability of smart contract for classification.

(3) We focus on testing the smart contract code built in Solidity language running on the Ethereum platform. The final results show that the detection method has an accuracy of 96.64%, 90,12% ,79% for reentrancy, timestamp dependence,infinite loop. These results show that applying our model to the detection of security vulnerabilities yields outstanding results than state-of-the-art methods.

3. Roadmap

The rest of our paper is organized as follows. Sect. II describes the related works and the background of smart contracts. Then, Sect. III introduces the details of our proposed method. We extend the previous algorithm to propose the imaging graph neural network (IGNN) with defined security pattern for improving smart contract vulnerability detection. After that, Sect. IV gives the experimental results and discussion. Finally, Sect. V concludes this paper.

II. Related Works

1. Some previous works

The security of smart contracts has always been of special concern, to detect smart contract vulnerabilities can use many ways. Symbolic execution is one of the common techniques used to detect smart contract vulnerabilities. For example, Oyente [4] is the first symbolic execution tool for smart contracts. It supports detection for transaction order dependency (TOD), timestamp dependency, reentrancy, mishandled exception and integer overflow. Oyente [4] relies on simplified semantics and use templates to define specific properties. However, the tool lacks a semantic characterization. In addition, it reduces path explosion by limiting the number of loops, therefore it is difficult to detect variuos security defects. Securify [5] uses formal verification methods to detect the vulnerabilities of smart contracts, which uses certain characteristics to analyze

```
pragma solidity ^0.8.10;

contract Reentrancy {

    mapping (address => uint) private
        balances;

    function withdrawBalance() public {
        uint amountToWithdraw =
            balances[msg.sender];
        require(msg.sender.call.value(
            amountToWithdraw)());
        balances[msg.sender] = 0;
    }
}
```

Figure 1. Reentrancy vulnerability

whether the smart contract contains vulnerabilities or not. Securify generates a dependency graph by analyzing and displaying the bytecode of the smart contract. According to certain characteristics, it analyzes whether the semantic information of the contract satisfies or violates those characteristics, and evaluates whether there are security holes (loopholes) in the smart contract or not. It's input is the bytecode of smart contract and a bunch of templates. They are described in a domain-specific language for security flaws. The output is the location of specific security flaws and vulnerabilities. Such security patterns can be written by the user, then Security can be extensible.

On the other hand, some works tried to apply deep learning to smart contract vulnerability detection, such as using graph neural networks (GNNs) [6] for smart contracts code vulnerability detection which constructs a contract graph to represent both syntactic and semantic structures of a smart contract code, then use the pre-designed normalized model to highlight the major nodes. Finally, such method used a degree-free graph convolutional neural network (DR-GCN) and a novel temporal message propagation network (TMP) to learn security patterns from the normalized graphs for vulnerability detection. Different from these methods, in this work, we mainly focused on using Combination Imaging Graph Neural Network with Defined Security Pattern to detect Smart Contract Vulnerability.

2. Background of smart contract

a) Blockchain and Smart Contract

A blockchain is a ledger (as a distributed database) that is shared among the nodes of a computer network. The blockchain network stores information electronically in a

```
pragma solidity ^0.8.10;

contract TimestampDependence {
    uint constant TICKET_AMOUNT = 5;
    uint public pot;

    function play() payable {
        assert(msg.value ==
            TICKET_AMOUNT);
        pot += msg.value;
        var random = uint(sha3(block.
            timestamp)) % 3;
        if (random == 0){
            msg.sender.transfer(pot);
            pot = 0;
        }
    }
}
```

Figure 2. Timestamp dependence vulnerability

digital form[2], to maintain a secure and decentralized record of transactions. The innovation with blockchain is that it ensures the integrity and security of data records and creates trust without the need for a trusted third party.

Smart contracts (SC) are programs that are deployed on blockchain network. It works by following simple "if/when... then..." statements. That means the network of nodes execute the pre-determined actions when the pre-determined conditions have been met and verified beforehand. These actions may include disbursing funds to the appropriate parties, registering vehicles, sending notices or issuing tickets, etc. The blockchain is then updated when the transaction is complete. Note that, the transaction cannot be changed anymore, then only authorized parties can see the result. In a smart contract, there may be many provisions necessary to satisfy the participants that the task is satisfactorily completed. To establish such conditions, participants must pre-define how their transactions and data are represented on the blockchain, agreeing to "if/when...then.." rules that governs those transactions, uncovering all possible exceptions.

Smart contracts on Ethereum[3] are typically written in a high-level language, then compiled in EVM bytecode. The most widely used language is Solidity[4]. Solidity is a simple contract-oriented high-level programming language. Its syntax is similar to Javascript.

[2]https://en.wikipedia.org/wiki/Bitcoin
[3]https://ethereum.org/
[4]https://docs.soliditylang.org/

```
pragma solidity ^0.8.10;

contract InfiniteLoop {
    //...
    //...
    function () payable {
        count ++;

        while (count < 20) {
            withdraw();
        }
    }
}
```

Figure 3. Infinite loop vulnerability

b) Smart Contract Vulnerabilities

(i) Reentrancy

The Reentrancy vulnerability is caused by attackers withdrawing funds from the target by recursively calling the target withdrawal function. Attackers execute withdrawals several times before checking the balance. Whenever the attacker receives ETH, his/her contract automatically calls the fallback function, which calls the withdrawal function many times until become zero. At this point the attack enters a recursive loop, therefore, the contract cannot update the attacker's balance.

