# Combining Graph Neural Networks with Expert Knowledge for Smart Contract Vulnerability Detection

Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang

**Abstract**—Smart contract vulnerability detection draws extensive attention in recent years due to the substantial losses caused by hacker attacks. Existing efforts for contract security analysis heavily rely on rigid rules defined by experts, which are *labor-intensive* and *non-scalable*. More importantly, expert-defined rules tend to be *error-prone* and suffer the inherent risk of being cheated by crafty attackers. Recent researches focus on the symbolic execution and formal analysis of smart contracts for vulnerability detection, yet to achieve a precise and scalable solution. Although several methods have been proposed to detect vulnerabilities in smart contracts, there is still a lack of effort that considers combining expert-defined security patterns with deep neural networks.

In this paper, we explore using graph neural networks and expert knowledge for smart contract vulnerability detection. Specifically, we cast the rich control- and data- flow semantics of the source code into a *contract graph*. To highlight the critical nodes in the graph, we further design a node elimination phase to normalize the graph. Then, we propose a novel temporal message propagation network to extract the graph feature from the normalized graph, and combine the graph feature with designed expert patterns to yield a final detection system. Extensive experiments are conducted on all the smart contracts that have source code in Ethereum and VNT Chain platforms. Empirical results show significant accuracy improvements over the state-of-the-art methods on three types of vulnerabilities, where the detection accuracy of our method reaches 89.15%, 89.02%, and 83.21% for reentrancy, timestamp dependence, and infinite loop vulnerabilities, respectively.

**Index Terms**—Deep learning, blockchain, smart contract, vulnerability detection, expert knowledge

## 1 Introduction

Blockchain and its killer applications, *e.g., Bitcoin and smart contract*, are taking the world by storm [1–6]. A blockchain is essentially a distributed and shared transaction ledger, maintained by all the miners in the blockchain network following a consensus protocol [7]. The consensus protocol and replicated ledgers enforce all the transactions immutable once recorded on the chain, endowing blockchain with decentralization and tamper-free nature.

**Smart contract.** Smart contracts are programs running on top of the blockchain [4, 8]. A smart contract can implement arbitrary rules for managing assets by encoding the rules into source code. The defined rules of a contract will be strictly and automatically followed during execution, effectuating the 'code is law' logic. Smart contracts make the automatic execution of contract terms possible, facilitating complex decentralized applications (DApps). Indeed, many DApps are basically composed of several smart contracts as the backend and a user interface as the frontend [9].

Millions of smart contracts have been deployed in various blockchain platforms, enabling a wide range of applications including wallets [10], crowdfunding, decentralized gambling [11], and cross-industry finance [12]. The number of smart contracts is still growing rapidly. For example, within the last six months, over 15,000 new contracts were deployed on Ethereum alone, which is the most famous smart contract platform.

**Security issues of smart contracts.** Smart contracts from various fields now hold more than 10 billion dollars worth of virtual coins. Undoubtedly, holding so much wealth makes smart contracts attractive enough to attackers. In June 2016, attackers exploited the reentrancy vulnerability of the DAO contract [13] to steal 3.6 million Ether, which was worth 60 million US Dollars. This case is not isolated and several security vulnerabilities are discovered and exploited every few months [13–15], undermining the trust for smart contract-based applications.

There are several reasons that make smart contracts particularly prone to errors. *First*, the programming languages (*e.g., Solidity*) and tools are still new and crude, leaving plenty of rooms for bugs and misunderstandings in the tools [8, 16]. *Second*, since smart contracts are immutable once deployed, developers are required to anticipate all possible status and environments the contract may encounter in the future, which is undoubtedly difficult. Distinct from conventional distributed applications that can be updated when bugs are detected, there is no way to patch the bugs of a smart contract without forking the blockchain (almost an impossible task), regardless of how much money the contract holds or how popular it is [8]. Therefore, effective vulnerability checkers for smart contracts before their de-

- *Zhenguang Liu, Peng Qian are with School of Computer and Information Engineering, Zhejiang Gongshang University and Zhejiang University, China. Email: liuzhenguang2008@gmail.com, messi.qp711@gmail.com*
- *Yuan Zhuang is with National University of Singapore, Singapore.*
- *Xiaoyang Wang is with School of Computer and Information Engineering, Zhejiang Gongshang University, China.*
- *Lin Qiu is with Southern University of Science and Technology, China.*
- *Xun Wang is with School of Computer and Information Engineering, Zhejiang Gongshang University and Zhejiang Lab, China. Email: xwang@zjgsu.edu.cn.*

*Corresponding authors: Peng Qian, Xun Wang*

ployment are essential.

**Drawbacks of conventional methods.** Conventional methods for smart contract vulnerability detection, *such as* [8, 16–18], employ classical static analysis or dynamic execution techniques to identify vulnerabilities. Unfortunately, they fundamentally rely on several expert-defined patterns. The manually defined patterns bear the inherent risk of being *error-prone* and some complex patterns are *non-trivial* to be covered. Crudely using several rigid patterns leads to high *false-positive* and *false-negative* rates, and crafty attackers may easily bypass the pattern checking using tricks. Moreover, as the number of smart contracts increases rapidly, it is becoming impossible for a few experts to sift through all the contracts to design precise patterns. A feasible solution might be: ask each expert to label a number of contracts, then collect all the labeled contracts from many experts to train a model that can automatically give a prediction on whether a contract has a specific type of vulnerability.

Recently, efforts have been made towards adopting deep neural networks for smart contract vulnerability detection [19–21], achieving improved accuracy. [19] utilizes LSTM based networks to sequentially process source code, while [20] models the source code into control flow graphs. [21] builds a sequential model to analyze the Ethereum operation code. However, these approaches either treat the source code or operation code as a text sequence instead of semantic blocks, or fail to highlight critical variables in the data flow [20], leading to insufficient semantic modeling and unsatisfactory results.

To fill the research gap, in this paper, we investigate more than 300,000 smart contract functions and present a fully automated and scalable approach that can detect vulnerabilities at the function level. Specifically, we cast the rich control- and data- flow semantics of the source code into graphs. The nodes in the graph represent critical variables and function invocations, while directed edges capture their temporal execution traces. Since not all nodes in the graph are of equal importance and most graph neural networks are inherently flat during information propagation on the graph, we design a node elimination phase to normalize the graph and highlight the key nodes. The normalized graph is then fed into a temporal message propagation network to learn the *graph* feature. In the meantime, we extract the *security pattern* feature from the source code using expert knowledge. Finally, the *graph* feature and *security pattern* feature are incorporated to produce the final vulnerability detection results.

We conducted experiments on all the 40k contracts that have source code in Ethereum and on all the contracts in VNT Chain, demonstrating significant improvements over state-of-the-art vulnerability detection methods: F1 score from 78% to 86%, 79% to 88%, 74% to 82% for *reentrancy, timestamp dependence*, and *infinite loop* vulnerabilities, respectively. Our implementations[1] are released to facilitate future research.

We would like to point out that this work is clearly distinct from the previous one [20] in three ways: 1) this work is to investigate whether combining graph neural networks with conventional expert patterns could achieve better vulnerability detection results, while the objective of the previous work is to explore the possibility of using neural networks for smart contract vulnerability detection. 2) In this work, we propose to extract vulnerability-specific expert patterns and combine them with the graph feature. We also explicitly model the key variables in the data flow. In contrast, in the previous work, we only utilize the graph feature while ignoring expert patterns and key variables. 3) This work consistently outperforms the previous one across different vulnerabilities, and overall provides more insights and findings in this field. Note that in the previous paper, we proposed two neural networks, *DR-GCN* and *TMP*, to explore the applicability of different graph neural networks on smart contract vulnerability detection. In this paper, we focus on extending TMP, which delivers better performance than *DR-GCN*. We will also extend *DR-GCN* and compare it with the extension of TMP.

**Contributions.** To summarize, the key contributions are:

- To the best of our knowledge, we are the first to investigate the idea of fusing conventional expert patterns and graph-neural-network extracted features for smart contract vulnerability detection.
- We propose to characterize the contract function source code as contract graphs. We also explicitly normalize the graph to highlight key variables and invocations. A novel temporal message propagation network is proposed to automatically capture semantic graph features.
- Our methods set the new state-of-the-art performance on smart contract vulnerability detection, and overall provide insights into the challenges and opportunities. As a side contribution, we have released our implementations to facilitate future research.

## 2 Related Work

### 2.1 Smart Contract Vulnerability Detection

Smart contract vulnerability detection is one of the fundamental problems in blockchain security. Early works on smart contract vulnerability detection verify smart contracts by employing formal methods [22–25]. For example, [22] introduces a framework, translating *Solidity* code (the smart contract programming language of Ethereum) and the EVM (Ethereum Virtual Machine) bytecode into the input of an existing verification system. [25] proposes a formal model for EVM and reasons the potential bugs in smart contracts by using the Isabelle/HOL tool. Further, [23] and [24] define formal semantics of the EVM using the F* framework and the $\mathbb{K}$ framework, respectively. Although these frameworks provide strong formal verification guarantees, they are still semi-automated.

Another stream of work relies on generic testing and symbolic execution, such as Oyente [8], Maian [26], and Securify [18]. Oyente is one of the pioneering works that perform symbolic execution on contract functions and flags bugs based on simple patterns. Zeus [27] leverages abstract interpretation and symbolic model checking, as well as the constrained horn clauses to detect vulnerabilities in smart contracts. [18] introduces compliance (negative) and violation (positive) patterns to filter false warnings.

