IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 6, 2020, accepted January 17, 2020, date of publication January 27, 2020, date of current version January 31, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2969429

Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models

PENG QIAN[1], ZHENGUANG LIU[1], QINMING HE[2], ROGER ZIMMERMANN[3], (Senior Member, IEEE), AND XUN WANG[1], (Member, IEEE)

[1]School of Computer and Information Engineering, Zhejiang Gongshang University, Hangzhou 310018, China
[2]Department of Computer Science, Zhejiang University, Hangzhou 310027, China
[3]School of Computing, National University of Singapore, Singapore 117417

Corresponding author: Zhenguang Liu (lzg@zjgsu.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB1401300 and Grant 2017YFB1401304, in part by the National Natural Science Foundation of China Grant 61902348, in part by the Natural Science Foundation of Zhejiang Province, China, Grant LQ19F020001, in part by the Science and Technology Innovation Program of Zhejiang Province, China, under Grant 2019R408070, and in part by the General Scientific Research Projects of Zhejiang Provincial Department of Education, China, under Grant Y201942758.

ABSTRACT In the last decade, smart contract security issues lead to tremendous losses, which has attracted increasing public attention both in industry and in academia. Researchers have embarked on efforts with logic rules, symbolic analysis, and formal analysis to achieve encouraging results in smart contract vulnerability detection tasks. However, the existing detection tools are far from satisfactory. In this paper, we attempt to utilize the deep learning-based approach, namely bidirectional long-short term memory with attention mechanism (BLSTM-ATT), aiming to precisely detect reentrancy bugs. Furthermore, we propose *contract snippet* representations for smart contracts, which contributes to capturing essential semantic information and control flow dependencies. Our extensive experimental studies on over 42,000 real-world smart contracts show that our proposed model and *contract snippet* representations significantly outperform state-of-the-art methods. In addition, this work proves that it is practical to apply deep learning-based technology on smart contract vulnerability detection, which is able to promote future research towards this area.

INDEX TERMS Blockchain, smart contract, deep learning, sequential models, vulnerability detection.

I. INTRODUCTION

Software or program carrying security flaws can potentially allow attackers to compromise systems and applications. The same holds for smart contracts on the blockchain networks. Smart contracts de facto are programs that control automatic execution for multiple transactions in the peer-to-peer network [1]. However, smart contracts hold cryptocurrency worth billions of dollars, making them attractive enough to attackers. With the increasing number of smart contracts, more and more security issues are exposing correspondingly. Attackers make maliciously use of smart contract vulnerabilities to invade blockchain networks, which causes huge losses to both blockchain systems and smart contract holders. According to the statistics of SlowMist Hacked [2], up to now, blockchain platforms containing ETH [3], EOS [5], and TRON [6] have suffered losses valued more than 1.2 billion USD due to the security attack to smart contracts.

Ethereum [3], one of the most popular blockchain platforms, has deployed tens of thousands of smart contracts, controlling billions of dollars worth of Ether (Cryptocurrency of Ethereum). Because of that, many Ethereum smart contract security events caused by attackers are also emerging, while losses are especially damaging due to the irreversibility and immutability of smart contracts. We can only watch Ether flow into the attacker's pocket but fail to stop it. In June 2016, over 3.6 million Ether was stolen by hackers exploiting the reentrancy vulnerability of *The DAO Attack* [7], incurring losses of over 60 million USD. Additionally, in November 2017, 300 million USD worth of Ether was frozen due to Parity's MultiSig wallet [8]. It can be considered that more and more vulnerabilities are discovered and exploited, highlighting an imperative requirement for the security of smart contracts. Therefore, effective vulnerability detection tools for smart contracts are essential and urgently required,

The associate editor coordinating the review of this manuscript and approving it for publication was Vincenzo Piuri.

especially with the expansion transactions reliance on smart contracts in almost blockchain platforms.

On one side, there has been an increasing interest in the security of smart contracts, attempting to discover and identify diverse vulnerabilities of smart contracts. Previous efforts focused on smart contract analysis are mainly based on formal analysis methods and expert-defined hard logic rules. Existing methods for automatically vulnerability detection have applied symbolic execution [9], [10] and dynamic analysis [11], [12]. However, due to the easy-prone and easy-bypass of hard logic rules, these approaches cannot adapt to more general scenarios, leading to low detection accuracy and precision. On the other side, with the advancement of deep learning technology, recurrent neural networks (RNNs) have achieved great success and widespread application in natural language processing (NLP). RNNs are powerful deep learning networks adapted to sequential data. Generally, RNNs are able to learn short-term dependencies rather than long-term ones. To this end, the long short-term memory (LSTM) model [17] has been correspondingly introduced and used to solve difficult sequential problems such as speech recognition [16], sentiment analysis [13], and text prediction [18]. Although more and more practices are based on deep learning, there is still a lack of effort exploring to use a deep learning-based method for smart contract vulnerability detection due to the novelty and complexity of smart contracts.