Figure. 1 shows an example of a re-entry vulnerability. The value of *call.value* (Line 9) implements the fallback function of the sender. It sends money, as much as the user's balance. If the fallback function in the sender's code calls the *drawBalance* function again, then a recurring deposit call loop will be generated multiple times before updating the balance status on Line 10. Thus, the smart contract of the victim will send unlimited money to the recipient. Reentrancy is a notorious vulnerability that has caused massive economic damage in the mining of DAOs (decentralized autonomous organizations). Following a DAO crash, safety development guidelines recommend completing state updates before performing any actions that may call other code.

(ii) Timestamp dependence

This timestamp dependency vulnerability in a smart contract occurs when it relies on the value of the block timestamp to perform an operation. The block timestamp value is generated by the node executing the smart contract, making it vulnerable to manipulation and attack.

The smart contract shown in Figure. 2 represents a game in which the user has to deposit an amount for the smart contract function *play()*. That amount must be equal to

the value of the TICKET_AMOUNT parameter. The smart contract then retrieves the actual time when the contract was executing, applies a formula to it, and stores the value in a random variable (see Line 10). The smart contract then checks if the value of the random variable is equal to "zero". If it is the user who submitted the transaction, it will become the winner. In order to achieve real-time, the contract smart use *v*ariable block.timestamp The value of the variable *block.timestamp* is provided by the node, so it can easily manipulate it by incrementing it for until receiving the result of Line 10 is "zero" which leads to a vulnerability in the smart contract.

(iii) Infinite loop

The computing power in the Ethereum blockchain is not free. Therefore, we need to pay Ether (ETH) to buy Gas[5] to get the amount of computing power to execute the transaction. Therefore, if you can reduce the number of calculation steps, you can save time and even save a lot of money. Adding loops to smart contracts is one way to increase gas costs. For example, an array already has a lot of loops. However, if the factors increase, then more integration is required to complete that particular loop. If an attacker can introduce infinite loops into the smart contract's process, then he or she can mine all the ETH Gas available in the user's wallet by himself.

Infinite loop vulnerability in smart contract is considered as a loop bug which unintentionally iterates looping. It makes to fail to jump out of the loop, then return an expected results. That means the attacker can influence the elements of the array length, which create a DOS (Denial-of-Service) issue and won't allow the system to jump out of the loop (as shown in Fig. 3). In additionally, it ensures that the contract is stalled as every contract comes with a Gas limit.

III. Our Proposed Method

Method overview. The overall architecture of our proposed method (shown in Fig. 4) consists of five phases: (1) graph contract generation and normalization phase, which create a contract graph to represent both syntactic and semantic structures of a smart contract function and normalize it, then (2) extract graph features from normalized graphs; (3) defined security pattern extraction phase, which extract security patterns predefined by experts from source code; (4) combination and grayscale image normalization phase, which combine graph features with security pattern features to create combined features. Finally, (5) vulnerability detection phase, which feeds grayscale normalized image to the Convolutional Neural Network (CNN) to learn for vulnerability detection.

[5]https://etherscan.io/gastracker

1. Smart Contract Graph Generation and Normalization

a) Smart Contract Graph Generation

The paper [7] shows that the smart contracts can be transformed into symbolic graph representations, which preserves the connection between the components in the program. Based on this idea, we build contract graphs from source code of smart contract by assigning it's components with different roles and calling those components nodes. To identify each link between the components in the program, we construct the edges in chronological order. Then, we determine the importance of the buttons and remove those that are not important.

We build three types of nodes representing 3 different important components in the program: *Main nodes*, *extra nodes* and *fallback nodes*. Such can be explained as follows:

Main nodes. The main node represents the key invitation and variables, which are the main factors that help determine if the program contains vulnerabilities. In particular, for reentrancy vulnerability, the main components used for vulnerability detection are the calls to *call.value*, the variables that represent the *balance*. In case of timestamp dependence vulnerability, the invocations to *block.timestamp*, and variables assigned by *block.timestamp*, and variables assigned by operations that use *block.timestamp*. In case of infinite loop of vulnerability, loop statements such as *for* and *while* can be failed, the loop condition variables, and recursive invocation. These components are considered to be the most important components for vulnerability detection.

Extra nodes. The extra nodes have the role of supporting vulnerability detection, it is combined with the main nodes to increase the accuracy of vulnerability detection. Specifically, reentrancy vulnerability, variables indirectly related to *user balance* or *bonus flag*. In case of timestamp dependence vulnerability, it is clear that invocations do not call *block.timestamp* and variables indirectly related to *block.timestamp*. These invocations and variables are defined as extra nodes E1, E2,..., En.

Fallback nodes. Fallback node: The fallback node F represents elements that directly interact with main nodes and extra nodes.