Researchers also explore smart contract vulnerability detection using dynamic execution. [17] presents ContractFuzzer to identify vulnerabilities by fuzzing and runtime

1. Github: https://github.com/Messi-Q/GPSCVulDetector

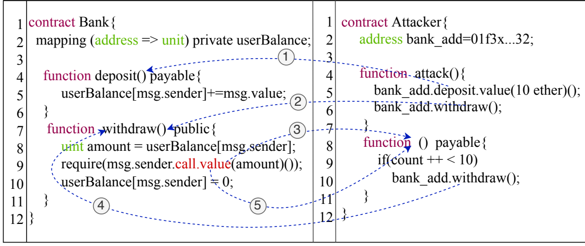



**Fig. 1 A simplified example of reentrancy vulnerability.**

behavior monitoring during execution. Similarly, [28] develops a fuzzing-based analyzer to identify the reentrancy vulnerability. Sereum [29] uses taint analysis to monitor runtime data flows during smart contract execution for vulnerability detection. However, dynamic execution methods require a hand-crafted agent contract to interact with the contract under test, preventing them from fully-automated applications and endowing them non-scalability.

Recently, a few attempts have been made to study using deep neural networks for smart contract vulnerability detection. [19] constructs the sequential *contract snippet* and feeds them into the BLSTM-ATT model to detect reentrancy bugs. [20] proposes to convert the source code of a contract into the *contract graph* and constructs graph neural networks as the detection model. [30] proposes ContractWard, extracting bigram features from the operation code of smart contracts and utilizing machine learning algorithms. However, although a few methods have been proposed, the field of contract vulnerability detection using deep learning is still in its infancy and the accuracy is still unsatisfactory. For common smart contract vulnerabilities and attacks, motivated readers may refer to [31] for a comprehensive survey.

### 2.2 Graph Neural Network

With the remarkable success of neural networks, graph neural network has been investigated extensively in various fields such as graph classification [32, 33], program analysis [34, 35], and graph embedding [36]. Existing approaches roughly cast into two categories: **(i)** *Spectral-based approaches* generalize well-established neural networks like CNNs to work on graph-structured data. For instance, GCN [37] implements a first-order approximation of spectral graph convolutions [38–40] and develops a layer-wise propagation network using the Laplacian matrix, which achieves promising performance on graph node classification tasks. [41] proposes a graph CNN which can take data of arbitrary graph structure as input. **(ii)** *Spatial-based methods* inherit ideas from recurrent GNNs and adopt information propagation to define graph convolutions. Early work such as [42] directly sums up the nodes' neighborhood information for graph convolutions. Another line of work, such as GAT [43] and GAAN [44], employs attention mechanisms to learn the weights of different neighboring nodes. Motivated by these spatial-based approaches, [45] outlines a message-passing neural network framework to predict the chemical properties of molecules.

Recently, [34, 35, 46, 47] attempt to apply GNNs to program analysis issues. Specifically, [35] introduces a gated graph recurrent network for variable prediction, while [46] proposes Gemini for binary code similarity detection, where functions in binary code are represented by attributed control flow graphs. [34] develops Devign, a general graph neural network-based model for vulnerability identification in C programming language. Different from these methods, we focus on the specific smart contract vulnerability task, and explicitly take into account the distinct roles and temporal relationships of program elements.

## 3 Problem Statement

In this section, we first formulate the problem, then introduce the three types of vulnerabilities studied in this paper, and present the reasons for focusing on these three vulnerabilities.

**Problem formulation.** Given the source code of a smart contract, we are interested in developing a fully automated approach that can detect vulnerabilities at the function level. In other words, we are to estimate the label $\hat{y}$ for each smart contract function $f$, where $\hat{y} = 1$ represents $f$ has a specific vulnerability while $\hat{y} = 0$ denotes $f$ is safe. In this paper, we focus on three types of vulnerabilities, which will be presented below. Before that, we first introduce the preliminary knowledge of the fallback mechanism in smart contracts, which is important in understanding the problem.

**Fallback mechanism.** Within a smart contract, each function is uniquely identified by a signature, consisting of its name and parameter types [31]. Upon a function invocation, the signature of the invoked function is passed to the called contract. If the signature matches a function of the called contract, the execution jumps to the corresponding function. Otherwise, it jumps to the fallback function. Money transfer is considered as an empty signature, which will trigger the fallback function as well. The fallback function is a special function with no name and no argument, which can be arbitrarily programmed [31]. After introducing this background knowledge, we now are ready to elaborate on the three types of vulnerabilities.

(1) **Reentrancy** is a well-known vulnerability that caused the infamous DAO attack. When a smart contract function $f_1$ transfers money to a recipient contract $C$, the fallback function $f_2$ of $C$ will be automatically executed. In its fallback function $f_2$, $C$ may invoke back to $f_1$ for conducting an invalid second-time transfer. Since the current execution of $f_1$ waits for the first-time transfer to finish, $C$ can make use of the intermediate state of $f_1$ to succeed in stealing money. A simplified example is shown in Fig. 1, where the *withdraw* function of contract *Bank* has a reentrancy vulnerability, contract *Attacker* steals money by exploiting the vulnerability. First, *Attacker* deposits 10 Ether (Ether is the virtual money of Ethereum) in contract *Bank* (step 1). Then, *Attacker* withdraws the 10 Ether by invoking the *withdraw* function (step 2). When the contract *Bank* sends 10 Ether to *Attacker* using ***call.value*** (*Bank*, line 9), the fallback function (*Attacker*, lines 8–11) of *Attacker* will be automatically invoked (step 3). In its fallback function, *Attacker* calls ***withdraw*** again (step 4). Since the *userBalance* of *Attacker* has not yet been set to 0 (*Bank*, line 10), *Bank* believes that *Attacker* still has 10 Ether in the contract, thus transfers 10 Ether to *Attacker* again (Step 5). The withdraw loop lasts for 9 times ($count++ < 10$, *Attacker* line 9). Finally, *Attacker* obtains much more Ether (100 Ether) than expected (10 Ether).

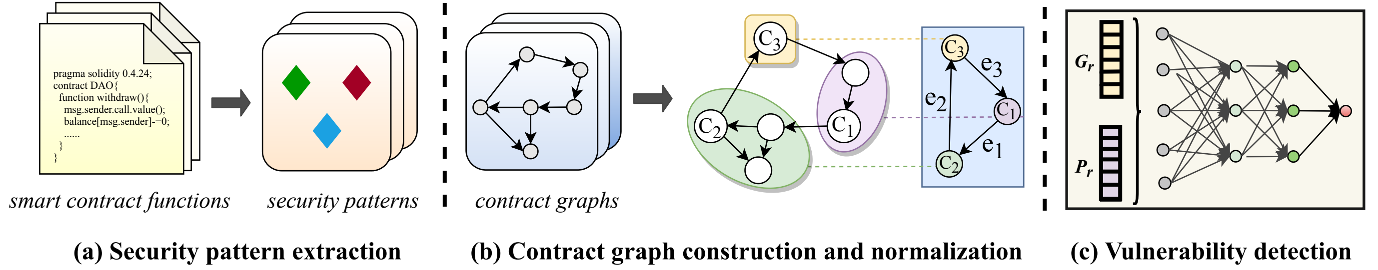



**Fig. 2 The overall architecture of our proposed method. (a) The expert pattern extraction phase; (b) the contract graph construction and normalization phase; (c) the vulnerability detection phase.**

(2) **Timestamp dependence** vulnerability exists when a smart contract uses the block timestamp as a triggering condition to execute some critical operations, e.g., using the *timestamp* of a future block as the source to generate random numbers so as to determine the winner of a game. The miner (a node in the blockchain) who mines the block has the freedom to set the timestamp of the block within a short time interval ($<$ 900 seconds) [17]. Therefore, miners may manipulate the block timestamps to gain illegal benefits.

(3) **Infinite loop** is a common vulnerability in smart contracts. The program of a function may contain a loop (e.g. *for* loop, *while* loop, and self-invocation loop) with no exit condition or the exit condition cannot be reached, namely an infinite loop.

**Why focus on these vulnerabilities.** We mainly focus on the three aforementioned vulnerabilities since: **(i)** In real attacks, blockchain networks have suffered more than 100 million USD losses due to the three vulnerabilities. For instance, attacks on reentrancy have caused one of the biggest losses (60 million USD in The Dao Event) in smart contract history. **(ii)** We empirically found that the three vulnerabilities may affect a significant number of smart contracts and are non-trivial to be detected. Specifically, we surveyed 40,932 Ethereum smart contracts, observing that around 5,013 out of 307,396 functions possess at least one invocation to *call.value*. Although possessing a *call.value* invocation does not necessarily mean that the contract has a reentrancy vulnerability, the contract has the potential to be affected by the *reentrancy* vulnerability and thus requires further checking. Similarly, around 4,833 functions have used *block.timestamp* and thus are potentially affected by the *timestamp dependence* vulnerability. Many functions have *for* or *while* loops, which may lead to the *infinite loop* vulnerability. In contrast, most other contract vulnerabilities affect a relatively smaller number of functions, *e.g.*, the *locked ether* vulnerability affects less than 900 functions, and the *integer overflow* vulnerability affects less than 1,000 functions.

## 4 Our Method

**Method overview.** The overall architecture of our proposed method is depicted in Fig. 2, which consists of three phases: (1) a security pattern extraction phase, which obtains the vulnerability-specific expert patterns from the source code; (2) a contract graph construction and normalization phase, which extracts the control flow and data flow semantics from the source code and highlights the critical nodes; and (3) a vulnerability detection phase, which casts the normalized contract graph into graph feature using temporal graph neural network, and combines the pattern feature and graph feature to output the detection result. In what follows, we elaborate on the details of the three components one by one.

