Smart Contract Vulnerability Detection Using Graph Neural Networks

Yuan Zhuang[1,*], **Zhenguang Liu**[1,*], **Peng Qian**[1,*], **Qi Liu**[2], **Xiang Wang**[3], **Qinming He**[4]
[1]Zhejiang Gongshang University
[2]University of Oxford
[3] National University of Singapore
[4] Zhejiang University
zhuangyuan2020@outlook.com, liuzhenguang2008@gmail.com, messi.qp711@gmail.com,
qi.liu@cs.ox.ac.uk, xiangwang1223@gmail.com, hqm@zju.edu.cn

Abstract

The security problems of smart contracts have drawn extensive attention due to the enormous financial losses caused by vulnerabilities. Existing methods on smart contract vulnerability detection heavily rely on fixed expert rules, leading to low detection accuracy. In this paper, we explore using graph neural networks (GNNs) for smart contract vulnerability detection. Particularly, we construct a *contract graph* to represent both syntactic and semantic structures of a smart contract function. To highlight the major nodes, we design an elimination phase to normalize the graph. Then, we propose a degree-free graph convolutional neural network (DR-GCN) and a novel temporal message propagation network (TMP) to learn from the normalized graphs for vulnerability detection. Extensive experiments show that our proposed approach significantly outperforms state-of-the-art methods in detecting three different types of vulnerabilities.

1 Introduction

Blockchain technology is developing rapidly due to its decentralization and tamper-free nature [Tsankov *et al.*, 2018]. A blockchain is essentially a distributed and shared transaction ledger, maintained by all the miners in the blockchain network following a consensus protocol [Sankar *et al.*, 2017]. Smart contracts are programs automatically running on the blockchain. However, ill-designed smart contracts expose vulnerabilities, which are perfect targets for network attacks. One notable example is the DAO event, where the hackers exploit the reentrancy bug of *The DAO* contract to steal 3.6 million Ether (Cryptocurrency of Ethereum). The case is not isolated and several security vulnerabilities are discovered and exploited every few months [†]. According to the statistics of SlowMist Hacked [‡], blockchain networks have suffered more than 10 billion USD losses due to the security issues of smart contracts.

Current approaches for smart contract vulnerability detection are mainly inspired by existing testing methods from the programming language community, revolving around symbolic execution [Luu *et al.*, 2016; Tsankov *et al.*, 2018] and dynamic execution methods [Jiang *et al.*, 2018; Liu *et al.*, 2018b]. We scrutinized the released implementation of existing methods, and empirically observe that they suffer from two key problems. *First*, existing methods heavily rely on several expert-defined hard rules (or patterns) to detect smart contract vulnerability. *However*, expert rules are error-prone and some complex patterns are non-trivial to be covered. Crudely using several hard rules leads to high *false-positive* and *false-negative* rates, and crafty attackers may easily bypass the rules to perform attacks. *Second*, since the rules are contributed by a few 'centralized' experts who develop the detection tools, their scalability is inherently limited. As the number of smart contracts is increasing rapidly, it is impossible for a few experts to sift through *all* the contracts to design precise rules, while the knowledge of other 'decentralized' experts cannot be incorporated to improve the model.

Our method. To address these problems, we propose novel methods beyond the rule-based framework. Specifically, we characterize the source code of a smart contract as a *contract graph* according to the data- and control- dependencies between program statements. Nodes in the graph represent critical function invocations or variables while edges capture their temporal execution traces. Since most GNNs are inherently flat during information propagation, we design an elimination phase to normalize the graph. We extend GCN to a degree-free GCN (DR-GCN) to handle the normalized graphs. Further, we take into account the distinct roles and temporal relationships of different program elements and propose a novel temporal message propagation network (TMP). We conducted extensive experiments on more than 300,000 real-world smart contract functions, results show that our approaches significantly and consistently outperform state-of-the-art methods on the detection of different types of vulnerabilities including *reentrancy*, *timestamp dependence*, and *infinite loop* vulnerabilities. Our implementations are released to facilitate future research.

[*]The first three authors are of equal contribution to this work. Zhenguang Liu contributes to the idea, Yuan Zhuang and Peng Qian contribute to implements and datasets. Zhenguang Liu is the corresponding author.

[†]The dao website, 2016. http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413

[‡]Slowmist hacked website, 2019. https://hacked.slowmist.io/en/.