Edges. In the contract graph, we defined that the graph nodes are closely related to each other will be in chronological order. Therefore, we define edges to create a connection between the component nodes in the contract graph. Each edge represents the association between two component nodes in chronological order. Note that, the properties of graph edge are expressed as a tuple (Vs, Ve, o, t), where Vs and Ve represent its starting and ending of nodes; o represents its temporal order, and t the edge type. We

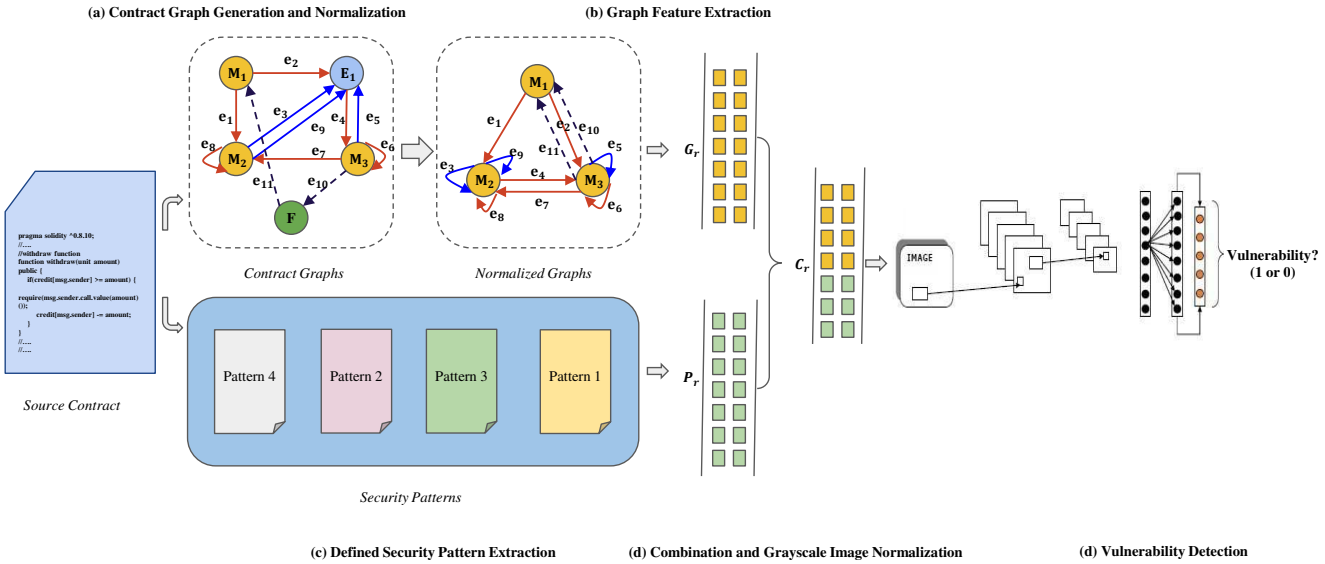


Figure 4. The overall architecture of our proposed method. (a) The contract graph generation and normalization phase; (b) the graph features extraction phase; (c) the defined security pattern extraction phase; (d) the combination and grayscale image normalization phase; (e) the vulnerability detection phase.

construct three categories of edges which are *control flow*, *data flow*, and *fallback edges*, respectively. The defined edges are shown in Table I.

b) Contract Graph Normalization

The nodes in the contract graph used to train the neural network have different roles. Moreover, each smart contract source code generates separate contract graphs, which makes it very difficult to train GNN. Therefore, we created a normalization process for the graph to remove the unimportant factors that interfere with the training process.

Nodes elimination. As explained in Sect. I), a contract graph consists of three components: main nodes $\{M_i\}_{i=1}^{|M|}$, extra nodes $\{E_i\}_{i=1}^{|E|}$ and fallback node F. We normalize the contract graph by removing the nodes that are less important in the graph, extra nodes E_i and fallback node F, and then move all the features to the nearest primary node. If extra nodes or fallback node have more than 2 neighbors, it will transfer all features to the nearest neighbor main node ensuring that the edges of the deleted node remain the same dimension and alignment order over time.

Feature of main nodes. After the extra nodes are removed from the graph, its properties are completely transferred to the neighboring main nodes, so that after normalization, the main nodes will update their features, new features of main nodes include three parts: (i) self-feature, called the feature of main-node M_i itself; (ii) in-features, called features of the extra-nodes $\{P_i\}_{j=1}^{|P|}$ that are merged to M_i and having a path pointing P_j to M_i; and (iii) out-feature, called features of the extra-nodes $\{Q_k\}_{k=1}^{|Q|}$ that are combined to M_i and have a path directed from Q_k from M_i. Fig. 5(e) illustrates the normalized graph of Fig. 5(d).