### 4.1 Expert Pattern Extraction

In this section, we summarize existing patterns and design new patterns for the three specific vulnerabilities respectively, and implement an open-sourced tool to automatically extract these patterns.

**Reentrancy.** Conventionally, the reentrancy vulnerability is considered as an invocation to *call.value* that can call back to itself through a chain of calls. That is, the invocation of *call.value* is successfully re-entered to perform the unexpected operation of repeated money transfer. By investigating existing works such as [8, 17, 27], we design three sub-patterns. The first sub-pattern is **callValueInvocation** that checks whether there exists an invocation to *call.value* in the function. The second sub-pattern **balanceDeduction** checks whether the user balance is deducted *after* money transfer using *call.value*, which considers the fact that the money stealing can be avoided if user balance is deducted each time *before* money transfer. The third sub-pattern **enoughBalance** concerns whether there is a check on the sufficiency of the user balance before transferring to a user. Note that *enoughBalance* is a new pattern designed in this paper.

**Timestamp dependence.** Generally, the timestamp dependence vulnerability exists when a smart contract uses the block timestamp as part of the conditions to perform critical operations [17]. By investigating previous works including [8, 17, 31], we design three sub-patterns that are closely related to timestamp dependence. *First,* sub-pattern **timestampInvocation** models whether there exists an invocation to opcode *block.timestamp* in the function. *Then*, the second sub-pattern **timestampAssign** checks whether the value of *block.timestamp* is assigned to other variables or passed to a function as a parameter, namely whether *block.timestamp* is actually used. *Last*, the third sub-pattern **timestampContamination** checks if *block.timestamp* may contaminate the triggering condition of a critical operation, which can be implemented by taint analysis. Sub-pattern *timestampContamination* is a new pattern designed in this paper.

**Infinite loop.** Infinite loop is conventionally considered as a loop bug which unintentionally iterates forever, failing to jump out of the loop and return an expected result. Specifically, we define three expert patterns for infinite loop as follows. (1) The first sub-pattern **loopStatement** checks whether the function possesses a loop statement such as *for* and *while*. (2) The second sub-pattern **loopCondition** models whether the exit condition can be reached. For example,

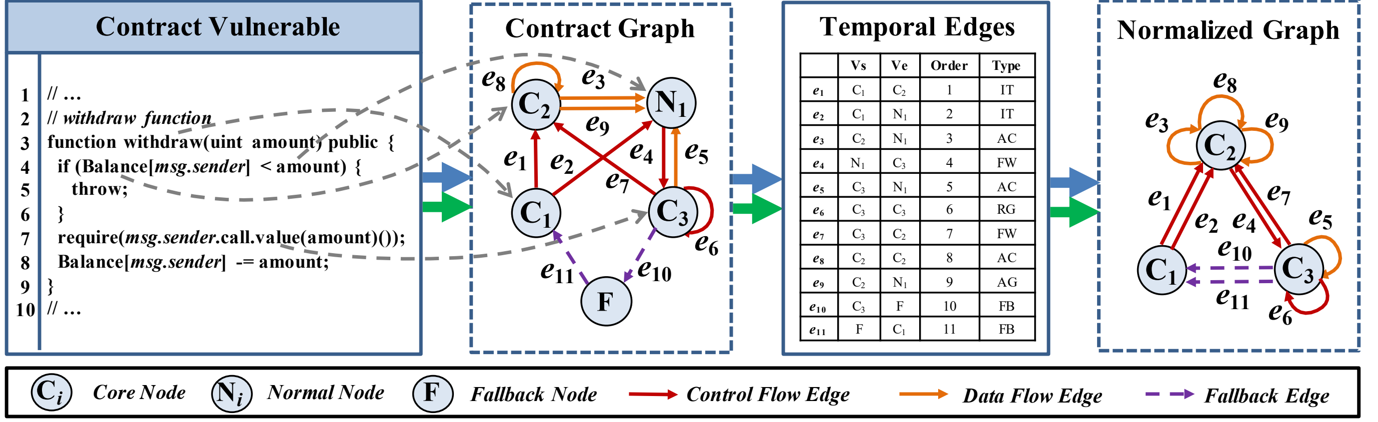



**Fig. 3 The contract graph construction and normalization phase. The first figure shows the source code of a contract function, while the second figure visualizes the contract graph extracted from the code. Nodes $\mathrm{C_i}$ denote core nodes, nodes $\mathrm{N_i}$ represent normal nodes, and node F denotes fallback node. The third figure illustrates the temporal edges in the extracted graph, where the types of edges are detailed in Table 1. The fourth figure demonstrates the graph after normalization.**

for a *while* loop, its exit condition $i < 10$ may not be reached if $i$ is never updated in the loop. (3) The third sub-pattern **selfInvocation** models whether the function invokes itself and the invocation is not in an *if* statement. This concerns the fact that if the self-invocation statement is not in an *if* statement, the self-invocation loop will never terminate.

**Pattern Extraction Implementations.** We implemented an open-sourced tool to extract the designed expert patterns from smart contract functions. Particularly, simple sub-patterns such as *callValueInvocation*, *timestampInvocation*, and *loopStatement* can be directly extracted by keyword matching. Sub-patterns *balanceDeduction*, *enoughBalance*, *loopCondition*, *timestampAssign*, and *selfInvocation* are obtained by syntax analysis. Complex sub-pattern *timestampContamination* is extracted by taint analysis where we follow the traces of the data flow and flag all the variables that may be affected along the traces.

## 4.2 Contract Graph Construction and Normalization

Existing works [35, 48] have shown that programs can be transformed into symbolic graph representations, which are able to preserve semantic relationships (e.g., data dependency and control dependency) between program elements. Inspired by this, we formulate smart contract functions into *contract graphs*, and assign distinct roles to different program elements (namely nodes). We also construct edges to model control and data flow between program elements, taking their temporal orders into consideration. Further, we design a node elimination process to normalize the *contract graph* and highlight important nodes. Next, we introduce contract graph construction and normalization, respectively.

### 4.2.1 Contract Graph Construction

**Nodes construction.** Our first insight is that different program elements in a function are not of equal importance in detecting vulnerabilities. Therefore, we extract three types of nodes, *i.e.*, *core nodes*, *normal nodes*, and *fallback nodes*.

*Core nodes.* Core nodes symbolize the key invocations and variables that are critical for detecting a specific vulnerability. In particular, for reentrancy vulnerability, core nodes model (i) an invocation to a money transfer function or the built-in *call.value* function, (ii) the variable that corresponds to *user balance*, and (iii) variables that can directly affect *user balance*. For timestamp dependence vulnerability, (i) invocations to *block.timestamp*, (ii) variables assigned by *block.timestamp*, and (iii) invocations to a random function that takes *block.timestamp* as the cardinal seed are extracted as core nodes. For infinite loop vulnerability, (i) all the loop statements such as *for* and *while* statements, (ii) the loop condition variables, and (iii) self invocations are considered as core nodes.

*Normal nodes.* While core nodes represent key invocations and variables, normal nodes are used to model invocations and variables that play an auxiliary role in detecting vulnerabilities. Specifically, invocations and variables that are not extracted as core nodes are modeled as normal ones, e.g., for timestamp dependence vulnerability, invocations that do not call *block.timestamp* and variables indirectly related to *block.timestamp* are considered as normal nodes.

*Fallback node.* Further, we construct a fallback node $F$ to stimulate the fallback function of a virtual attack contract, which can interact with the function under test.

*A simplified example.* Taking contract *Vulnerable* presented in the left of Fig. 3 as an example, suppose we are to evaluate whether its *withdraw* function possesses a reentrancy vulnerability. As shown by the arrows in the left two figures of Fig. 3, function *withdraw* itself is first modeled as a core node $C_1$ since its inner code contains *call.value*. Then, following the temporal order of the code, we treat the critical variable $Balance[msg.sender]$ as a core node $C_2$, while variable $amount$ is modeled as normal node $N_1$. The invocation to *call.value* is extracted as a core node $C_3$, and the *fallback* function of a virtual attack contract is characterized by the fallback node $F$.

**Edges construction.** Our second insight is that the nodes are closely related to each other in a temporal manner rather than being isolated. To capture rich semantic dependencies between the nodes, we construct three categories of edges, namely *control flow, data flow*, and *fallback* edges. Each edge describes a path that might be traversed through by the function under test, and the temporal number of the edge characterizes its sequential order in the function. We investi-

| Type (Abbreviation) | 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 |

**TABLE 1 Semantic edges summarization. All edges are classified into three categories, namely control-flow, data-flow, and fallback edges.**

gated various functions and summarized the semantic edges in Table 1. All edges are classified into three categories.

*Control flow edges.* Control flow edges capture the control semantics of the code. Specifically, a control flow edge is constructed for a *conditional* statement or *security handle* statement, such as a *if, for, assert*, and *require* statement. The edge directs from the previous node encountered, which represents the critical function call or variable preceding to the current statement, to the node representing the function call or variable in the current statement. In particular, we use forward edges to describe the natural control flow of the code sequence. A forward edge connects two nodes in the adjacent statements. The main benefit of such encoding is to reserve the programming logic reflected by the sequence of the source code. The control flow edges are depicted with red arrows in Fig. 3.

*Data flow edges.* Data flow edges track the usage of variables. A data flow edge involves the access or modification of a variable. The data flow edges are demonstrated with orange arrows in Fig. 3. For example, the *access* and *assign* statement $Balance[msg.sender]$-=$amount$ (line 8, *Vulnerable*, Fig. 3) is characterized by two data flow edges, i.e., an access edge $e_7$ starting from the $Balance[msg.sender]$ variable node $C_2$ to itself, and an assign edge $e_8$ starting from $C_2$ to the *amount* variable node $N_1$.