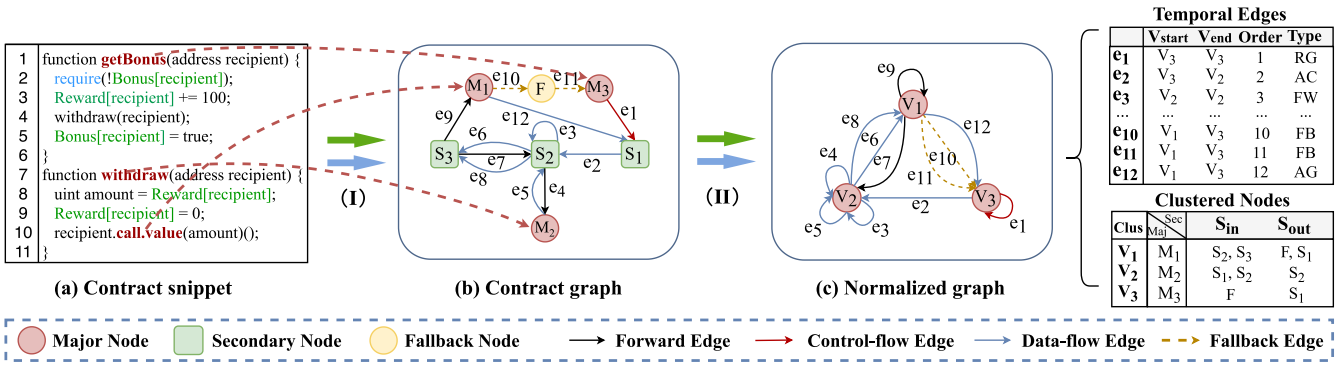


Figure 1: The graph generation and normalization phases of our method. (a) shows the source code of a smart contract; (b) visualizes the graph extracted from the source code. Nodes in circle denote major nodes and nodes in square represents secondary nodes. (c) demonstrates the graph after normalization.

Contributions. To summarize, our key contributions are: i) We introduce a novel temporal message propagation network (TMP) and a degree-free GCN (DR-GCN) to automatically detect smart contract vulnerabilities. ii) We propose to characterize the contract function source code as *contact graphs*, and explicitly normalize the graph for highlighting the key nodes. iii) Our methods set the new state-of-the-art performance on smart contract vulnerability detection, and overall provide insights into the challenges and opportunities.

2 Problem Statement

Problem formulation. Presented with the source code of a smart contract, we are interested in developing a fully automated approach that can detect vulnerabilities at the function level. That is, we are to estimate the label $\hat{y}$ for each smart contract function *SC*, where $\hat{y} = 1$ represents *SC* has a vulnerability of a certain type while $\hat{y} = 0$ denotes *SC* is safe. In this paper, we focus on three types of vulnerabilities:

Reentrancy is a well-known vulnerability that caused the infamous DAO attack. In Ethereum, when a smart contract function F_1 transfers money to a recipient contract C_1, the fallback function of C_1 will be automatically triggered. C_1 may invoke back to F_1 in its fallback function to reenter F_1 for stealing money. Since the current execution of F_1 waits for the transfer to finish, C_1 can make use of the intermediate state F_1 is in to succeed in stealing.

Infinite loop is a common vulnerability in smart contracts. The program of a function may contain an iteration or loop with no exit condition or the exit condition cannot be reached, i.e., an infinite loop. The fallback mechanism in smart contracts rises a new possibility of this non-termination bug, namely a cycled call between functions and the fallback function. *For example*, function A invokes function B with incorrect arguments, which will automatically trigger the execution of the fallback function in this contract. Suppose the fallback function further invokes function A, this will leads to a call loop between A and the fallback function.

Timestamp dependence vulnerability exists when a smart contract uses the block timestamp as a triggering condition to execute some critical operations, e.g., sending Ether or determining the winner of a game. The miner in Ethereum has the freedom to set the timestamp of a block within a short time interval (< 900 seconds) [Jiang *et al.*, 2018]. Therefore, miners may manipulate the block timestamps to gain illegal benefits.

3 Our Method

Method overview. The overall architecture of our method consists of three phases: (1) a graph generation phase, which extracts the control flow and data flow semantics from the source code and explicitly models the fallback mechanism, (2) a graph normalization phase inspired by k-partite graph, and (3) novel message propagation networks for vulnerability modeling and detection. Next, we introduce the three phases, respectively.

3.1 Graph Generation

Existing work [Allamanis *et al.*, 2018] has shown that programs can be transformed into symbolic graph representations, which are able to preserve semantic relationships between program elements. Inspired by this, we formulate a smart contract function into a *contract graph*, and assign distinct roles to different program elements (nodes). Further, we construct edges by taking their temporal order into consideration. Figs. 1(a) & (b) demonstrate a contract snippet and the graph constructed for its $getBonus$ function, respectively.

Our first insight is that different program elements in a function are not of equal importance. Therefore, we extract three categories of nodes, i.e., *major nodes*, *secondary nodes*, and *fallback nodes*.