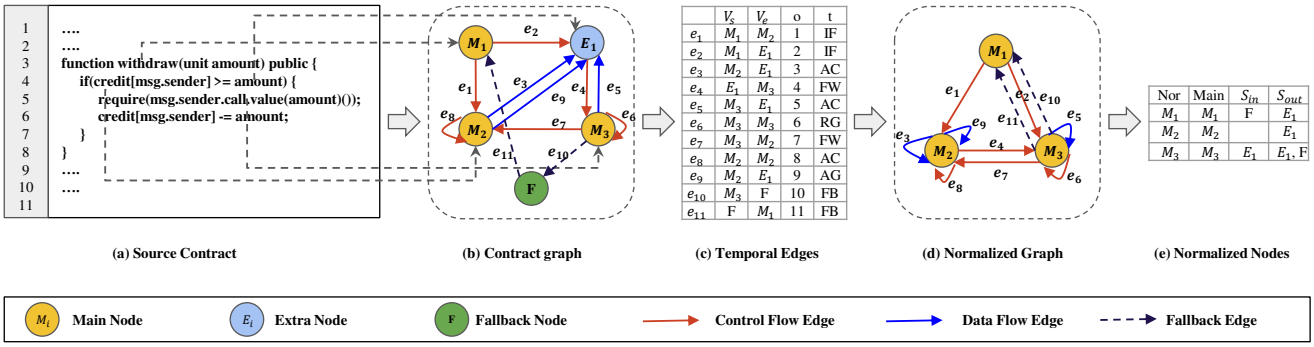
TABLE I
SEMANTIC EDGES SUMMARIZATION. ALL EDGES ARE CLASSIFIED INTO THREE CATEGORIES, NAMELY CONTROL-FLOW, DATA-FLOW, AND FALLBACK EDGES.

Symbol	Semantic Fact	Category
AH	assert{X}	Control-flow
RG	require{X}	
IR	if{...} revert	
IT	if{...} throw	
IF	if{X}	
GB	if{...} else {X}	
GN	if{...} then {X}	
WH	while{X} do{...}	
FR	for{X} do{...}	
FW	natural sequential relationships	
AG	assign{X}	Data-flow
AC	access{X}	
FB	interactions with fallback function	Fallback

2. Graph Feature Extraction

We use a temporal message propagation network. The features extraction process of the graph is divided into two stages, called message propagation phase, readout phase, respectively. Firstly, the network transmits information along its edges in chronological order. Then, it use the *readout* function to create the graph feature G_r, which contains final states of all nodes in the smart contract graph.

Message propagation phase. Let $G = \{V, E\}$ be the normalized contract graph, where V includes of the main



	V_s	V_e	o	t
e_1	M_1	M_2	1	IF
e_2	M_1	E_1	2	IF
e_3	M_2	E_1	3	AC
e_4	E_1	M_3	4	FW
e_5	M_3	E_1	5	AC
e_6	M_3	M_3	6	RG
e_7	M_3	M_2	7	FW
e_8	M_2	M_2	8	AC
e_9	M_2	E_1	9	AG
e_{10}	M_3	F	10	FB
e_{11}	F	M_1	11	FB

Nor	Main	S_{in}	S_{out}
M_1	M_1	F	E_1
M_2	M_2		E_1
M_3	M_3	E_1	E_1, F

Figure 5. The contract graph construction and normalization phase. (a) shows the source code of a smart contract; (b) visualizes the contract graph extracted from the code. Nodes M_i denote main nodes, nodes E_i denote extra nodes, and node F denotes fallback node; (c) illustrates the temporal edges in the extracted graph, where the types of edges are detailed in Table 1; (d) demonstrates the graph after normalization; (e) illustrates nodes in the normalized graph.

nodes, and E includes of all edges of that graph. Note that $E = \{e_1, e_2, \ldots, e_N\}$, where e_k is the k^{th} temporal edge of graph. The messages are propagated along the edges of the graph, one edge per time step. Firstly, initialize the hidden state h_i^0 for each node V_i of the graph. Hidden state is initialized for each node with its features. Afterwards, the message flows are through the k^{th} temporal edge e_k and updates the hidden state h_{ek} of the end node of e_k at the step k. In detail, the message m_k is firstly computed basing on the hidden state h_{sk} of the start node of e_k, and the edge type t_k. The calculation of each can be shown as follows:

$$x_k = h_{sk} \oplus t_k$$
$$m_k = W_k x_k + b_k,$$

where $\oplus$ is concatenation, W_k and b_k are network parameters. The original message x_k contains the information from the start node of e_k and edge e_k itself, which are continuously transformed into a vector embedding using matrix W_k and bias b_k.

The end node of e_k updates its hidden state after receiving the message. That means h_{ek} is updated as follows:

$$\hat{h} = tanh(Um_k + Xh_{ek} + b1),$$
$$h'_{ek} = softmax(R\hat{h}_{ek} + b2),$$

where the matrices U, X, R are network parameters, while $b1$ and $b2$ are bias vectors.

Readout phase. We proceed to extract features after traversing all edges in G by reading out the final states of all nodes. Suppose h_i^T is the final hidden state of the i^{th} node. In this case, the differences between the final hidden state h_i^T and the original hidden state h_i^0 are informative in the vulnerability detection problem. Therefore, the graph feature G_r is generated as follows:

$$s_i = h_i^T \oplus h_i^0$$
$$g_i = softmax(W_g^{(2)}(\tanh(b_g^{(1)} + W_g^{(1)} s_i)) + b_g^{(2)})$$
$$o_i = softmax(W_g^{(2)}(\tanh(b_o^{(1)} + W_o^{(1)} s_i) + b_o^{(2)})$$
$$G_r = FC(\textstyle\sum_{i=1}^{|V|} o_i \odot g_i),$$

where $\oplus$ denotes concatenation, $\odot$ denotes elementwise product. W_j , $b_j^{(1)}$, and $b_j^{(2)}$, $j \in \{g, o\}$ are network parameters.