*Fallback edges.* In order to explicitly model the specific fallback mechanism, two fallback edges are constructed. The first fallback edge connects from the first *call.value* invocation to the fallback node, while the second edge directs from the fallback node to the function under test. The fallback edges are shown by dashed purple edges in Fig. 3.

**Node and edge features.** Fig. 4 illustrates the extracted features for edges and nodes, respectively. Specifically, the feature of an edge is extracted as a tuple ($V_{start}$, $V_{end}$, *Order*, *Type*), where $V_{start}$ and $V_{end}$ represent its start and end nodes, *Order* denotes its temporal order, and *Type* stands for edge type. For nodes, different kinds of nodes possess different features. 1) The feature of a node that models function invocation consists of (*ID*, *AccFlag*, *Caller*, *Type*), where *ID* denotes its identifier, *Caller* represents the caller address of the invocation, and $Type$ stands for the node type. Interestingly, the modifier of a smart contract function $\Psi$ may trigger the pre-check of certain conditions, e.g., modifier *owner* will check whether the caller of $\Psi$ is the owner of the contract before executing $\Psi$. Therefore, we use *AccFlag* to capture this semantics, where *AccFlag* = 'LimitedACC' specifies the function has limited access while AccFlag ='NoLimited' denotes non-limited access. 2) In contrast, the feature of a fallback node or a node that

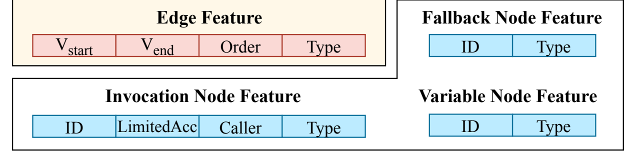



**Fig. 4 Illustration of the edge feature and node feature.**

models variable consists of only *ID* and *Type*.

### 4.2.2 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 functions yield distinct graphs, hindering the training of graph neural networks. Therefore, we propose a node elimination process to normalize the *contract graph*.

**Nodes elimination.** As introduced in Section 4.2.1, the nodes of a *contract graph* are partitioned into core nodes $\{C_i\}_{i=1}^{|C|}$, normal nodes $\{N_i\}_{i=1}^{|N|}$, and the fallback node $F$. We remove each normal node $N_i$, but pass the feature of $N_i$ to its nearest core nodes. For example, the normal node $N_1$ in the second figure of Fig. 3 is removed with its feature aggregated to nearest core nodes $C_2$ and $C_3$. For a node $N_i$ that has multiple nearest core nodes, its feature is passed to all of them. The edges connected to the removed normal nodes are preserved but with their start or end node moving to the corresponding core node. The fallback node is also removed similar to the normal node.

**Feature aggregation.** After removing normal nodes, features of core nodes are updated by aggregating features from their neighboring normal nodes. More precisely, the new feature of $C_i$ is composed of three components: **(i)** self-feature, namely the feature of core node $C_i$ itself; **(ii)** in-features, namely features of the normal nodes $\{P_j\}_{j=1}^{|P|}$ that are merged to $C_i$ and having a path pointing from $P_j$ to $C_i$; and **(iii)** out-feature, namely features of the normal nodes $\{Q_k\}_{k=1}^{|Q|}$ that are merged to $C_i$ and having a path directs from $Q_k$ to $C_i$. Note that features of different normal nodes that model variables and invocations are added respectively when aggregating to the same node.

## 4.3 Vulnerability Detection

In this subsection, we introduce the proposed vulnerability detection network CGE (Combining Graph feature and Expert patterns). First, we obtain the expert pattern feature $P_r$ by passing the extracted sub-patterns (introduced in subsection 4.1) into a feed-forward neural network (FNN). Then, we extract the graph feature $G_r$ from the normalized *contract graph* by our proposed temporal message propagation network, consisting of a *message propagation* phase and a *readout* phase. Finally, we use a fusion network to combine the graph feature $G_r$ and the pattern feature $P_r$, outputting the detection results. The process is demonstrated in Fig. 5 with details presented below.

**Security pattern feature $P_r$ extraction.** For the sub-patterns closely related to a specific vulnerability, we utilize a one-hot vector to represent each sub-pattern, and append a 0/1 digit to each vector, which indicates whether the function under test has the sub-pattern. The vectors for all sub-patterns related to a specific vulnerability are concatenated

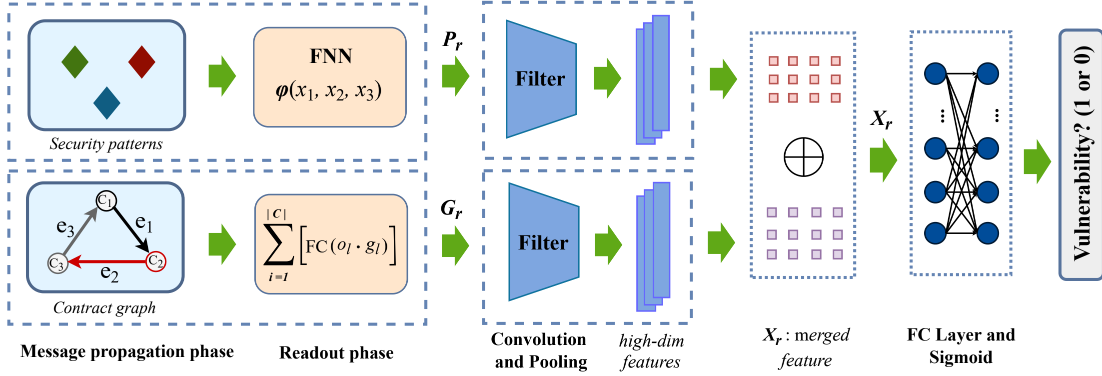



**Fig. 5 The process of vulnerability detection. First, a feed-forward neural network generates the pattern feature $P_r$ for the security patterns extracted from the source code. Then, the temporal message propagation network is used to extract the graph feature $G_r$ from the contract graph. Finally, the CGE network combines $G_r$ and $P_r$ into the merged feature $X_r$, which is fed into the FC and sigmoid layers to output the vulnerability detection results.**

into a final vector $x$. Taking $x$ as the input, and the ground truth of whether the function has the specific vulnerability as the target label, we utilize a feed-forward neural network $\varphi(x)$ to extract high-dimensional semantic feature $P_r \in \mathbb{R}^d$.

**Contract graph feature $G_r$ Extraction**. After extracting security pattern feature $P_r$, we further obtain the semantic feature of the contract graph by using our proposed temporal-message-propagation network, which consists of a *message propagation* phase and a *readout* phase. In the message propagation phase, the network passes information along the edges successively by following their temporal orders. Then, it generates the graph feature $G_r$ by using a readout function, which aggregates the final states of all nodes in the contract graph.

*Message propagation phase.* Formally, we denote the normalized contract graph as $G = \{V, E\}$, where $V$ consists of the core nodes, and $E$ consists of all edges. Denote $E = \{e_1, e_2, \dots, e_N\}$, where $e_k$ represents the $k^{th}$ temporal edge.

Messages are passed along the edges, one edge per time step. At first, the hidden state $h_i^0$ for each node $V_i$ is initialized with its own node feature. Then, at time step $k$, message flows through the $k^{th}$ temporal edge $e_k$ and updates the hidden state $h_{ek}$ of the end node of $e_k$.

More specifically, message $m_k$ is first computed basing on the hidden state $h_{sk}$ of the start node of $e_k$, and the edge type $t_k$:

$$x_k = h_{sk} \oplus t_k \tag{1}$$
$$m_k = W_k x_k + b_k \tag{2}$$

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

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:

$$\hat{h}_{ek} = tanh(Um_k + Zh_{ek} + b_1) \tag{3}$$
$$h'_{ek} = softmax(R\hat{h}_{ek} + b_2) \tag{4}$$

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

*Readout phase.* After successively traversing all the edges in $G$, we extract the feature 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 find 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 consider to generate the graph feature $G_r$ by

$$s_i = h_i^T \oplus h_i^0 \tag{5}$$
$$g_i = softmax(W_g^{(2)}(tanh(b_g^{(1)} + W_g^{(1)} s_i)) + b_g^{(2)}) \tag{6}$$
$$o_i = softmax(W_o^{(2)}(tanh(b_o^{(1)} + W_o^{(1)} s_i)) + b_o^{(2)}) \tag{7}$$
$$G_r = FC(\sum_{i=1}^{|V|} o_i \odot g_i) \tag{8}$$

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

**Vulnerability detection by combining $P_r$ and $G_r$.** After obtaining the security pattern feature $P_r$ and the contract graph feature $G_r$, we combine them to compute the final label $\hat{y} \in (0, 1)$, indicating whether the function under test has the specific vulnerability. To this end, we first filter $P_r$ and $G_r$ using a convolution layer and a max pooling layer, then we concatenate the filtered features and pass them to a network consisting of 3 fully connected layers and a sigmoid layer. The process can be formulated as:

$$X_r = \psi(P_r) \oplus \psi(G_r) \tag{9}$$
$$\hat{y} = sigmoid(FC(X_r)) \tag{10}$$

The convolutional layer learns to assign different weights to different elements of the semantic vector, while the max pooling layer highlights the significant elements and avoids overfitting. The fully connected layer and the non-linear sigmoid layer produce the final estimated label $\hat{y}$.