Major nodes construction. Major nodes symbolize the invocations to customized or built-in functions that are important for detecting the specific vulnerability. For example, for reentrancy vulnerability, a major node models the invocation to a transfer function or the built-in *call.value* function, which is key to detect reentrancy. For timestamp dependence vulnerability, the built-in function invocation *block.timestamp* is extracted as a major node. For infinite loop, all the customized functions within the contract are treated as major

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

Table 1: Semantic edges summarization. All edges are classified into 4 types, namely control-flow, data-flow, forward, and fallback.

nodes. Formally, we characterize all the critical functions as major nodes, which are denoted by $M_1, M_2, \ldots, M_n$.

Secondary nodes construction. While major nodes represent important invocations, secondary nodes are used to model critical variables, e.g., *user balance* and *bonus flag*. Formally, the critical variables are defined as secondary nodes $S_1, S_2, \ldots, S_n$.

Fallback node construction. Further, we construct a *fallback* node F to stimulate the *fallback* function of an attack contract, which can interact with the function under test. The *fallback* function is a special design in smart contracts, and is the cause of many security vulnerabilities.

Edges construction. We further construct edges to model the relationships between nodes. Each edge describes a path that might be traversed through by the contract function under test, and the temporal number of the edge characterizes its order in the function. Specifically, the feature of an edge is extracted as a tuple (V_s, V_e, o, t), where V_s and V_e represent its starting and end nodes, o denotes its temporal order, and t the edge type. To capture rich semantic dependencies between nodes, we construct four types of edges, namely *control flow*, *data flow*, *forward* and *fallback edges*. The details of the semantic edges are listed in Table 1.

3.2 Contract Graph Normalization

Most graph neural networks are inherently flat when propagating information, ignoring that some nodes play more central roles than others. Moreover, different contract source code yield distinct graphs, hindering the training of graph neural networks. Therefore, we propose a node elimination process to normalize graphs.

Nodes elimination. As introduced in Section 3.1, the node of a graph is partitioned into major nodes $\{M_i\}_{i=1}^{|M|}$, secondary nodes $\{S_i\}_{i=1}^{|M|}$, and the fallback node F. We remove each secondary node S_i but pass the feature of S_i to its nearest major node. Note that if S_i has multiple nearest major nodes, its feature is passed to all of them. The fallback node is also removed similar to the secondary node. The edges connecting to the removed node are preserved but with their starting or end node moving to the corresponding major node.

Feature of major nodes. Features of major nodes are updated by aggregating features from their neighboring removed nodes. To distinguish between the original major node and its corresponding major node after aggregation, we denote the new major node of M_i as V_i. The feature of V_i is composed of three parts: i) self-feature, namely the feature of major node M_i; ii) in-features, namely features of the secondary nodes $\{P_j\}_{j=1}^{|P|}$ that are merged to M_i and having a path pointing from P_j to M_i; and iii) out-feature, namely features of the secondary nodes $\{Q_k\}_{k=1}^{|Q|}$ that are merged to M_i and having a path directs from Q_k from M_i. Fig. 1(c) illustrates the normalized graph of Fig. 1(b).

3.3 Message Propagation Neural Networks

In this subsection, we first extend the GCN to a degree-free GCN (DR-GCN), then propose a novel temporal message propagation network (TMP). Both the two proposed networks take the normalized graph G of a smart contract function as input, and output the label $\hat{y} \in \{0, 1\}$ indicating whether the function has a vulnerability of a certain type.

DR-GCN. [Kipf and Welling, 2017] proposes to apply convolutional neural networks to graph-structured data, which develops a layer-wise propagation network as:

$$X_{l+1} = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}X_l W_l\right) \tag{1}$$

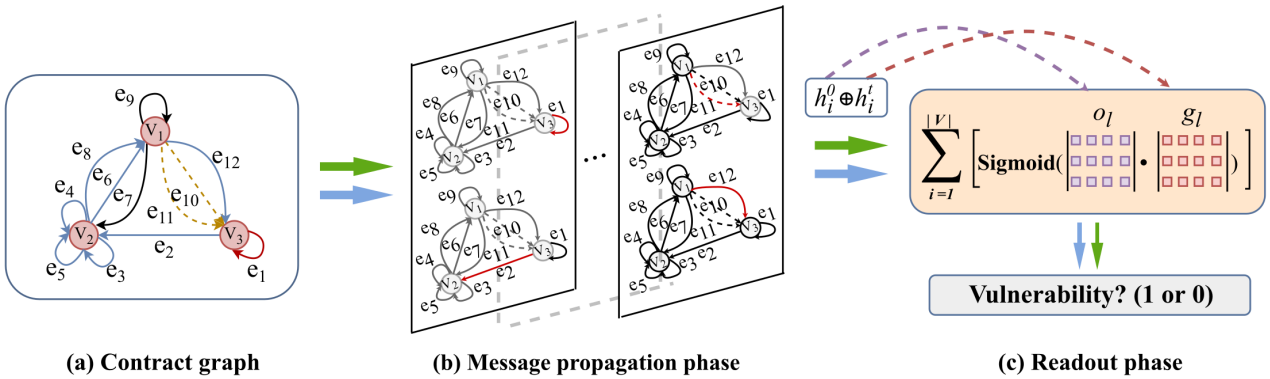
where $\hat{A} = A + I$ is the adjacency matrix (A) enhanced with self-loops (I), X_l is the feature matrix of layer l, and W_l is a trainable weight matrix. In the equation, the diagonal node degree matrix $\hat{D}$ is used to normalize $\hat{A}$. We first increase the connectivity between nodes in the normalized graph G by using the square of A. Then, we further take into account that the graph is already well normalized in our setting, and therefore remove matrix $\hat{D}$ from the equation. Finally, we arrive at the solution: $X_{l+1} = \sigma\left((A^2 + I)X_l W_l\right)$.