3. Defined Security Pattern Extraction

For each vulnerability we design sub-patterns to represent the features used to detect the vulnerability. We use one-hot encoding to construct one-hot vectors representing each sub-pattern, then Check the occurrence of the feature in the source code and append 1/0 to the one-hot vector representing whether the sub-pattern is present in the source code or not. The vectors of sub-patterns related to a specific vulnerability are concatenated into a vector P_r, which are the pattern features. For example, reentrancy vulnerability have three sub-pattern are call.value invocation pattern, balance deduction pattern, enough balance pattern and three one-hot vectors corresponding to these three patterns are [1, 0, 0], [0, 1, 0], [0, 0, 1]. When the features corresponding to the sub-patterns of the reentrancy vulnerability are detected in the source code, append 1/0 to the end of the corresponding one-hot vectors. For example, when a call.value invocation is detected, it will append 1 to the end of the one-hot vector of the call.value invocation pattern to become a vector [1, 0, 0, 1].

4. Combination and Grayscale Image Normalization

Combining graph features G_r and pattern features P_r. After extracting graph features G_r and pattern features P_r. We construct a stage that combines G_r and P_r together to create a vector C_r. Specifically, we keep the features

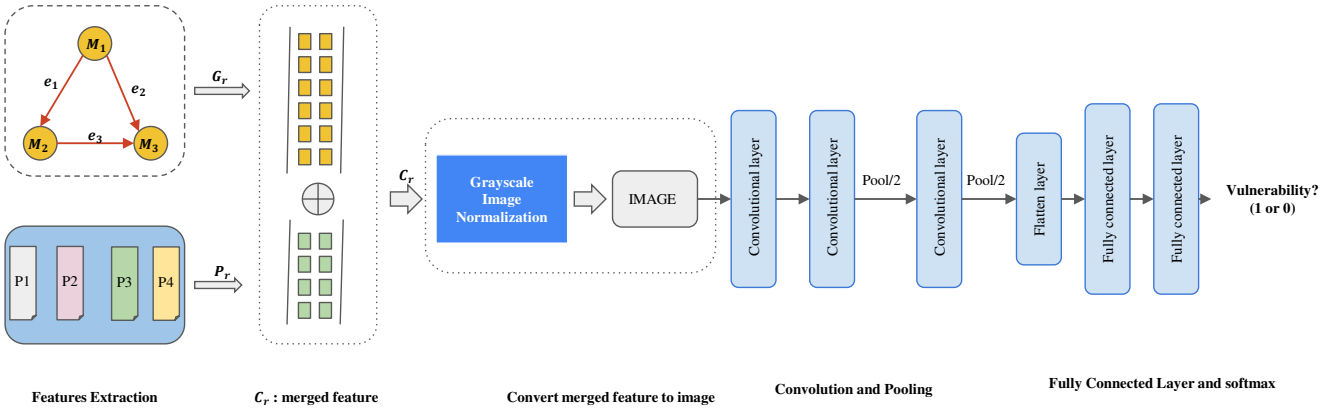
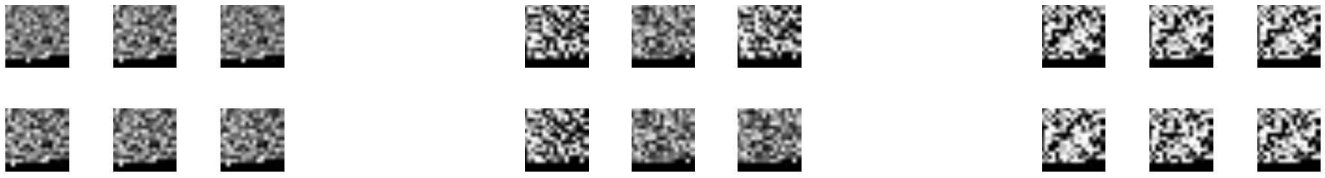


Figure 6. The process of vulnerability detection. First, extract features from smart contract source code. Then, combines G_r and P_r into the merged feature C_r. Then, the merged feature C_r is normalized to create gray image. Finally, Convolutional neural network is used to output the vulnerability detection results.



(a) Reentrancy (a) Timestamp dependence (a) Infinite loop

Figure 7. The source code after normalizing to a gray scale image of 3 vulnerabilities

of G_r as it is a 1-dim vector, at the same time reshape P_r into a 1-dim vector, which has a length equal to the sum of the lengths of G_r and P_r. The reason is that we convert vector P_r from 2-dimensional vector to 1-dimensional so that we can combine it with 1-dimensional vector G_r. After normalizing both vectors G_r and P_r to a 1-dimensional vector, we proceed to combine these two vectors by concatenating vector P_r to vector G_r to form vector C_r, which has a length equal to the sum of the lengths of G_r and P_r. After combining graph features G_r and pattern features P_r, we continue to append k features with zero value at the end of C_r so that the size of C_r is equivalent to the size of an mxn image after it is stretched.