Towards this target, we introduce sequential models for reentrancy detection at a contract source code level. Inspired by [20], we apply the bidirectional long short-term memory with the attention mechanism in our sequential models, namely BLSTM-ATT. Since the BLSTM-ATT model has been applied in other areas of smart contracts, such as smart contract automatic classification [19], we deem that the BLSTM-ATT applied to smart contract vulnerability detection and achieving encouraging results is reasonable and operable. Further, considering many program statements in a smart contract is irrelevant information to reentrancy detection, we propose using a *contract snippet* to represent a smart contract for capturing key semantic sentences. A *contract snippet* is a number of lines of a smart contact that are not only semantically related to each other but also critically to capture essential information such as control flow or data dependencies. After obtaining the *contract snippets*, they can be vectorized as input to our sequential models.

To the best of our knowledge, this is the first deep learning-based approach to smart contract reentrancy detection with regard to a smart contract source code level. Our key contributions are as follows:

- We show that our deep learning-based approach (i.e., sequential models) outperforms state-of-the-art smart contract vulnerability detection tools.
- We present a method of transforming smart contract source code and assembling *contract snippets* for the reentrancy detection task. Extensive experiments are conducted on the *contract snippets* dataset that we plan to open-source, achieving high performance with an 88.47% accuracy and 88.26% F1-Score via our sequential model BLSTM-ATT.
- By proving that our proposed deep learning approach is a competitive alternative to current formal analysis tools, we initiate the study for future work using deep learning methods to ensure smart contracts away from intrusion.

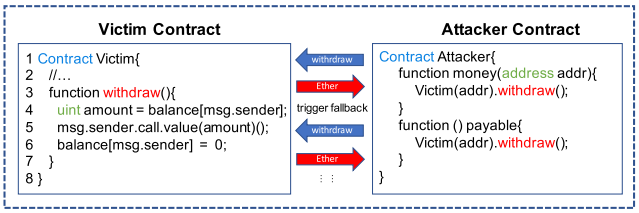


FIGURE 1. An real-world instance of smart contract reentrancy attack.

The remainder of the paper is as follows. We first discuss the motivations of our work in Section II. Next, we present some related works in Section III. After that, we elaborate on our proposed methodology in Section IV. We then conduct experimental evaluations in Section V. Finally, we conclude the whole paper and future work in Section VI.

II. MOTIVATION

Since smart contract security issues have led to a loss of over 1.2 billion USD, it gives rise to great doubts about the security status of blockchain, which further affects the expansion and popularization of blockchain technology at its initial stage. In particular, the impact of *The Dao Attack* [7] once made the development of the blockchain industry at low tide for a long time. The main reason for this notorious event is the reentrancy vulnerability of a smart contract. Generally, when an attacker initiates an Ether transfer operation to a certain contract address, it will force the execution of the *fallback* function that lies in the attack contract itself. At this point, malicious behaviors hidden in the *fallback* function can probably activate a re-enter behavior, which leads to duplicate transfer operations.

In a smart contract, there is a *fallback* function with no name, which neither takes no arguments nor any return values. This *fallback* function, in such two cases, will be triggered automatically: (1) when a contract called but no function matched, the *fallback* function then will be called by default; (2) when a contract receives Ether transfer, the *fallback* function can also be executed.

FIGURE 1 shows a real-world instance of a smart contract reentrancy attack. This attack exploits the second trigger condition by the smart contract *fallback* function described in the previous paragraph. The function *money*, in the contract *Attacker*, attempts to execute a withdraw operation by calling the function *withdraw* of contract *Victim*, which involves the Ether transfer method. Consequently, this will irreversibly make the *fallback* function in the contract *Attacker* to be activated, and the *fallback* function performs the function *withdraw* again, which incurs the withdraw operation to be executed until Ether in the contract *Victim* is exhausted. This process indicates the constant loss of Ether with the most

typical case is *The DAO Attack*. People can only watch Ether flow into the intruder's pocket without any rescue measure.