TMP. We also propose a TMP network, consisting of a message propagation phase and a readout phase (Fig. 2). In the message propagation phase, TMP passes information along the edges successively by following their temporal order. Then, TMP computes a label for the entire graph G by using a readout function, which aggregates the final states of all nodes in G. Formally, $G = \{V, E\}$, where V consists of all the major nodes and E contains all the edges. Denote $E = \{e_1, e_2, \ldots, e_N\}$, where e_k represents the k^{th} temporal edge.

Message propagation phase. Messages are passed along the edges, one edge per time step. At time step 0, the hidden state h_i^0 for each node V_i is initialized with the feature of V_i. At time step k, message flows through the k^{th} temporal edge e_k and updates the hidden state of V_{ek}, namely the end node of e_k. Particularly, message m_k is computed basing on h_{sk}, the hidden state of the starting node of e_k, and the edge type t_k:

$$x_k = h_{sk} \oplus t_k \tag{2}$$

$$m_k = W_k x_k + b_k \tag{3}$$



(a) Contract graph (b) Message propagation phase (c) Readout phase

Figure 2: The overall architecture of our proposed TMP. (a) The input normalized graph; b) The message propagation phase; (c) The readout phase that outputs the vulnerability detection result.

where $\oplus$ denotes concatenation operation, matrix W_k and bias vector b are network parameters. The original message x_k contains information from the starting node of e_k and edge e_k itself, which are then transformed into a vector embedding using W_k and b. After receiving the message, the end node of e_k updates its hidden state h_{ek} by aggregating information from the incoming message and its previous state. Formally, h_{ek} is updated according to:

$$\hat{h}_{ek} = tanh(Um_k + Zh_{ek} + b_1) \tag{4}$$

$$h'_{ek} = softmax(R\hat{h}_{ek} + b_2) \tag{5}$$

where U, Z, R are matrices, while b_1 and b_2 are bias vectors.

Readout phase. After successively traversing all the edges in G, TMP computes a label for G by reading out the final hidden states of all nodes. Let h_i^T be the final hidden state of the i^{th} node, we may generate the prediction label $\hat{y}$ by

$$\hat{y} = \sum_{i=1}^{|V|} f(h_i^T) \tag{6}$$

where f is a mapping function, e.g., a neural network, and $|V|$ denotes the number of major nodes. However, we found that the differences between the final hidden state h_i^T and the original hidden state h_i^0 are informative in the vulnerability detection task. Therefore, we instead consider to compute $\hat{y}$ as follows:

$$s_i = h_i^T \oplus h_i^0 \tag{7}$$

$$g_i = softmax(W_g^{(2)}(tanh(b_g^{(1)} + W_g^{(1)}s_i)) + b_g^{(2)}) \tag{8}$$

$$o_i = softmax(W_o^{(2)}(tanh(b_o^{(1)} + W_o^{(1)}s_i)) + b_o^{(2)}) \tag{9}$$

$$\hat{y} = \sum_{i=1}^{|V|} Sigmoid(o_i \odot g_i) \tag{10}$$

where $\odot$ denotes element-wise product. W_j, $b_j^{(1)}$, and $b_j^{(2)}$, with subscript $j \in \{g, o\}$ are model parameters to be learned.

Both the two networks DR-GCN and TMP are trained for contract vulnerability detection. During training, networks are fed with a large number of normalized graphs constructed from smart contract functions, together with their ground truth labels. Then, the trained models are employed to absorb a normalized graph and yield a vulnerability detection label. We would like to point out that we developed automation tools for converting source code to normalized graphs, therefore, the whole procedure is fully automated.

4 Experiments

4.1 Datasets and Experimental Settings

Datasets. Extensive experiments are conducted on all the smart contracts that have source code on the Ethereum and VNT Chain platforms. We denote the two real-world smart contract datasets as `ESC` (Ethereum Smart Contracts) and `VSC` (VNT chain Smart Contracts), respectively.