Grayscale Image Normalization. After combining graph features G_r with pattern features P_r and appending 0 values to form a vector with the same size as an mxn image. However, the values of features of C_r still belong to different value domains, so we build a normalization procedure to convert the values of these features to the values of pixels [0,255]. Specifically, we use the Min-Max Scaling method [8] to transform the c_i values of C_r to the interval [0;1], then normalize the scaled C_r to C'_r in the interval [0;255] by multiplying by 255. Finally, reshape the vector C'_r into a matrix of size $m \times n$ corresponding to a grayscale image. Note that, the normalized grayscale images from the original raw data are the source code of the smart contract such as the solidity source code. Labeling will be carried out from the analysis of these source codes and after normalizing to gray scale images, the labels are also reassigned.

$$c'_i = \frac{c_i - min(C_i)}{max(C_i) - min(C_i)} \times 255$$

The images after normalizing the grayscale for each vulnerability will have the form as shown in Fig. 7. Obviously, the feature of transformed grayscale image can be used for classification.

5. Vulnerability Detection

After extracting graph features and pattern features from smart contract source code. We combine them and normalize grayscale images. We construct a CNN to train and detect smart contract vulnerability. This networks are fed with a large number of the normalized images generated from the functions of smart contract with the ground truth labels. After that, the trained models are employed to absorb the normalized image, then to yield a vulnerability detection label. We filter the image by multiple convolution layers and max pooling layers, then use a flatten layer, a fully connected layer and a softmax layer.

The convolution layer learns to select the different weights to the pixels, while the max pooling layer high-

lights the important pixels. Then flatten layer to flatten the n-dimensional matrix into a 1-dimensional matrix before feeding into the fully connected layer and softmax layer to give the final estimated label $\hat{y}$.

IV. EVALUATION

We perform model testing on two datasets which are real information such as ESC[6] is collected from Ethereum for reentrancy and timestamp dependence vulnerabilities, VSC[7] is collected from VNT Chain platforms for infinite loop vulnerability.

The ESC dataset has been collected 40,932 smart contracts from Ethereum [9], from which 307,396 functions have been extracted. Of the extracted data samples, about 5013 functions contain signs of reentrancy vulnerability, about 4833 functions contain signs of timestamp dependence vulnerability and about 56,800 functions contain signs of infinite loop vulnerability.

The VSC dataset [10]. has been collected 4170 smart contracts, from which 13761 functions have been extracted.

1. Experimental Setup

We use the datasets collected from blockchain networks to perform experiments. The dataset is large, so it has high requirements on CPU performance, hard disk space and memory size. To meet the needs of our experiments, we use a PC with an Intel Core i7 CPU at 2.8 GHz, a GPU at 1080TI and 16GB of memory. Another algorithms are implemented with python, while our IGNN is implemented by using tensorflow.

In our experiments, we divide the dataset into 2 parts as training data (70% dataset) and test data (30% dataset). In most machine learning experiments, the 70%:30% data split is because testing that data ratio gives high results. The 70%:30% ratio is also used in most of the research models that we refer to, which ensures fairness when comparing our research results with previous studies. Also, in our testing, we use results such as accuracy, recall, precision, F1, which are completely similar to the results of other tests. In addition, we also try to use similar data sets and experimental environments with other experiments to ensure fairness when comparing those results.

2. Comparison with Existing Methods

We compare with the existing methods that do not apply deep learning. Such methods are explained as follows.

- Oyente [4]: is a tool that uses the symbolic execution on control flow graph (CFG) for detecting the security patterns.
- Mythril [11]: uses symbolic execution for SMT solving and taint analysis to detect a variety of security vulnerabilities. It is used as a security analysis tool for EVM bytecode.
- Smartcheck [12]: is an extensible static analysis tool for discovering vulnerabilities, other bug code issues in the smart contracts of Ethereum.
- Securify [13]: is a formal-verification based software for detecting the bugs code of smart contract bugs issue by checking compliance, violation patterns to filter false positives.
- Slither [14]: is a testing-tool that enables developers to find vulnerabilities, enhance the code comprehension, and quickly prototype custom analyses.

Comparison on Reentrancy Vulnerability Detection. In Table II, we have detailed the test results of our IGNN method and five existing methods. Comparing the test results, we see that among the state-of-the-art existing method, SLITHER has the highest result of 77.12%. Our IGNN method gives superior results with an accuracy of 95.78% higher than Slither 18.66%. F1 score of IGNN is higher than Slither's 22.63%. That shows that using of IGNN to vulnerability detection has great potential.

Comparison on Timestamp Dependence Vulnerability Detection. We continue to compare the results of vulnerability detection timestamp dependence. Our comparison results are shown in the middle part of Table II. The state-of-the-art existing methods that achieves the highest accuracy result is 74.20% while IGNN achieves 90.12% which is 15.92% higher than that. This result shows that existing methods using predefined rules can be easily fooled and fail to detect new signs of vulnerability. Moreover, IGNN keeps delivering the best performance in terms of all the four metrics.