All in all, our main motivations can be summarized as the following three parts: (1) The damage of smart contract reentrancy vulnerability is tremendous and irreversible. (2) The analysis and detection of reentrancy vulnerability are difficult and challenging. (3) Existing work that relies on formal analysis would fail to achieve precise judgment whilst catching high *false positive* and *false negative*. Therefore, solutions to smart contract security issues are urgently required.

III. RELATED WORK

On one hand, smart contract security issues promote the birth of several smart contract detection tools, such as *Securify* [9], *Mythril* [21], *Oyente* [10], and *Smartcheck* [22]. On the other hand, the adaptability of maturity deep learning model to sequential data, especially recurrent neural networks (RNNs), has also fostered the development of deep learning-based program vulnerability detection. Here we first introduce the related content containing smart contracts and reentrancy. We then review existing works on vulnerability detection and deep learning-based methods on security analysis, which are the most relevant to our work.

A. SMART CONTRACTS AND REENTRANCY

With the rapid development in recent times, blockchain technologies have been applied to many other areas, such as the Internet of Things [38], [39] and copyright protection [40]. Smart contracts are programs running on the blockchain network, which can execute transactions automatically and irreversibly. The principle of smart contracts was created in 1996 by Nick Szabo, who is an important historical figure of blockchains as the inventor of Bit Gold [23]. As mentioned in many corners of the world, smart contracts are lines of code that deployed on a blockchain network and automatically execute transactions in a transparent and decentralized way [24]. Once deployed, a smart contract is immutable and almost no remedy can be used to handle bugs during the contract execution process. Ethereum smart contracts are the most universal and frequent-used under the emergence of various blockchain platforms. Nevertheless, because of that, the types of smart contract vulnerabilities are correspondingly more diversified.

In our work, we focus on the most common and critical vulnerability that has been reported for EVM-based smart contracts, i.e., reentrancy vulnerability. This bug is exploited when a contract attempts to send Ether before having updated its internal state. A reentrancy attack can occur when you create a function that makes an external call to another untrusted contract before it resolves any effects. If the attacker can control the untrusted contract, they can make a recursive call back to the original function, repeating interactions that would have otherwise not run after the effects were resolved.

B. EXISTING EFFORTS FOR SMART CONTRACT VULNERABILITY DETECTION

There have been increasing efforts in order to prevent such attacks and make smart contracts more secure in general. There are some works that rely on formal analysis and symbolic execution methods, such as *Oyente* and *Securify*. *Oyente* [10] is one of the early works to uses symbolic execution to automatically check for smart contract vulnerabilities, while *Securify* [9] is a static analysis tool that checks security properties of the EVM bytecode of smart contracts. *Mythril* [21] uses taint analysis and control flow checking to detect vulnerabilities, while*Smartcheck* [22] runs bug analysis in solidity source code and automatically checking. Another static analysis tool is *ZEUS* [34], which can check for a vast range of vulnerabilities such as reentrancy, unhandled exceptions, transaction order dependency, and others. *Maian* [35] is also a tool to analyze contracts but instead of using static analysis to find bugs in the contract, it tries to capture vulnerabilities across long sequences of invocations of a contract. Recent researches on vulnerability detection exploit dynamic contract execution. *ContractFuzzer* [12] is introduced to identify vulnerabilities by fuzzing and runtime behavior monitoring during execution. Similarly, Liu et al. [11] present a fuzzing-based analyzer named *Reguard* to automatically detect reentrancy bugs. In our experiments, we mainly focused on the above four tools including *Securify*, *Mythril*, *Oyente*, and *Smartcheck*, while the reasons of other tools (e.g. *Reguard* and *ContractFuzzer*) have no comparison are that: (1) no open-sourced; (2) required to manually write attack contracts to trigger reentrancy bugs when detecting reentrancy vulnerability, which is impractical in our settings.

C. SECURITY ANALYSIS WITH DEEP LEARNING

With deep learning technology widely used in the field of program security, a variety of security detection tasks emerge one after another. Li *et al.* [25] initiate the study of using a deep learning-based method to detect vulnerability in the C/C++, which uses the long short-term memory (LSTM). Gupta *et al.* [26] use RNN Encoder-Decoder to fix common C language errors. The work in the [27] also shows that LSTM is even more effective in code modeling, which inspires later works to use it for learning vulnerability features. Huo *et al.* [28] use convolutional neural networks (CNNs) for bug localization in the source code. Mou *et al.* [36] explore the potential of deep learning for program analysis by embedding the nodes of the abstract syntax tree representations of source code and training a tree-based convolutional neural network (TBCNN) for simple supervised classification problems. Harer *et al.* [37] propose an adversarial learning approach and trained an RNN to detect vulnerabilities motivated by the problem of automated repair of software vulnerabilities. These methods based on deep learning have achieved good results in their respective tasks.