- `ESC` consists of 40,932 Ethereum smart contracts with roughly $307,396$ functions in total. Among the functions, around $5,013$ functions possess at least one invocation to $call.value$, making them potentially affected by the reentrancy vulnerability. Around $4,833$ functions contain the $block.timestamp$ statement, making them susceptible to the timestamp dependence vulnerability.
- `VSC` consists of $4,170$ smart contracts collected from the VNT Chain [*], roughly containing $13,761$ functions. VNT Chain is an experimental public blockchain platform proposed by companies and universities from Singapore, China, and Australia.

Experimental settings. We compared our approaches (DR-GCN and TMP) with a total of twelve other methods, namely four existing smart-contract vulnerability detection methods (*Oyente* [Luu *et al.*, 2016], *Mythril* [Mueller, 2017], *Smartcheck* [Tikhomirov *et al.*, 2018], and *Securify* [Tsankov *et al.*, 2018]), four neural network based methods (Vanilla-RNN, LSTM, GRU, and GCN), and four program loop detection methods (*Jolt* [Carbin *et al.*, 2011], *PDA* [Ibing and Mai, 2015], *SMT* [Kling *et al.*, 2012], and *Looper* [Burnim *et al.*, 2009]). For each dataset, we randomly pick 20% contracts as the training set while the remainings are utilized for the testing set. In the comparison, metrics *accuracy*, *recall*, *precision*, and *F1 score* are all involved. In consideration of the

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

Methods	Reentrancy				Timestamp dependence				Methods	Infinite Loop			
	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
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**

Table 2: Performance comparison in terms of *accuracy*, *recall*, *precision*, and *F1 score*. A total of fourteen methods are investigated in the comparison, including state-of-the-art vulnerability detection methods, neural network based alternatives, our methods DR-GCN and TMP. '–' denotes not applicable.

Metrics	Reentrancy				Timestamp dependence				Infinite loop			
	DR-GCN-WON	DR-GCN	TMP-WON	TMP	DR-GCN-WON	DR-GCN	TMP-WON	TMP	DR-GCN-WON	DR-GCN	TMP-WON	TMP
Acc(%)	77.08	**81.47**	81.91	**84.48**	72.56	**78.68**	80.03	**83.45**	63.93	**68.34**	70.03	**74.61**
Recall(%)	76.83	**80.89**	80.49	**82.63**	74.38	**78.91**	81.30	**83.82**	63.16	**67.82**	71.82	**74.32**
Precision(%)	68.36	**72.36**	71.44	**74.06**	67.46	**71.29**	72.69	**75.05**	60.11	**64.89**	69.94	**73.89**
F1(%)	72.35	**76.39**	75.70	**78.11**	70.75	**74.91**	76.75	**79.19**	61.59	**66.32**	70.87	**74.10**

Table 3: Accuracy comparison between DR-GCN, TMP, and their variants on the three vulnerability detection tasks.

distinct features of different platforms, experiments on *reentrancy* vulnerability and *timestamp dependence* vulnerability are conducted on the ESC dataset, while experiments on *infinite loop* vulnerability detection are conducted on the VSC dataset.

4.2 Comparison with Existing Methods

In this subsection, we first benchmark the proposed approaches (DR-GCN and TMP) against state-of-the-art methods on the *reentrancy*, *timestamp dependence*, and *infinite loop* vulnerabilities, respectively. Then, we compare our approaches with other neural network based methods.

Comparison on Reentrancy Vulnerability Detection

First, we compare our DR-GCN and TMP methods with state-of-the-art smart contract vulnerability detection methods, namely *Oyente* [Luu *et al.*, 2016], *Mythril* [Mueller, 2017], *Smartcheck* [Tikhomirov *et al.*, 2018], and *Securify* [Tsankov *et al.*, 2018], on the reentrancy vulnerability detection task. The performance of different methods is presented in the left of Table 2, where metrics *accuracy*, *recall*, *precision*, and *F1 score* are engaged.

From the quantitative results of Table 2, we have the following observations. First, we find that existing tools have not yet achieved a satisfactory accuracy on reentrancy vulnerability detection, e.g., the state-of-the-art tool yields a 71.89% accuracy. Second, TMP outperforms state-of-the-art methods by a large margin. More specifically, TMP achieves an accuracy of 84.48%, gaining a 12.39% accuracy improvement over state-of-the-art tools. Besides, the F1 score of TMP is 24.54% higher than existing methods. Thirdly, DR-GCN also achieves better results than other existing methods in terms of all the four metrics. The strong empirical evidences reveal the great potential of applying graph neural networks to smart contract vulnerability detection.