## 5 EXPERIMENTS

In this section, we empirically evaluate our proposed methods on all the Ethereum smart contracts that have source code verified by Etherscan [49], as well as on all the available

smart contracts on another blockchain platform VNT Chain [50]. We seek to answer the following research questions:

- **RQ1**: Can the proposed method effectively detect the reentrancy, infinite loop, and timestamp dependence vulnerabilities? How are its *accuracy*, *precision*, *recall*, and *F1 score* performance against the state-of-the-art conventional vulnerability detection approaches?
- **RQ2**: Is our proposed method useful for detecting new types of vulnerabilities, e.g., sharing-variable reentrancy, which is difficult for existing methods?
- **RQ3**: Can the proposed method outperform other neural network-based methods?
- **RQ4**: How do the proposed *security pattern*, *graph normalization*, *message propagation* modules, and *different network layers* in *CGE* affect the performance of the proposed method?

Next, we first present the experimental settings, followed by answering the above research questions one by one.

### 5.1 Experimental Settings

**Datasets.** We conducted experiments on two real-world smart contract datasets, namely `ESC` (Ethereum Smart Contracts) and `VSC` (VNT chain Smart Contracts), which are collected from Ethereum and VNT Chain platforms, respectively. Experiments for reentrancy and timestamp dependence vulnerabilities are conducted on `ESC`, while the infinite loop vulnerability is evaluated on `VSC`.

- The `ESC` dataset consists of 307,396 smart contract functions from 40,932 smart contracts in Ethereum [51]. 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. Around 56,800 functions contain *for* or *while* loop statements.
- The `VSC` dataset contains $13,761$ functions, which are collected from all the available $4,170$ smart contracts in the VNT Chain network [50]. VNT Chain is an experimental public blockchain platform proposed by companies and universities from Singapore, China, and Australia. The VNT Chain runs smart contracts written in Class C language.

**Implementation details.** All the experiments are conducted on a computer equipped with an Intel Core i7 CPU at 3.7GHz, a GPU at 1080Ti, and 32GB of Memory. Our vulnerability detection system consists of three main components: the auto *CodeExtractor* tool for extracting the security patterns and contract graphs from the source code; the *Normalization* tool for normalizing contract graphs; the CGE network that outputs results by combining pattern feature and graph feature. The *CodeExtractor* and *Normalization* tools are implemented with Python, while the CGE network is implemented with TensorFlow. The implementations of our vulnerability detection system are available at https://github.com/Messi-Q/GPSCVulDetector.

**Parameter settings.** The adam optimizer is employed in the CGE network. We apply a grid search to find out the best settings of hyper-parameters: the learning rate $l$ is tuned amongst {0.0001, 0.0005, 0.001, 0.002, 0.005, 0.01}, the dropout rate $d$ is searched in {0.1, 0.2, 0.3, 0.4, 0.5}, and batch size $\beta$ in {8, 16, 32, 64, 128}. To prevent overfitting, we tuned the L2 regularization $\lambda$ in {$10^{-6}$, $10^{-5}$, $10^{-4}$, $10^{-3}$, $10^{-2}$, $10^{-1}$}. Without special mention in texts, we report the performance of all neural network models with following default setting: 1) $l = 0.002$, 2) $d = 0.2$, 3) $\beta = 32$, and 4) $\lambda = 10^{-4}$. For each dataset, we randomly select 80% of them as the training set and the other 20% as the testing set for several times, and report the averaged result. The ground truth labels for contract functions are provided by experts.

### 5.2 Comparison with State-of-the-art Existing Methods (RQ1)

In this section, we benchmark our proposed method against existing non-deep-learning vulnerability detection approaches, which include:

- **Oyente** [8]: A well-known symbolic verification tool for smart contract vulnerability detection, which performs symbolic execution on the CFG (control flow graph) to check vulnerable patterns.
- **Mythril** [52]: A security analysis method, which uses concolic analysis, taint analysis, and control flow checking to detect smart contract vulnerabilities.
- **Smartcheck** [16]: An extensible static analysis tool for discovering smart contract code vulnerabilities.
- **Securify** [18]: A formal-verification based tool for detecting Ethereum smart contract bugs, which checks compliance and violation patterns to filter false positives.
- **Slither** [53]: A static analysis framework designed to find issues in Ethereum smart contracts by converting a smart contract into an intermediate representation of *SlithIR*.

**Comparison on reentrancy vulnerability detection.** First, we compare our CGE approach with the five existing methods on the reentrancy vulnerability detection task. The performance of different methods is presented in the left of Table 2, where metrics of *accuracy*, *recall*, *precision*, and *F1 score* are engaged. We would like to highlight that all metrics are computed over only the susceptible smart contract functions having invocation(s) to *call.value*, i.e., the functions that may be infected with the reentrancy vulnerability. Functions with no *call.value* invocation are known to be immune to reentrancy vulnerability and is trivial to be handled (using purely keyword matching), thus we do not involve those functions in the calculation to better investigate the problem. From the quantitative results of Table 2, we have the following observations. First, we find that conventional non-deep-learning methods have not yet achieved a satisfactory accuracy on the reentrancy vulnerability detection task, e.g., the state-of-the-art method (i.e., Slither) yields a 77.12% accuracy. Second, our proposed method substantially outperforms the existing methods on reentrancy vulnerability detection. Specifically, CGE achieves a 89.15% accuracy, gaining a 12.03% accuracy improvement over conventional methods. The strong empirical evidences suggest the great potential of combing graph neural networks with expert patterns for reentrancy vulnerability detection.

By looking into the existing methods, we believe that the reasons for the low precision and recall of conventional methods are: (1) they heavily rely on simple and fixed patterns to detect vulnerabilities, *e.g., Mythril checks whether the call.value invocation is not followed by any internal function*

| 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** | **83.21** | **82.29** | **81.97** | **82.13** |

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

*call to detect reentrancy*, and (2) the rich data dependencies and control dependencies within smart contract code are not characterized with fine-grained details in these methods.

**Comparison on timestamp dependence vulnerability detection.** We further compare the proposed CGE with the five methods on the timestamp dependence vulnerability detection task. The comparison results are demonstrated in the middle of Table 2. The state-of-the-art conventional method (i.e., Slither) has obtained a 74.20% 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, CGE keeps delivering the best performance in terms of all the four metrics. In particular, CGE gains a 14.82% accuracy improvement over state-of-the-art conventional methods.

**Comparison on infinite loop vulnerability detection.** We also evaluated our methods on the infinite loop vulnerability. Specifically, we compare our methods against available infinite loop detection methods including:

- **Jolt** [54]: The tool detects infinite loop bugs by monitoring the program state of two consecutive loop iterations.
- **SMT** [55]: An algorithm that relies on satisfiability modulo theories for automated detection of infinite loop bugs.
- **PDA** [56]: A method that performs program path-based checking for infinite loop detection.
- **Looper** [57]: Loop detection based on symbolic execution.

Quantitative results are illustrated in the right of Table 2. From the table, we see that CGE consistently and significantly outperforms other methods on the infinite loop vulnerability detection task. In particular, CGE achieves a 83.21% accuracy and a 82.13% F1 score. In contrast, state-of-the-art detection tools Looper are 59.56% and 53.87%, and TMP are 74.61% and 74.10%. The improvements may come from the fact that we consider key variables and rich dependencies between program elements in smart contracts.

We further visualize the quantitative results of Table 2 in Figs. 6(a), (b), and (c). Specifically, Fig. 6(a) and Fig. 6(b) present comparison results of reentrancy vulnerability detection and timestamp dependence vulnerability detection, respectively. The 7 rows (in different colors) from front to back denote methods *Smartcheck*, *Oyente*, *Mythril*, *Securify*, *Slither*, TMP, and CGE, respectively. For each row in the figures, *accuracy*, *recall*, *precision*, and *F1 score* are respectively demonstrated from left to right. Fig. 6(c) shows comparison results of infinite loop vulnerability detection, where the 6 rows from front to back denote *Jolt*, *PDA*, *SMT*, *Looper*, TMP, and CGE methods, respectively. We can clearly observe that CGE outperforms existing methods by a large margin.

### 5.3 A Case Study Towards Better Understanding of the Reasons Behind the Results (RQ2)

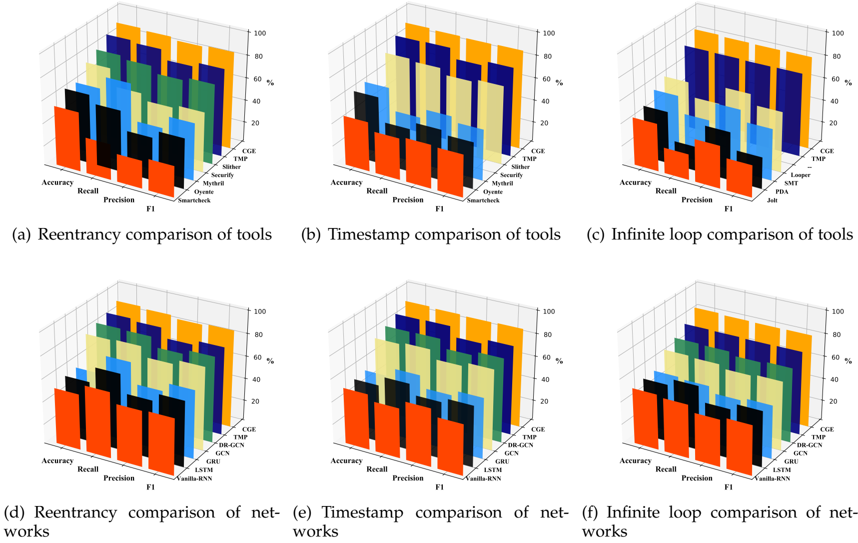
In this subsection, we present an interesting case of smart contract vulnerabilities, which may bring new insights into the abilities of the studied methods. Particularly, we investigate a new type of reentrancy vulnerability, *i.e., sharing-variable* reentrancy. To our knowledge, most existing methods cannot precisely detect such vulnerabilities.