However, to our knowledge, although more and more program vulnerability detection practices explore deep learning, there is still a lack of deep learning-based approaches for smart contract vulnerability detection. The reasons can be concluded as: (1) The explosive growth of deep learning began in recent years. Even for general program languages, such as C/C++, Java, deep learning has not been applied

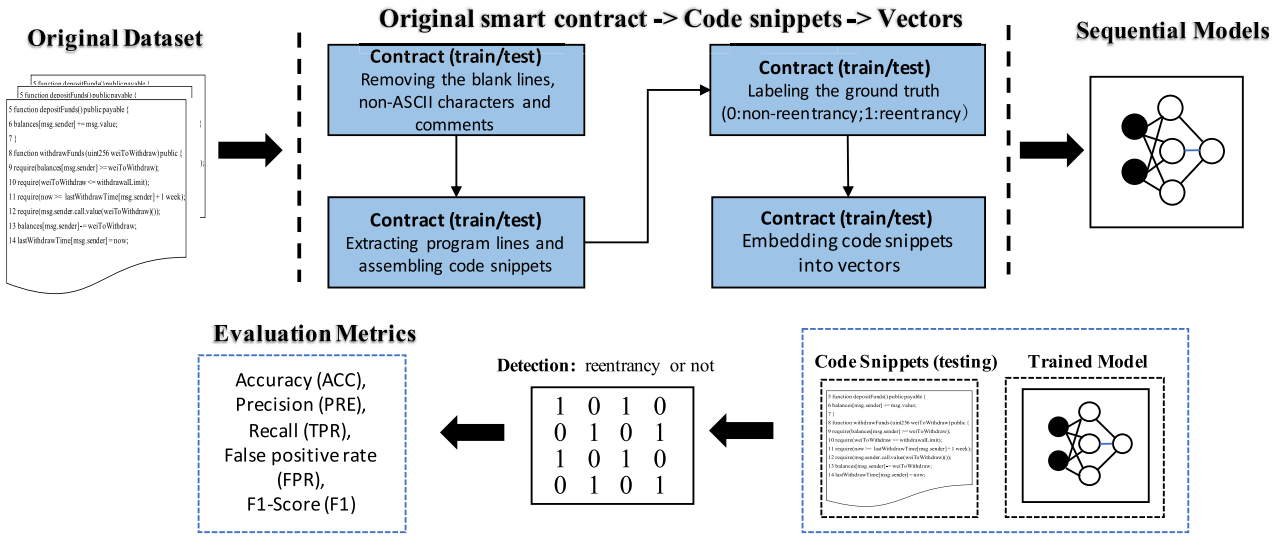


FIGURE 2. The overall process of automated reentrancy detection that contains multiple steps.

to the security vulnerability detection of these programming languages until recent times. (2) In the past, there are no enough smart contracts that can support the training of a neural network. Now as the total number of smart contracts in various blockchain platforms becomes large enough, time is ripe for adopting neural networks for vulnerability detection. In fact, this is also the original intention of our work, aiming to explore the possibility of detecting vulnerabilities of smart contracts via deep learning methods.

IV. METHODOLOGY

A. OVERVIEW

Our target is to design a vulnerability detection approach that can automatically tell whether a given smart contract is reentrancy or not. Here our process to automatically reentrancy detection goes through multiple steps as shown in FIGURE 2. First, given an original smart contract, data cleaning is necessary such as removing blank lines, non-ASCII characters, and irrelevant comments. We then consider transforming original smart contracts into *contract snippets*, which composes of key program statements. Next, we label the ground truth for each *contract snippet*. Afterward, we parse each *contract snippet* into a sequence of code tokens, which are embedded into feature vector representations. Finally, in the experimental stage, we attempt to feed the feature vectors into our sequential models to train the detector, achieving the reentrancy bug detection. For evaluation, we give the details in section V.

Next, we explore three parts of the aforementioned steps in detail. We first elaborate on the *contract snippet* representations for smart contracts in subsection IV-B. We then introduce the vector representation for *contract snippets* in subsection IV-C, while the sequential model BLSTM-ATT built in subsection IV-D.