Comparison on Timestamp Dependence Vulnerability Detection

We then compare the proposed methods with state-of-the-art smart contract vulnerability detection tools on the timestamp dependence vulnerability detection task. The comparison results are demonstrated in the middle of Table 2. The state-of-the-art method has obtained a 61.08% accuracy on timestamp dependence vulnerability detection, which is quite low. This may stem from the fact that most of existing methods detect timestamp dependence vulnerability by crudely checking whether there is *block.timestamp* statement in the function. Moreover, in consistent with the results on reentrancy vulnerability detection, TMP keeps delivering the best performance in terms of all the four metrics, while DR-GCN ranks the second. In particular, TMP gains a 22.37% accuracy improvement over state-of-the-art method.

We further look into the existing smart contract vulnerability detection tools to investigate the reasons behind the observations. *Smartcheck* fundamentally depends on a few rigid and simple logic rules to detect vulnerabilities, which leads to low accuracy and F1 score. *Oyente* employs data flow analysis to improve the accuracy, while its underlying patterns for detecting vulnerabilities are not so accurate. Regarding *Mythril*, it requires sophisticated techniques such as taint analysis or manual audit, which attains a medium accuracy. Unlike other methods, *Securify* classifies smart contract functions into violations, warnings, and compliances, where violation denotes the function is guaranteed to have the vulnerability (positive), and compliance denotes the function is safe (negative). We treat all warnings as negative since users are usually attracted by violations while ignoring a lot of warnings. *Securify* performs better than other existing methods, but has a high false negative rate.

Comparison on Infinite Loop Vulnerability Detection

For the infinite loop vulnerability detection, we compare our methods against available tools including *Jolt* [Carbin *et al.*,

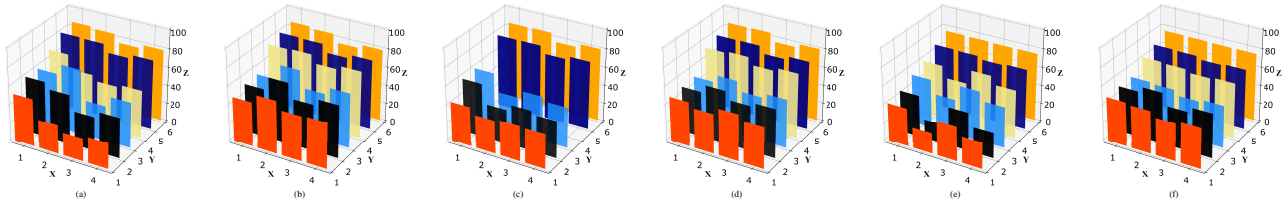
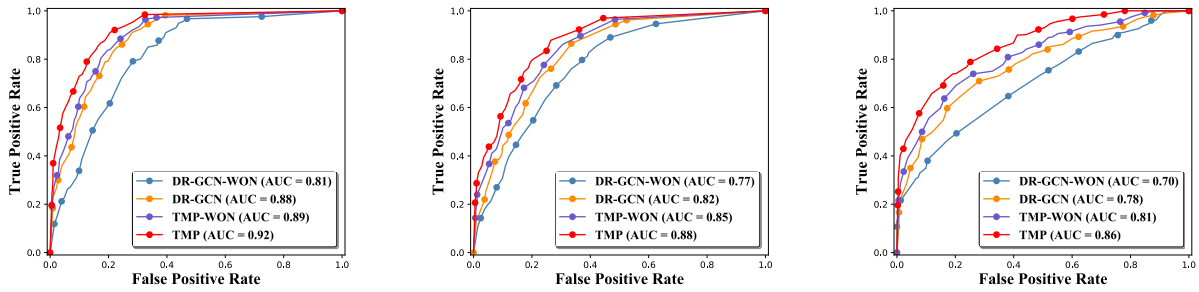


Figure 3: Visually comparison: (a) & (b) present comparison results of reentrancy vulnerability detection on the ESC dataset, while (c) & (d) present comparison results of timestamp dependence detection, (e) & (f) show comparison results of infinite loop vulnerability detection on the VSC dataset. In (a) & (c), the 6 rows from front to back denote the Smartcheck, Oyente, Mythril, Securify, DR-GCN, and TMP methods, respectively. In (e), the 5 rows from front to back denote the Jolt, PDA, SMT, Looper, DR-GCN, and TMP methods, respectively. In (b) & (d) & (f), the 6 rows from front to back denote the Vanilla-RNN, LSTM, GRU, GCN, DR-GCN, and TMP methods, respectively. For each row in the figures, *accuracy*, *recall*, *precision*, and *F1 score* are respectively demonstrated from left to right.



(a) Reentrancy (b) Timestamp (c) Infinite loop

Figure 4: ROC analysis for DR-GCN, TMP, and their variants on the three vulnerability detection tasks. AUC stands for area under the curve.