Besides classical reentrancy introduced in Fig. 1 and section 3, a reentrancy attack is also possible when a transfer function shares internal variables with another function, which we define as sharing-variable reentrancy.

In Fig. 7, we illustrate a real-world sharing-variable reentrancy example, where the *Malicious* contract plays an attack role against the *Vulnerable* contract. More specifically, contract *Vulnerable* contains two functions: *getBonusWithdraw* and *withdrawAll*. Function *withdrawAll* allows a user to withdraw all her rewards, while function *getBonusWithdraw* allows a user to withdraw all her rewards together with a 0.1 Ether bonus for each new user.

**Attack.** As demonstrated in Fig. 7, contract *Malicious* first uses its *attack* function to call the *getBonusWithdraw* function of contract *Vulnerable* (step 1). As *getBonusWithdraw* invokes the *withdrawAll* function (*Vulnerable*, line 6) to send the rewards and bonus to *Malicious* (step 2). This will automatically trigger the fallback function of *Malicious* (step 3), where *Malicious* invokes *getBonusWithdraw* again to steal money (step 4). Since the bonus flag *Bonus*[*msg.sender*] has yet been set to true, *Vulnerable* believes *Malicious* has not got the new user bonus yet and thus gives 0.1 Ether bonus again to *Vulnerable* (*Vulnerable*, line 5), then function *withdrawAll* is re-entered to withdraw the 0.1 Ether illegal bonus (step 5). *Malicious* actually invokes *getBonusWithdraw* 9 times (*Malicious*, line 9) in its fallback function to steal 1 Ether.

**Underlying issue.** This example reveals that although in the *withdrawAll* function, contract *Vulnerable* updates the user balance (i.e., *Reward*) before money transfer, *Malicious* can still be attacked. The novel attack utilizes the shared variable (*Reward*) to steal money. Although *withdrawAll* function itself is safe, the malicious contract may




(a) Reentrancy comparison of tools (b) Timestamp comparison of tools (c) Infinite loop comparison of tools

(d) Reentrancy comparison of networks (e) Timestamp comparison of networks (f) Infinite loop comparison of networks

**Fig. 6 Visuallization of the quantitative results in Table 2: (a) & (d) present comparison results of reentrancy vulnerability detection, while (b) & (e) present comparison results of timestamp dependence detection, (c) & (f) show comparison results of infinite loop vulnerability detection. In (a) & (b), the 7 rows from front to back denote the Smartcheck, Oyente, Mythril, Securify, Slither, TMP, and CGE methods, respectively. In (c), the 6 rows from front to back denote the Jolt, PDA, SMT, Looper, TMP, and CGE methods, respectively. In (d) & (e) & (f), the 7 rows from front to back denote the Vanilla-RNN, LSTM, GRU, GCN, DR-GCN, TMP, and CGE methods, respectively. For each row in the figures, *accuracy*, *recall*, *precision*, and *F1 score* are respectively demonstrated from left to right.**

call *getBonusWithdraw* to modify the shared variable *Reward* to enable attacks.

Unfortunately, such kind of attacks cannot yet be detected by existing methods. We empirically checked the *Vulnerable* contract using the state-of-the-art tools including *Oyente [8], Securify [18], Smartcheck [16], Slither [53], and Mythril [52]*, and manually inspected their generated reports. Oyente, Smartcheck, Slither, and Mythril fail to identify the reentrancy bug, whereas Security presents a lot of warnings all at the wrong places and misses the sharing-variable reentrancy vulnerability as well. In contrast, CGE successfully detects the vulnerability. These evidences reveal that the underlying detection rules of existing reentrancy vulnerability detection methods indeed can be cheated by the *sharing variable* trick and some vulnerability patterns are hard to be covered. The current rules check only the *user balance* variable that is directly related to the *call.value* invocation, while ignoring dependencies between variables, e.g., other variables may affect the *user balance* variable. In this regard, an essential highlight of our method is the capability of capturing data dependencies between critical variables.

```
1  contract Vulnerable{
2    ...
3    function getBonusWithdraw(){
4      require(!Bonus[msg.sender]);
5      Reward[msg.sender] += 0.1 ether;
6      withdrawAll(msg.sender);
7      Bonus[msg.sender] = true;
8    }
9    function withdrawAll() {
10     unit amount = Reward[msg.sender];
11     Reward[msg.sender] = 0;
12     require(msg.sender.call.value(amount )());
13   }
14 }
```

```
1  contract Malicious{
2    address vul_add=01a5f...43;
3    ...
4    function attack() {
5      vul_add.getBonusWithdraw();
6    }
7    function () payable{
8      count++;
9      if (count < 10) {
10       vul_add.getBonusWithdraw();
11     }
12   }
13 }
```

**Fig. 7 A real-world smart contract with the sharing-variable reentrancy vulnerability.**

### 5.4 Comparison with Neural Network-based Methods (RQ3)

We further compare our methods with other neural network alternatives to seek out which neural network architectures could succeed in the smart contract vulnerability detection task. The compared methods are summarized below.

- **Vanilla-RNN** [58]: A two-layer recurrent neural network, which takes the code sequence as input and evolves its hidden states recurrently to capture the sequential pattern lying in the code.
- **LSTM** [59]: The most widely used recurrent neural network for processing sequential data. LSTM is short for long short term memory, which recurrently updates the cell state upon successively reading the code sequence.
- **GRU** [60]: The gated recurrent unit, which uses gating mechanisms to handle the code sequence.
- **GCN** [37]: Graph convolutional network that takes the contract graph as input and implements layer-wise convolution on the graph using graph Laplacian.
- **DR-GCN** [20]: The degree-free graph convolutional network, which increases the connectivity of nodes and removes the diagonal node degree matrix.
- **TMP** [20]: The temporal message propagation network, which learns the contract graph feature by flowing information along the edges successively following their temporal order. The final graph feature is used for vulnerability prediction.

For a feasible comparison, Vanilla-RNN, LSTM, and GRU are fed with the contract function code sequence, represented as vectors. GCN, DR-GCN, and TMP are presented

with the normalized graph extracted from the source code and are required to detect the corresponding vulnerabilities.

We illustrate the results of different models in terms of *accuracy*, *recall*, *precision*, and *F1 score* in Table 2, while Figs. 6(d), (e), and (f) further visualize the results. Interestingly, experimental results show that Vanilla-RNN, LSTM, and GRU perform relatively worse than the state-of-the-art conventional (non-deep-learning) methods. In contrast, graph neural networks GCN, DR-GCN, and TMP, which are capable of handling graphs, achieve significantly better results than conventional 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 processing code sequentially loses valuable information from smart contract code since they ignore the structural information of contract programs, such as the data-flow and invocation relationships. The accuracies of GCN and DR-GCN are lower than TMP, this may due to the fact that GCN and DR-GCN fail to capture the temporal information induced by data flow and control flow, which is explicitly considered in TMP using ordered edges. Further, we attribute the improved performance of CGE over TMP to that TMP does not consider known security patterns and ignores key variables.

### 5.5 Ablation Study (RQ4)

By default, CGE adopts the *graph normalization* module to highlight the core nodes in the *contract graph*, it is interesting to study the effect of removing this module. Moreover, CGE incorporates an *expert pattern extraction* module and a *message propagation* module to aggregate information from both security patterns and the contract graph. It is useful to evaluate the contributions of the two modules by removing them respectively from CGE. Finally, we are also interested in exploring the effect of different network layers in CGE. In what follows, we conduct experiments to study the four aforementioned modules.

**Effect of the graph normalization module.** We removed the graph normalization module (introduced in subsection 4.2.2) from CGE, and compared it with the default CGE. The variant is denoted as CGE-WON, where WON is short for *without normalization*. Quantitative results are summarized in Table 3. We can observe that with the proposed graph normalization phase, the performance of CGE is better. For example, for reentrancy vulnerability detection task, the CGE model obtains a 2.81% and 2.55% improvement in terms of accuracy and F1 score, respectively.

Figs. 8(a) & (b) & (c) further plot the ROC curves of CGE and CGE-WON. 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. 8(a) demonstrates that CGE performs better on the reentrancy detection task, the AUC increases by 0.03 with the graph normalization module. On the timestamp dependence detection task, CGE obtains a 0.03 improvement in AUC (shown in Fig. 8(b)). On the infinite loop detection task, CGE gains a 0.04 improvement in AUC (shown in Fig. 8(c)). In the figures, we also demonstrate the effect of removing the graph normalization module of another method, namely TMP. Similar findings are observed. The experimental results suggest that program elements should contribute distinctly to vulnerability detection rather than having equal contributions.

**Effect of the security pattern module.** To evaluate the effect of our proposed security pattern module, we analyze the performance of CGE with and without the security pattern module. Towards this, we modify CGE by removing the expert pattern extraction module, utilizing only the graph feature for vulnerability learning and detection. This variant is denoted as CGE-WOE, where WOE is short for without expert pattern. The empirical findings are demonstrated in Table 3, while the visual curves are illustrated in Fig. 8(d). In Fig. 8(d), the red curve demonstrates the accuracy of CGE over different epochs on the reentrancy vulnerability detection. Obviously, we can observe that the performance of CGE is consistently superior to CGE-WOE across all epochs, revealing that incorporating security patterns is necessary and important to improve the performance. Quantitative results on all the three vulnerabilities, which are presented in Table 3, further reconfirm the finding.