B. CONTRACT SNIPPET REPRESENTATION FOR SMART CONTRACT

A smart contract (in Ethereum) is a program written by the high-level language *Solidity*, which is actually composed of multiple lines of codes. However, some lines of codes in a contract might be unrelated to the reentrancy vulnerability analysis. For example, there may be code explanatory comments and irrelevant functions in a smart contract, which is unhelpful to our goal (i.e., reentrancy vulnerability detection). Hence, we try to condense a smart contract into a highly-expressive *contract snippet*, facilitating more precise feature extraction.

A *contract snippet* not only incurs semantically related to each other in terms of control flow dependency but also highlights the key point to reentrancy detection (i.e., *call.value*). Given a smart contract, *call.value* is an essential factor to reentrancy due to its no gas limitation. Secondly, the control flow dependency between program statements is also the key point. As shown in FIGURE 1, assuming the statement (line 6) is moved before the transfer statement *"msg.sender.call.value(amount)();"* (line 5), the occurrence of the next transfer will be avoided in that its balance has been set to zero, which means that this smart contract is safe. On the contrary, if the vulnerable contract transfers funds before it sets the balance to zero, the attacker can recursively call the withdraw function repeatedly and drain the Ether of the whole contract as illustrated in FIGURE 1.

As such, corresponding to the key points of smart contracts, *contract snippets* can be generated by the means of control flow analysis. It is worth mentioning that the *contract snippet* we end up with is uncommented and no blank lines, which have no effect to do with our detection target.

C. VECTOR REPRESENTATION FOR CONTRACT SNIPPET

Since deep neural networks generally take vectors as input, we need to represent the *contract snippets* of smart contracts as vectors that are semantically meaningful for the reentrancy detection. First of all, before each *contract snippet* is generated as a vector, we try to obtain its symbolic representation with operations as: (1) mapping user-defined variables to symbolic names (e.g., "VAR1", "VAR2"); (2) mapping user-defined functions to symbolic names

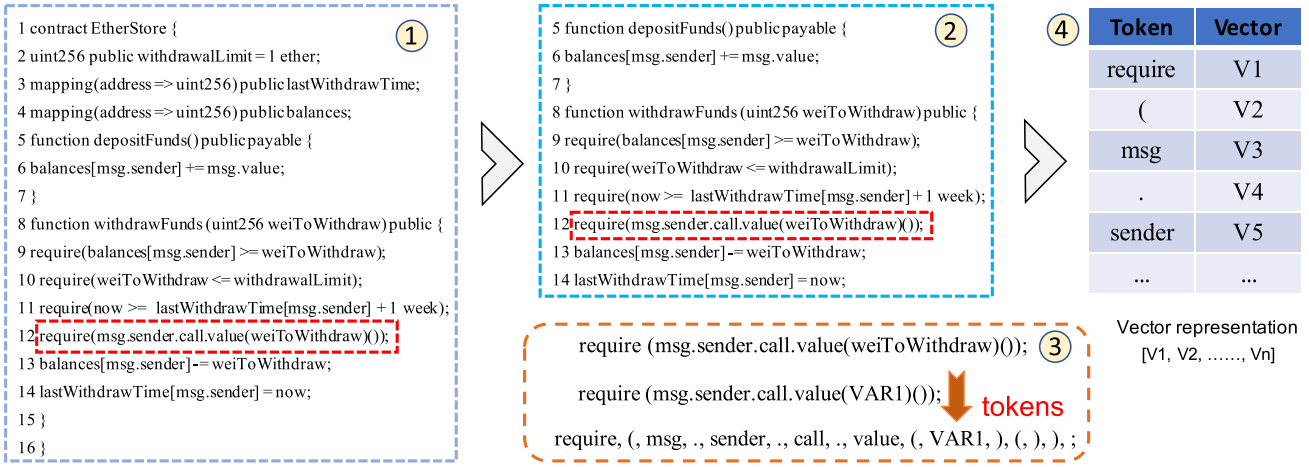


FIGURE 3. A specific representation process from contract source code to vector representation, including (1) contract snippet representation; (2) symbolic representation; (3) vector representation.

(e.g., "FUN1", "FUN2"). After that, we divide a *contract snippet* in the symbolic representation into a sequence of tokens via lexical analysis. For example, a specific conversion process has been depicted in step 3 of FIGURE 3, in which the original line is represented by a sequence of several tokens, including keywords, operations, built-in normalized variables, and symbols. Actually, a code token state also contains information from other code tokens that come before it, which captures the information of semantics and control flow dependency.