2011], *SMT* [Kling *et al.*, 2012], *PDA* [Ibing and Mai, 2015], and *Looper* [Burnim *et al.*, 2009]. We empirically find that almost all existing methods fail to detect the infinite loop bug caused by the *fallback* mechanism of smart contracts. In contrast, our methods can successfully identify this vulnerability. This is because we explicitly model the fallback mechanism of smart contracts and consider data dependencies and control dependencies between program elements. Quantitative results are illustrated in the right of Table 2. From the table, we see that TMP consistently and significantly outperforms the other methods on the infinite loop vulnerability detection task. In particular, TMP and DR-GCN respectively achieve a 15.05% and 8.78% accuracy improvement over state-of-the-art methods.

We further visualize the comparison results in Fig. 3(a), (c), and (e). Fig. 3(a) and Fig. 3(c) present comparison results of reentrancy vulnerability detection and timestamp dependence vulnerability detection, respectively. The 6 rows (in different colors) from front to back denote the Smartcheck, Oyente, Mythril, Securify, DR-GCN, and TMP methods, respectively. For each row in the figures, *accuracy*, *recall*, *precision*, and *F1 score* are respectively demonstrated from left to right. Fig. 3(e) shows comparison results of infinite loop vulnerability detection, where the the 6 rows from front to back denote the Jolt, PDA, SMT, Looper, DR-GCN, and TMP methods, respectively. We can clearly observe that DR-GCN and TMP outperform existing methods.

Comparison with Neural Network Based Methods

In order to seek out which neural network architectures could succeed in smart contract vulnerability detection, we also compare our methods with other neural network alternatives. Specifically, Vanilla-RNN, LSTM, GRU, and GCN are compared with our DR-GCN and TMP networks. For fair comparison, all the methods are presented with the vector representation of the normalized graph extracted from the source code and are required to detect the corresponding bugs. We illustrate the results of different models in terms of *accuracy*, *recall*, *precision*, and F1 score in Table 2. Fig. 3(b), (d), and (f) further visualize the results.

Interestingly, experimental results show that conventional recurrent neural networks Vanilla-RNN, LSTM, and GRU perform no better than existing vulnerability detection methods. In contrast, graph neural networks GCN, DR-GCN, and TMP, which are capable of handling graphs, achieve significantly better results than existing methods. This suggests that blindly treat the source code as a sequence is not suitable for the vulnerability detection task, while modeling the source code into graphs and adopting graph neural networks is promising. We conjecture that conventional recurrent models lose valuable information from smart contract code since they ignore the structural information of contract programs, such as the data-flow and invocation relationships.

We would like to highlight that the proposed *TMP* and *DR-GCN* model consistently and significantly outperforms other neural network models in terms of all the 4 metrics. Besides TMP and DR-GCN, the GCN model performs the best. The

accuracies of GCN and DR-GCN are lower than TMP. We attribute this to the fact that GCN fails to capture the temporal information induced by data flow and control flow, which is explicitly addressed in our TMP model using ordered edges.

4.3 Study on The Effect of Graph Normalization

By default TMP adopts the graph normalization module to highlight the major nodes in the graph, it is interesting to see the effect of removing this module. We removed the graph normalization phase from TMP and DR-GCN, and compared them with the default TMP and DR-GCN. The two variants are respectively denoted as TMP-WON and DR-GCN-WON, where WON is short for *without normalization*. Quantitative results are summarized in Table 3. We can see that with the proposed normalization module, the performance of both DR-GCN and TMP is better. For example, on the reentrancy vulnerability detection task, the DR-GCN model obtains a 4.39% and 4.04% improvement in terms of accuracy and F1 score, respectively, while TMP gains a 2.57% and 2.41% improvement in accuracy and F1 score.

Fig. 4 further plots the ROC curves of DR-GCN, TMP, and their variants. We adopt Receiver Operating Characteristic (ROC) analysis to show the impact of the graph normalization module. AUC (area under the curve) is used as the measure for performance, the higher AUC the better performance. Fig. 4(a) demonstrates that DR-GCN and TMP achieve better results on the reentrancy detection task with the normalization module, namely the AUC increases by 0.07 and 0.03, respectively. Regarding the timestamp dependence detection task, DR-GCN and TMP obtain a 0.05 and 0.03 improvement in AUC (shown in Fig. 4(b)). For the infinite loop detection task, DR-GCN and TMP gain a 0.08 and 0.05 improvement in AUC (shown in Fig. 4(c)). The experimental results suggest that program elements should contribute distinctly in vulnerability detection rather than having equal contributions.