We also conduct experiments to extend other neural networks with expert patterns, and empirically compared these methods with CGE. The results are illustrated in Table 4, where '-EP' denotes combining with expert patterns. We can observe that neural networks combined with expert patterns indeed achieve better results compared to their pure neural network counterparts. For example, DR-GCN-EP gains a 4.92% accuracy improvement over DR-GCN in average, and LSTM-EP obtains a 6.91% accuracy improvement over LSTM. These results indicate the effectiveness of combining neural networks with expert patterns. On the other hand, the proposed method CGE consistently outperforms other approaches including DR-GCN-EP. DR-GCN-EP ranks second in the tested methods.

**Effect of the contract graph feature extraction module.** We further investigate the impact of the contract graph feature extraction module in CGE by comparing it with its variant. Towards this, we remove the proposed contract graph construction and temporal message propagation module, while utilizing only the security pattern feature. The new variant is denoted as CGE-WOG, namely CGE without contract graph feature. Fig. 8(e) visualizes the results, where the red curve demonstrates the accuracy of CGE over different epochs, while the blue curve shows the accuracy of CGE-WOG. Clearly, the performance of CGE is consistently better compared to its variant across all epochs. Quantitative results are further presented in Table 3, where all the three vulnerabilities are involved. The results, together with the experimental results on CGE-WOE, suggest that the contract graph feature contributes significant performance gain in CGE and leads to a higher gain than the security pattern feature.

**Effect of different feature fusion networks.** When combining security pattern features and contract graph features, CGE uses a neural network with a convolution layer and a max pooling layer followed by 3 fully connected layers and a sigmoid layer. To verify this network architecture, we also study five other alternatives. First, we replace the convolution and max pooling layer with a fully connected layer, which we denote as CGE(FC). We also try replacing

| Metrics | Reentrancy | | | | Timestamp dependence | | | | Infinite loop | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | CGE-WOG | CGE-WOE | CGE-WON | CGE | CGE-WOG | CGE-WOE | CGE-WON | CGE | CGE-WOG | CGE-WOE | CGE-WON | CGE |
| Acc(%) | 82.09 | 84.42 | 86.34 | **89.15** | 81.30 | 83.52 | 86.61 | **89.02** | 72.23 | 74.68 | 79.51 | **83.21** |
| Recall(%) | 80.18 | 82.65 | 84.38 | **87.62** | 80.68 | 82.89 | 84.06 | **88.10** | 70.08 | 74.21 | 77.14 | **82.29** |
| Precision(%) | 72.15 | 78.94 | 83.35 | **85.24** | 78.42 | 80.16 | 83.90 | **87.41** | 71.44 | 73.86 | 76.26 | **81.97** |
| F1(%) | 75.95 | 80.75 | 83.86 | **86.41** | 79.53 | 81.50 | 83.98 | **87.75** | 70.75 | 74.03 | 76.70 | **82.13** |

**TABLE 3 Accuracy comparison between CGE and its variants on the three vulnerability detection tasks.**

| Variants | Reentrancy | | | | Timestamp dependence | | | | Infinite Loop | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Acc(%) | Recall(%) | Precision(%) | F1(%) | Acc(%) | Recall(%) | Precision(%) | F1(%) | Acc(%) | Recall(%) | Precision(%) | F1(%) |
| Vanilla-RNN-EP | 56.06 | 60.24 | 58.21 | 59.20 | 54.58 | 49.65 | 59.35 | 54.07 | 54.72 | 52.62 | 49.94 | 51.24 |
| LSTM-EP | 60.15 | 72.26 | 58.68 | 64.77 | 59.82 | 63.38 | 58.28 | 56.29 | 56.52 | 59.98 | 49.75 | 54.39 |
| GRU-EP | 62.08 | 75.01 | 60.13 | 66.75 | 61.22 | 64.18 | 58.45 | 61.18 | 57.09 | 60.54 | 49.81 | 54.65 |
| GCN-EP | 80.96 | 81.05 | 76.84 | 78.89 | 79.32 | 79.94 | 73.65 | 76.67 | 70.06 | 69.81 | 64.29 | 66.94 |
| DR-GCN-EP | 85.14 | 84.12 | 79.38 | 81.68 | 83.74 | 84.02 | 80.59 | 82.27 | 74.36 | 73.08 | 69.45 | 71.22 |
| CGE(LSTM) | 86.74 | 85.18 | 82.85 | 84.00 | 87.92 | 85.08 | 87.13 | 86.09 | 79.18 | 78.25 | 76.80 | 77.52 |
| CGE(FC) | 87.64 | 85.74 | 82.97 | 84.33 | 88.12 | 87.98 | 85.04 | 86.49 | 80.62 | 78.96 | 77.24 | 78.09 |
| CGE(1-FC) | 88.54 | 86.12 | 83.80 | 84.94 | 86.62 | 87.82 | 81.73 | 84.66 | 81.43 | 81.25 | 80.98 | 81.11 |
| CGE(2-FC) | 88.89 | 86.47 | 84.51 | 85.48 | 87.05 | 84.96 | 85.02 | 84.98 | 81.82 | 81.76 | 80.54 | 81.15 |
| CGE(AP) | 88.02 | 85.92 | 83.45 | 84.67 | 85.25 | 85.16 | 81.84 | 83.47 | 79.53 | 78.58 | 76.94 | 77.75 |
| **CGE** | **89.15** | **87.62** | **85.24** | **86.41** | **89.02** | **88.10** | **87.41** | **87.75** | **83.21** | **82.29** | **81.97** | **82.13** |

**TABLE 4 Upper: Performance comparison between CGE and other neural networks combined with expert patterns. '-EP' denotes combining with expert patterns. Lower: Comparison with other feature fusion network architectures.**

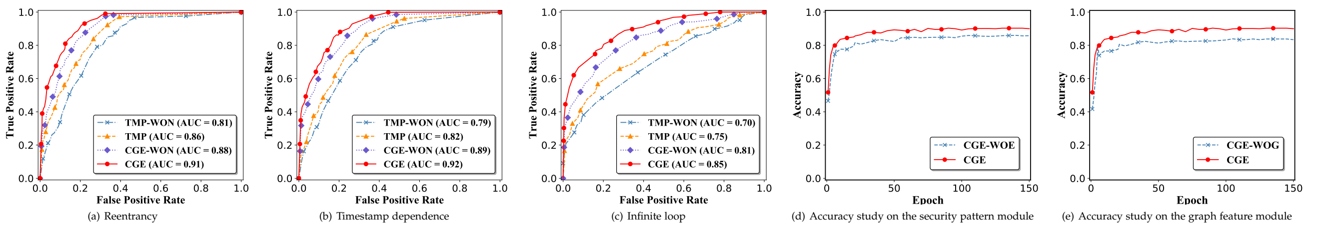



**Fig. 8 Curves comparison: (a), (b), and (c) present the ROC analysis of graph normalization module for TMP, CGE, and their variants on the three vulnerability detection tasks, where AUC stands for area under the curve. In (d), the two curves study the effect of removing the security pattern extraction module, while (e) presents the study on removing the contract-graph feature extraction module.**

them with an LSTM layer, which we term as CGE(LSTM). Then, we keep the convolution and max pooling layer, but change the 3 fully connected layers to 1 or 2 fully connected layers. The two variants are denoted as CGE(1-FC) and CGE(2-FC), respectively. Finally, we explore replacing the max pooling layer with an average pooling layer, namely CGE(AP), while keeping other layers fixed. The empirical results are illustrated in Table 4. The results reveal that: 1) RNN architectures such as LSTM are not suitable for the feature fusion task, 2) the default setting of CGE yields better results than the five alternatives, and 3) using average pooling or changing the number of fully connected layers leads to a slight performance drop.

## 6 Discussions

**Specialty of our method in dealing with smart contracts.** Distinct from conventional programs that consume only CPU resources, users have to pay a fee for executing each line of smart contract code. The fee is approximately proportional to how much code needs to run and is referred to as *gas*. Therefore, in the proposed method, we studied the infinite loop vulnerability since an *infinite loop* will consume a lot of gas but all the gas is consumed in vain. This is because the infinite loop is unable to change any state (any execution that runs out of gas is aborted). Moreover, the function libraries of the smart contracts and other program languages are quite different. For example, *call.value* and *block.timestamp* are unique and specially designed in smart contracts. We implement an open-sourced tool to analyze the specific syntax of smart contract statements. We also use core nodes to symbolize invocations and variables closely related to a specific vulnerability, and represent other variables and invocations as normal nodes. We would like to point out that there is a unique fallback mechanism in smart contracts, which is different from other programming languages. In the contract graph, we build a fallback node to stimulate the fallback function of a virtual attack contract, which can interact with the function under test.

**Discussions on the contract graph.** Existing efforts adopted the *control flow graph*, *code property graph*, and *abstract syntax tree* to represent program code. The differences between them and our *contract graph* can be summarized as: (i) Control flow graph utilizes a node to model a basic block, *i.e. a straight-line piece of code without any jumps*, and uses edges to represent jumps [61]. They focus mainly on execution path jumps and tend to consider each node as of equal importance. (ii) *Code property graph* [62, 63] models statements as nodes, and represents the control flow between statements as edges. (iii) *Abstract syntax tree* [64, 65] adopts a tree representation of the abstract syntactic structure of source code, which relies on a tree structure and has difficulties in fully characterizing the rich semantic information between nodes. (iv) In our contract graph, nodes are used to model variables and invocations related

to a specific vulnerability and are classified into different categories, i.e. *core nodes, normal nodes*, and *fallback nodes*. We also explicitly model the order of the edges following their temporal order in the code and consider the specific fallback mechanism of the smart contracts.