Then, with a large corpus of tokens of *contract snippets* obtained, we convert these tokens into vectors via the *word2vec* [29]. This tool is one of the most popular techniques to learn word embeddings using a shallow neural network, which maps a token to an integer that is then converted to a fixed-dimension vector. Owing to *contract snippets* that may have different numbers of tokens, the corresponding vectors converted may have different lengths. Therefore, in order to take the equal-length vectors as input, we make such adjustments: (1) pad zeros to the end of the vector when the length is less than the fixed-dimension; (2) truncate the ending part of the vector when the length exceeds the fixed-dimension.

To sum up, the *contract snippet* and vector representation are the key steps of smart contract source code processing, which is conducive to improving the performance of reentrancy detection. We have shown a vivid example from smart contracts source code to *contract snippets* to vectors representation in FIGURE 3. Next, we describe our sequence model (i.e., BLSTM-ATT) in detail in the upcoming subsection.

D. SEQUENTIAL MODELS FOR VULNERABILITY DETECTION

1) SEQUENTIAL MODELS CONSTRUCTION

Recurrent neural networks (RNNs) are popular and powerful models that have shown great promise in many areas such as audio analysis [14], and other NLP tasks [13], [15], [30], which the idea behind RNNs is to make use of sequential information. Mathematically the simple RNN can be formulated as follows:

$$h(t) = f_H(W_{IH}x(t) + W_{HH}h(t-1)) \quad (1)$$

$$y(t) = f_O(W_{HO}h(t)) \quad (2)$$

where $x(t)$ and $y(t)$ are regarded as the input and output vectors. W_{IH}, W_{HH}, and W_{HO} are treated as the weight matrices, while f_H and f_O are the hidden and output unit activation functions, respectively. However, vanilla RNNs are able to easily learn short-term dependency but not the long-term ones.

In order to address the long-term dependency of vanilla RNNs, the long short-term memory (LSTM) network was proposed, which is capable of learning long-term dependency. A single LSTM unit is composed of an input gate i_t, an output gate o_t, a forget gate f_t and a cell state C_t as depicted in FIGURE 5, which facilitates the cell to remember values for an arbitrary amount of time and controls the flow of information. The hidden state h_t for an LSTM cell can be calculated as follows:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \quad (3)$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (4)$$

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (5)$$

$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (6)$$

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad (7)$$

$$h_t = o_t \odot tanh(C_t) \quad (8)$$

where $\tilde{C}_t$ is a vector of new candidate values, σ is the sigmoid function, *tanh* is the hyperbolic tangent function, and $\odot$ denotes matrix multiplication and element-wise product. Note that they have the exact same equations, just with different parameter matrices W. Since the standard LSTM yet cannot capture the future information in its sequence, we add a layer of reverse LSTM, namely Bidirectional-LSTM [30].

Furthermore, in order to highlight the importance of some output results for vulnerability detection, we introduce the attention mechanism [33] making the final sequential model

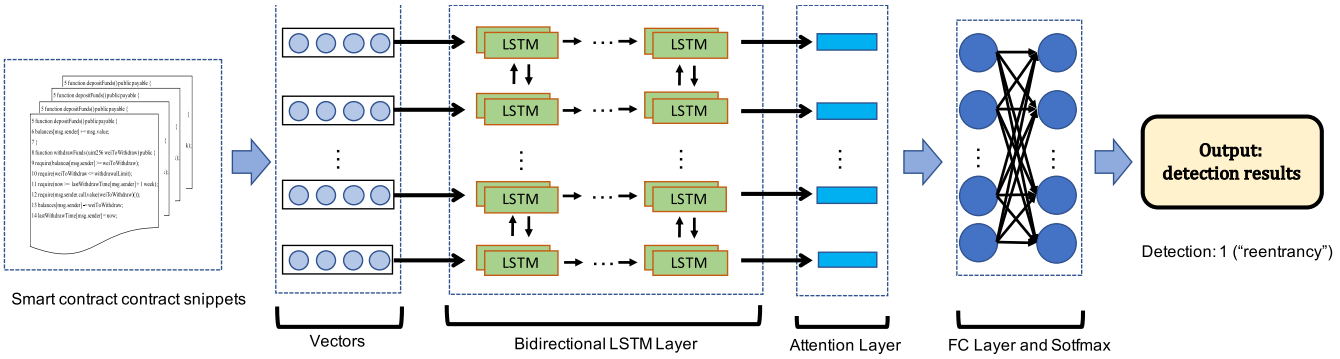


FIGURE 4. The architecture of our Bidirectional-LSTM with an attention mechanism. First, the vectorized smart *contract snippet* input to the BLSTM layer. Then, an attention layer is added to highlight important weight. Last, through the FC layer and Softmax, the detection result is produced.