Comparison on Infinite Loop Vulnerability Detection. We evaluate our IGNN for infinite loop vulnerability by comparing test results with several methods including:

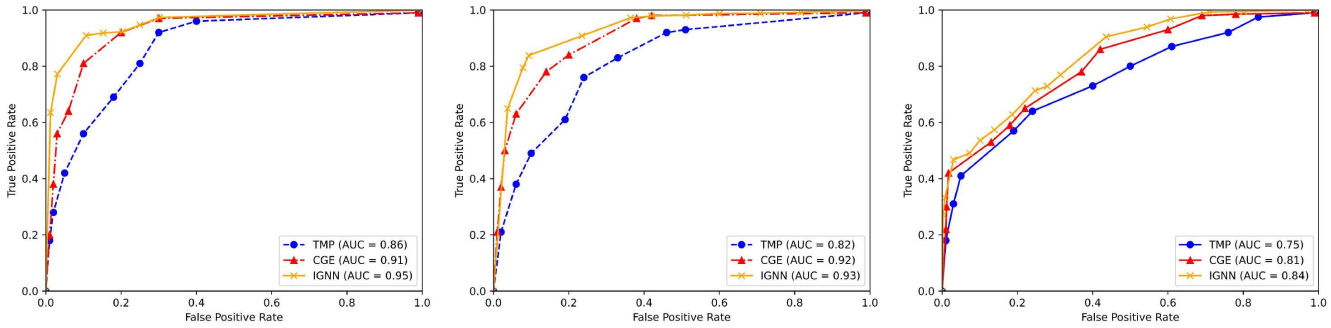
- Jolt [15] checks of 2 consecutive loops, thereby detecting the infinite loop security vulnerability.
- SMT [16] automates detection of infinite loop bugs by relying on satisfiability modulo theories.
- PDA [17] performs program path-based checking for infinite loop detection.
- Looper [18] uses symbolic execution to detect loop bugs.

The last vulnerability we compare is the Infinity Loop. Comparing our method with 4 other methods, the results are shown in the right part of Table II. Looking at Table II, it can be seen that the accuracy of the existing vulnerability detection methods is quite low. One of the reasons is the variety of Infinity vulnerability detection rules, this

[6]Ethereum Smart Contracts
[7]VNT chain Smart Contracts

TABLE II
PERFORMANCE COMPARISON (ACCURACY, RECALL, PRECISION, AND F1 SCORE). A TOTAL OF SEVENTEEN METHODS ARE INVESTIGATED IN THE COMPARISON, INCLUDING STATE-OF-THE ART VULNERABILITY DETECTION METHODS, NEURAL NETWORK-BASED ALTERNATIVES, DR-GCN, TMP, CGE, AND IGNN. "–" DENOTES NOT APPLICABLE

Methods	Reentrancy (ESC dataset)				Timestamp dependence (ESC dataset)				Methods	Infinite Loop (VSC dataset)			
	Acc(%)	Recall(%)	Precision(%)	F1(%)	Acc(%)	Recall(%)	Precision(%)	F1(%)		Acc(%)	Recall(%)	Precision(%)	F1(%)
Smartcheck	52.97	32.08	25.00	28.10	44.32	37.25	39.16	38.18	Jolt	42.88	23.11	38.23	28.81
Oyente	61.62	54.71	38.16	44.96	59.45	38.44	45.16	41.53	PDA	46.44	21.73	42.96	28.26
Mythril	60.54	71.69	39.58	51.02	61.08	41.72	50.00	45.49	SMT	54.04	39.23	55.69	45.98
Securify	71.89	56.60	50.85	53.57	–	–	–	–	Looper	59.56	47.21	62.72	53.87
Slither	77.12	74.28	68.42	71.23	74.20	72.38	67.25	69.72	–	–	–	–	–
Vanilla-RNN	49.64	58.78	49.82	50.71	49.77	44.59	51.91	45.62	Vanilla-RNN	49.57	47.86	42.10	44.79
LSTM	53.68	67.82	51.65	58.64	50.79	59.23	50.32	54.41	LSTM	51.28	57.26	44.07	49.80
GRU	54.54	71.30	53.10	60.87	52.06	59.91	49.41	54.15	GRU	51.70	50.42	45.00	47.55
GCN	77.85	78.79	70.02	74.15	74.21	75.97	68.35	71.96	GCN	64.01	63.04	59.96	61.46
DR-GCN	81.47	80.89	72.36	76.39	78.68	78.91	71.29	74.91	DR-GCN	68.34	67.82	64.89	66.32
TMP	84.48	82.63	74.06	78.11	83.45	83.82	75.05	79.19	TMP	74.61	74.32	**73.89**	74.10
CGE	89.15	87.62	85.24	86.41	89.02	88.10	87.41	87.75	CGE	77.26	56.38	67.51	61.44
IGNN	**95.78**	**94.69**	**93.04**	**93.86**	**90,12**	**89.09**	**87.79**	**89.04**	**IGNN**	**79.75**	**77.47**	73.64	**84.40**



(a) Reentrancy (a) Timestamp dependence (a) Infinite loop

Figure 8. ROC analysis for TMP, CGE, IGNN on the three vulnerability detection tasks. AUC stands for area under the curve.

vulnerability can appear under many different identifiers. For our IGNN, the detection results are much higher with an accuracy of 79.75%, higher than existing methods with the highest accuracy of 20.19%.