# 7 Conclusion and Future Work

In this paper, we have proposed a fully automated approach for smart contract vulnerability detection at the function level. In contrast to existing approaches, we combine both expert patterns and contract graph semantics, consider rich dependencies between program elements, and explicitly model the fallback mechanism of smart contracts. We also explore the possibility of using novel graph neural networks to learn the graph feature from the contract graph, which contains rich control- and data- flow semantics. Extensive experiments are conducted, showing that our method significantly outperforms the state-of-the-art vulnerability detection tools and other neural network-based methods. We believe our work is an important step towards revealing the potential of combining deep learning with conventional patterns on smart contract vulnerability detection tasks. For future work, we will investigate the possibility of extending this method to smart contracts that have only bytecode, and explore this architecture on more other vulnerabilities.

# Acknowledgments

This paper was supported by the Natural Science Foundation of Zhejiang Province, China (Grant No. LQ19F020001), the National Natural Science Foundation of China (No. 61902348, 61802345), and the Research Program of Zhejiang Lab (2019KD0AC02).

# References


[1] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *ICMD*, 2017, pp. 1085–1100.

[2] D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain technology overview," *arXiv preprint arXiv:1906.11078*, 2019.

[3] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas, "Bitcoin as a transaction ledger: A composable treatment," in *Annual International Cryptology Conference*, 2017, pp. 324–356.

[4] M. Dhawan, "Analyzing safety of smart contracts," in *Proceedings of the NDSS*, 2017, pp. 16–17.

[5] M. Tsikhanovich, M. Magdon-Ismail, M. Ishaq, and V. Zikas, "Pd-ml-lite: Private distributed machine learning from lighweight cryptography," *arXiv preprint arXiv:1901.07986*, 2019.

[6] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang, "Untangling blockchain: A data processing view of blockchain systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 7, pp. 1366–1385, 2018.

[7] L. S. Sankar, M. Sindhu, and M. Sethumadhavan, "Survey of consensus protocols on blockchain applications," in *Proceedings of the ICACCS*, 2017, pp. 1–5.

[8] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *CCS*, 2016, pp. 254–269.

[9] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*, 2018.

[10] A. Bahga and V. K. Madisetti, "Blockchain platform for industrial internet of things," *Journal of Software Engineering and Applications*, vol. 9, no. 10, p. 533, 2016.

[11] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, p. 37, 2014.

[12] J. Kokina, R. Mancha, and D. Pachamanova, "Blockchain: Emergent industry adoption and implications for accounting," *Journal of Emerging Technologies in Accounting*, vol. 14, no. 2, pp. 91–100, 2017.

[13] "The dao smart contract," Website, 2016, http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413.

[14] "King of the ether," Webiste, 2016, https://www.kingoftheether.com/postmortem.html.

[15] "An in-depth look at the parity multisig bug," Website, 2017, http://hackxingdistributed.com/2017/07/22/deep-dive-parity-bug.

[16] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *WETSEB*, 2018, pp. 9–16.

[17] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the ASE*, 2018, pp. 259–269.

[18] 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 CCS*, 2018, pp. 67–82.

[19] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, "Towards automated reentrancy detection for smart contracts based on sequential models," *IEEE Access*, vol. 8, pp. 19 685–19 695, 2020.

[20] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network," in *Proceedings of the IJCAI-20*, 7 2020, pp. 3283–3290.

[21] W. J. Tann, X. J. Han, S. S. Gupta, and Y. Ong, "Towards safer smart contracts: A sequence learning approach to detecting vulnerabilities," *CoRR*, 2018.

[22] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, 2016, pp. 91–96.

[23] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*, 2018, pp. 243–269.

[24] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *CSF*, 2018, pp. 204–217.

[25] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *International Conference on Financial Cryptography and Data Security*, 2017, pp. 520–535.

[26] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Annual Computer Security Applications Conference*, 2018, pp. 653–663.

[27] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *NDSS*, 2018.

[28] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the ICSE*, 2018, pp. 65–68.

[29] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proceedings of the NDSS*, 2019.

[30] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *TNSE*, 2020.

[31] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*, 2017, pp. 164–186.

[32] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, "An end-to-end deep learning architecture for graph classification," in *AAAI*, 2018.

[33] H. Wang, P. Zhang, X. Zhu, I. W.-H. Tsang, L. Chen, C. Zhang, and X. Wu, "Incremental subgraph feature selection for graph classification," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 1, pp. 128–142, 2016.

[34] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 10 197–10 207.

[35] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations (ICLR)*, 2018.

[36] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 9, pp. 1616–1637, 2018.

[37] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proceedings of the ICLR*, 2017.

[38] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional

neural networks on graphs with fast localized spectral filtering," in *Advances in neural information processing systems*, 2016, pp. 3844–3852.
[39] X. Zhou, F. Shen, L. Liu, W. Liu, L. Nie, Y. Yang, and H. T. Shen, "Graph convolutional network hashing," *IEEE transactions on cybernetics*, 2018.
[40] Y. Wei, X. Wang, L. Nie, X. He, R. Hong, and T.-S. Chua, "Mmgcn: Multi-modal graph convolution network for personalized recommendation of micro-video," in *Proceedings of the 27th ACM MM*, 2019, pp. 1437–1445.
[41] R. Li, S. Wang, F. Zhu, and J. Huang, "Adaptive graph convolutional neural networks," in *AAAI*, 2018.
[42] A. Micheli, "Neural network for graphs: A contextual constructive approach," *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 498–511, 2009.
[43] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.
[44] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D.-Y. Yeung, "Gaan: Gated attention networks for learning on large and spatiotemporal graphs," *arXiv preprint arXiv:1803.07294*, 2018.
[45] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the ICML*, 2017, pp. 1263–1272.
[46] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *CCS*, 2017, pp. 363–376.
[47] S. Shen, S. Shinde, S. Ramesh, A. Roychoudhury, and P. Saxena, "Neuro-symbolic execution: Augmenting symbolic execution with neural constraints." in *Proceedings of the NDSS*, 2019.
[48] R. A. Rossi, R. Zhou, and N. Ahmed, "Deep inductive graph representation learning," *IEEE Transactions on Knowledge and Data Engineering*, 2018.
[49] "Etherscan," Website, 2015, https://etherscan.io/.
[50] "Vntchain," Website, 2018, https://github.com/vntchain/go-vnt.
[51] "Ethereum," Website, 2015, https://github.com/ethereum/go-ethereum.
[52] B. Mueller, "A framework for bug hunting on the ethereum blockchain," Webiste, 2017, https://github.com/ConsenSys/mythril.
[53] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *WETSEB*, 2019, pp. 8–15.
[54] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard, "Detecting and escaping infinite loops with jolt," in *European Conference on Object-Oriented Programming*, 2011, pp. 609–633.
[55] 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.
[56] A. Ibing and A. Mai, "A fixed-point algorithm for automated static detection of infinite loops," in *IEEE 16th International Symposium on High Assurance Systems Engineering*, 2015, pp. 44–51.
[57] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen, "Looper: Lightweight detection of infinite loops at runtime," in *Proceedings of ASE*, 2009, pp. 161–169.
[58] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *Proceedings of ICNN*, vol. 1, 1996, pp. 347–352.
[59] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory recurrent neural network architectures for large scale acoustic modeling," in *Fifteenth annual conference of the international speech communication association*, 2014.
[60] 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.
[61] A. V. Phan, M. Le Nguyen, and L. T. Bui, "Convolutional neural networks over control flow graphs for software defect prediction," in *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2017, pp. 45–52.
[62] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.
[63] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, "Learning to map source code to software vulnerability using code-as-a-graph," *arXiv preprint arXiv:2006.08614*, 2020.
[64] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, 2016.
[65] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

**Zhenguang Liu** is currently a professor of Zhejiang Gongshang University. He had been a research fellow in National University of Singapore and A*STAR. He respectively received his Ph.D. and B.E. degrees from Zhejiang University and Shandong University, China. His research interests include smart contract security and multimedia data analysis. Dr. Liu has served as technical program committee member for conferences such as ACM MM, CVPR, AAAI, IJCAI, and ICCV, session chair of ICGIP, local chair of KSEM, and reviewer for IEEE TVCG, IEEE TPDS, ACM TOMM, etc.

**Peng Qian** received his BSc degree in software engineering from Yangtze University, MSc degree in computer science from Zhejiang Gongshang University, in 2018 and 2021. He is currently pursuing a Ph.D. at Zhejiang University. His research interests include blockchain security, graph neural network, and deep learning.

**Xiaoyang Wang** received the BSc and MSc degrees in computer science from Northeastern University, China, in 2010 and 2012, respectively, and the PhD degree from the University of New South Wales, Australia, in 2016. He is a professor in Zhejiang Gongshang University, Hangzhou, China. His research interest includes query processing on massive graph data.

**Yuan Zhuang** received her PhD from the College of Computer Science and Technology (CCST), Jilin University, China. Her research interests include blockchain security, machine learning, big data processing and distributed computing

**Lin Qiu** is a PhD candidate at the Department of Information Systems and Analytics, National University of Singapore, Singapore. Before that, she obtained her bachelor degree from Xiamen University, China. Her research interests lie in deep learning, healthcare, and blockchain.

**Xun Wang** is currently a professor at the School of Computer Science and Information Engineering, Zhejiang Gongshang University, China. He received his BSc in mechanics, Ph.D. degrees in computer science, all from Zhejiang University, China, in 1990 and 2006, respectively. His research interests include intelligent information processing and computer vision. He has published over 100 papers in high-quality journals and conferences. He is a member of the IEEE and ACM, and a distinguished member of CCF.