obtained (i.e., BLSTM-ATT), which authentically improves the effect of experiments. For example, as to important words in the code lines (e.g., *call.value*), we use the attention mechanism to give weight that can be formalized as:

$$u_t = tanh\left(Wh_t + b\right) \tag{9}$$

$$\alpha_t = \frac{exp(u^T u)}{\sum_t (exp(u_t^T u))} \tag{10}$$

where α_t is a normalized weight by attention mechanism. The specific model architecture is shown in FIGURE 4.

2) REENTRANCY DETECTION

In the previous subsection, we described our proposed BLSTM-ATT model which can be employed to perform reentrancy detection. We feed the feature vectors generated by *word2vec* into this sequential model to learn the model parameters. Learning involves computing the gradient during the back propagation phase and updating the model parameters. Once the training phase completed, we utilize the trained model to perform reentrancy detection.

Given one or multiple *contract snippets* of smart contracts from the testing set, we transform them into vector representations and feed vectors into the sequential model. The model outputs the results of each target smart contract to tell whether it is reentrancy ("1") or not ("0"). Formally, we use a softmax classifier to predict label y^* for a *contract snippet* S. The detector takes the hidden state h_* as input:

$$\hat{p}\left(y|S\right) = softmax\left(Wh^* + b\right) \tag{11}$$

$$y^* = \underset{y}{argmax}\, \hat{p}\left(y|S\right) \tag{12}$$

In summary, we can obtain the final results of the reentrancy identification by this detector.

V. EMPIRICAL EVALUATIONS

In this section, we conduct extensive experiments on real-world Ethereum smart contracts to evaluate our proposed *contract snippets* and sequential models. We start by introducing the evaluation metrics adopted in our experiments. Next, we present our datasets and processing. We then give details of how we trained the sequential models. Last, we analyze the experimental results and performance compared with state-of-the-art approaches.

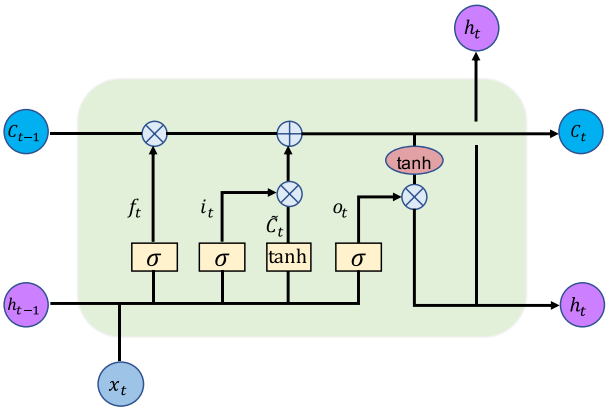


FIGURE 5. The structure of an LSTM unit.

A. EVALUATION METRICS

To evaluate the performance of our deep learning-based reentrancy detection models, we adopt the widely used metrics include accuracy (ACC), true positive rate or recall (TPR), false positive rate (FPR), precision (PRE), and F1-Score (F1). All these metrics are supported by true positive (TP): the number of samples with true reentrancy detected; false positive (FP): the number of samples with false reentrancy detected; false negative (FN): the number of samples with true reentrancy undetected; true negative (TN): the number of samples with false reentrancy undetected. Then, we can compute metrics as ACC $= \frac{TP+TN}{TP+TN+FP+FN}$, TPR $= \frac{TP}{TP+FN}$, FPR $= \frac{FP}{FP+TN}$, PRE $= \frac{TP}{TP+FP}$, and F1 $= \frac{2*PRE*TPR}{PRE+TPR}$. We prefer to achieving high TPR and low FPR. In addition, the main visualization technique that we study the performance of our reentrancy detection using the ROC curves, which a large area under the curve (i.e., AUC area) reveals a good detection performance.