5 Related Work

Smart contract vulnerability detection. Smart contract vulnerability detection is one of the fundamental problems in blockchain security. Current work mainly relies on symbolic execution methods, such as Oyente [Luu *et al.*, 2016], Maian [Nikolić *et al.*, 2018] and Securify [Tsankov *et al.*, 2018], which suffer from high false negative rates due to the inability to explore all possible program paths. Recent work [Jiang *et al.*, 2018] explores dynamic execution for vulnerability detection, but requires a hand-crafted agent contract for reentrancy detection, preventing it from fully automated application.

Graph neural networks (GNNs). With remarkable success of neural networks in various fields [Cheng *et al.*, 2019] [Liu *et al.*, 2018a], graph neural network has received increasing attention. Existing approaches roughly cast into two categories: (1) *Spectral-based approaches* generalize well-established neural models like CNNs for graph data. For instance, GCN [Kipf and Welling, 2017] implements a first-order approximation of spectral graph convolutions [Defferrard *et al.*, 2016], while [Li *et al.*, 2018] proposes a graph CNN capable of processing input data of arbitrary graph structure. (2) *Spatial-based methods* inherit ideas from recurrent GNNs and adopt message passing for graph convolutions. [Micheli, 2009] directly sums up a node's neighborhood information for graph convolutions, while recent work [Veličković *et al.*, 2017] and [Zhang *et al.*, 2018] learn different weights of neighboring nodes using attention mechanisms.

6 Conclusion

In this paper, we have proposed a fully automated vulnerability analyzer for smart contracts. In contrast to existing methods, we explicitly model the fallback mechanism of smart contracts, consider rich dependencies between program elements, and explore the possibility of using novel graph neural networks for vulnerability detection. Extensive experiments show that our method significantly outperforms state-of-the-art methods and other neural networks. We believe our work is an important step towards revealing the potential of deep learning methods on smart contract vulnerability detection tasks.

Acknowledgments

This paper is supported by the National Key R&D Program of China (2017YFB1401300, 2017YFB1401304), the National Natural Science Foundation of China (No. 61902348), the Natural Science Foundation of Zhejiang Province, China (Grant No. LQ19F020001).

References

[Allamanis *et al.*, 2018] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

[Burnim *et al.*, 2009] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of the International Conference on Automated Software Engineering*, pages 161–169. IEEE Computer Society, 2009.

[Carbin *et al.*, 2011] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C Rinard. Detecting and escaping infinite loops with jolt. In *European Conference on Object-Oriented Programming*, pages 609–633. Springer, 2011.

[Cheng *et al.*, 2019] Zhiyong Cheng, Xiaojun Chang, Lei Zhu, Rose Catherine Kanjirathinkal, and Mohan S. Kankanhalli. MMALFM: explainable recommendation by leveraging reviews and images. *ACM Trans. Inf. Syst.*, 37(2):16:1–16:28, 2019.

[Defferrard *et al.*, 2016] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

[Ibing and Mai, 2015] Andreas Ibing and Alexandra Mai. A fixed-point algorithm for automated static detection of infinite loops. In *International Symposium on High Assurance Systems Engineering*, pages 44–51. IEEE, 2015.

[Jiang *et al.*, 2018] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the International Conference on Automated Software Engineering*, pages 259–269. ACM, 2018.

[Kipf and Welling, 2017] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[Kling *et al.*, 2012] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. *ACM SIGPLAN Notices*, 47(10):431–450, 2012.

[Li *et al.*, 2018] Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. Adaptive graph convolutional neural networks. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[Liu *et al.*, 2018a] An-An Liu, Ning Xu, Hanwang Zhang, Weizhi Nie, and Yuting Su. Multi-level policy and reward reinforcement learning for image captioning. 2018.

[Liu *et al.*, 2018b] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of International Conference on Software Engineering: Companion Proceeedings*, pages 65–68. ACM, 2018.

[Luu *et al.*, 2016] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Conference on computer and communications security*, pages 254–269. ACM, 2016.

[Micheli, 2009] Alessio Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

[Mueller, 2017] Bernhard Mueller. A framework for bug hunting on the ethereum blockchain. Webiste, 2017. https://github.com/ConsenSys/mythril.

[Nikolić *et al.*, 2018] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the Annual Computer Security Applications Conference*, pages 653–663. ACM, 2018.

[Sankar *et al.*, 2017] Lakshmi Siva Sankar, M Sindhu, and M Sethumadhavan. Survey of consensus protocols on blockchain applications. In *International Conference on Advanced Computing and Communication Systems*, pages 1–5. IEEE, 2017.

[Tikhomirov *et al.*, 2018] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 9–16. IEEE, 2018.

[Tsankov *et al.*, 2018] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Conference on Computer and Communications Security*, pages 67–82. ACM, 2018.

[Veličković *et al.*, 2017] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[Zhang *et al.*, 2018] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294*, 2018.