3. Comparison Results

We continue to compare our method with other neural networks to discover which method performs the best in detecting smart contract vulnerabilities. Specifically, we compare with some methods such as Vanilla-RNN (Recurrent neural network) [19], LSTM (Long Short-Term Memory networks)[20], GRU (gated recurrent units) [21], GCN (graph convolutional network) [22], DR-GCN [6], TMP (temporal message propagation network) [6], CGE (combining graph feature and expert patterns) [23].

We give the results of comparing methods on Table II and Fig. 8. Looking at Table II, it can be seen that IGNN gives very good results s in terms of all the 4 metrics. The experimental results show that conventional recurrent neural networks Vanilla-RNN, LSTM, and GRU perform no better than State-of-the-art existing methods. However, the GNN-based methods such as GCN, DR-GCN, TMP, CGE and IGNN which are capable of handling graphs, achieve significantly better results than existing methods. Furthermore, it can be seen that converting graph data to images shows better results, it also shows that detecting feature features based on images has better results when done with graphs.

V. CONCLUSION

We have proposed using the Convolutional Neural Network model for automatic detection of smart contract vulnerabilities. Different from the existing methods of using rules defined by experts, leading to low detection accuracy and non-scalable, We combine contract graph with defined security pattern and normalize to grayscale images. We also explore the use of Convolutional Neural Network to learn from normalized images from contract graph and security pattern. Our experiments show superiority over existing vulnerability detection methods and other neural network-based methods. We believe this will be an important step

to further research and develop deep learning techniques in smart contract vulnerability detection. In the future, we will explore this architecture on other vulnerabilities and extend the ability to combine the features of the vulnerabilities instead of training each vulnerability separately.

REFERENCES

[1] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F.-Y. Wang, "An overview of smart contract: Architecture, applications, and future trends," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp. 108–113.

[2] Y. Liu, J. Xu, and B. Cui, "Smart contract vulnerability detection based on symbolic execution technology," in *Cyber Security*, W. Lu, Y. Zhang, W. Wen, H. Yan, and C. Li, Eds. Singapore: Springer Nature Singapore, 2022, pp. 193–207.

[3] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," 08 2019.

[4] 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*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[5] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[6] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.

[7] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, vol. abs/1711.00740, 2017. [Online]. Available: http://arxiv.org/abs/1711.00740

[8] "everything you need to know about min-max normalization," https://towardsdatascience.com/everything-you-need-to-know-about-min-max-normalization-in-python-b79592732b79, 2020, website.

[9] "Ethereum," https://github.com/ethereum/go-ethereum, 2015, website.

[10] "Vntchain," https://github.com/vntchain/go-vnt, 2018, website.

[11] "A framework for bug hunting on the ethereum blockchain," https://github.com/ConsenSys/mythril, 2017, website.

[12] 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*, 2018, pp. 9–16.

[13] 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*, 2018, pp. 67–82.

[14] 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)*. IEEE, 2019, pp. 8–15.

[15] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard, "Detecting and escaping infinite loops with jolt," in *European Conference on Object-Oriented Programming*. Springer, 2011, pp. 609–633.

[16] M. Kling, S. Misailovic, M. Carbin, and M. Rinard, "Bolt: on-demand infinite loop escape in unmodified binaries," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 431–450, 2012.

[17] A. Ibing and A. Mai, "A fixed-point algorithm for automated static detection of infinite loops," in *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*. IEEE, 2015, pp. 44–51.

[18] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen, "Looper: Lightweight detection of infinite loops at runtime," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 161–169.

[19] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *Proceedings of International Conference on Neural Networks (ICNN'96)*, vol. 1. IEEE, 1996, pp. 347–352.

[20] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," 2014.

[21] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv preprint arXiv:1412.3555*, 2014.

[22] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks. 2017," *ArXiv abs/1609.02907*, 2017.

[23] 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*, 2021.

Ta Minh Thanh is currently an associate professor and vice dean of Faculty of Information Technology in Le Quy Don Technical University Vietnam. He is also a Postdoctoral Fellow of the Department of Mathematical and Computing Sciences at Tokyo Institute of Technology. He received his B.S. and M.S in Computer Science from National Defense Academy, Japan, in 2005 and 2008 and his Ph.D. from Tokyo Institute of Technology, Japan, in 2015, respectively.He is the member of IPSJ Japan and IEEE. His research interests lie in the area of watermarking, network security, and computer vision.
Email: thanhtm@lqdtu.edu.vn

Pham Trong Linh is currently studying network technology at Le Quy Don University. His field of research is digital watermarking for digital multimedia .
Email: tronglinhmta0611@gmail.com