B. DATASET AND PROCESSING

We conduct experiments on two datasets, RSC[1] and RCS. Specifically, we named the original smart contract dataset

[1]RSC is only used to verify the effectiveness of our proposed *contract snippets*, while basically experiments are conducted on the RCS.

as RSC and the normalized *contract snippet* dataset as RCS. RSC contains the original and unprocessed smart contracts, while RCS consists of normalized *contract snippets*, which will be further described in the upcoming paragraph.

RSC Focused on the Ethereum smart contracts, we crawled a total of over 42,000 contracts source code from the Etherscan [4] (Official platform of Ethereum) according to the addresses of smart contracts. As required by our detection task (i.e. reentrancy detection), we review all contract source code and differentiate over 2,000 contracts with *call.value*, which is the key point and potential causing reentrancy. We construct a dataset of reentrancy smart contracts based on original and unprocessed contracts (RSC) and give the ground truth for each contract as "1" (i.e., reentrancy) or "0" (i.e., non-reentrancy), which covers almost all the potential reentrancy contracts.

RCS To carry out a better empirical evaluation, we exploit our proposed approach assembling contract statements into *contract snippets*. A *contract snippet* consists of multiple lines of code that are semantically related to each other, especially those logical semantics closely related to reentrancy. Then, we scrutinize each *contract snippet* and label the ground truth as "1" (i.e., reentrancy) or "0" (i.e., non-reentrancy). Furthermore, in order to highlight the key features of a *contract snippet*, we transform *contract snippets* into their symbolic representations, mainly includes three steps as: (1) remove the blank lines, non-ASCII characters, and comments; (2) map user-defined variables to symbolic names (e.g., "VAR1", "VAR2"); (3) map user-defined functions to symbolic names (e.g., "FUN1", "FUN2"). As such, we can obtain a dataset of reentrancy *contract snippets* (RCS).

To transform *contract snippets* into the regular input-form of our sequential models, we vectorize them by the *word2vec* tool, which is widely used in word vector representation. Finally, we employ *word2vec* to embed the *contract snippets* into vectors of different dimensions for training so that can find the vector dimensions in the best performance situation.

C. TRAINING DETAILS FOR SEQUENTIAL MODELS

We implemented our proposed sequential model in Keras [31] with TensorFlow backend[2] running in Python. All the experiments are conducted on a computer equipped with an Intel Core i7 CPU at 3.7GHz, GPU at 1080Ti, and 32GB of Memory.

As to parameter settings, for each sequential model, we adopt 10-fold cross-validation to select and train the best parameter values corresponding to the effectiveness of reentrancy detection. We learned all the models by optimizing the binary cross-entropy loss. The optimal Algorithm *Adam* for gradient descent is employed for all experiments. The learning rate *lr* is searched in [0.0001, 0.0005, 0.001, 0.002, 0.005] for our models. To prevent overfitting, we tune the dropout rate *dr* as searched in [0.2, 0.4, 0.6, 0.8]. The batch size β of all methods is set to 64. Moreover, the number of tokens in the vector representation of *contract snippets* is set to 100, while the dimension of vectors *vm* is searched in [200, 300, 400, 500]. Without special mention in texts, we report the performance of all the models with following default setting: 1) $lr = 0.002$, 2) $dr = 0.2$, 3) $\beta = 64$, and 4) $\text{vm} = 300$.

[2]Our implementation is available: https://github.com/Messi-Q/VulDeeSmartContract

TABLE 1. The details of datasets RSC and RCS.

	Training set	Lines	Testing set	Lines
RSC	1,600	306,985	400	76,747
RCS	1,600	18,996	400	5,675

The general training details are concluded as follows: (1) we collect and use a total number of over 42,000 smart contracts; (2) we have differentiated 2,000 smart contracts with the keyword *call.value*, of which 400 are unique; (3) we then review 2,000 smart contracts and label the ground truth, including 666 with reentrancy vulnerability. (4) we construct two datasets (i.e., RSC and RCS) and divide them into an 8:2 training set and test set. Among them, the *original contracts* in the training set have 1,600 (306,985 lines) and in the testing set have 400 (76,747 lines), while the *contract snippets* are 18,996 lines and 5,675 lines, respectively. TABLE 1 reveals the specific details of the two datasets.

D. RESULTS AND PERFORMANCE COMPARISON

In this section, we illustrate the experimental results and performance comparison of our sequential models on smart contract reentrancy detection task. Our experiments are centered on the following three research questions.

RQ1: How effective are our sequential models, especially BLSTM-ATT, on reentrancy detection task? For answering this question, we conducted experiments on a variety of sequential models.

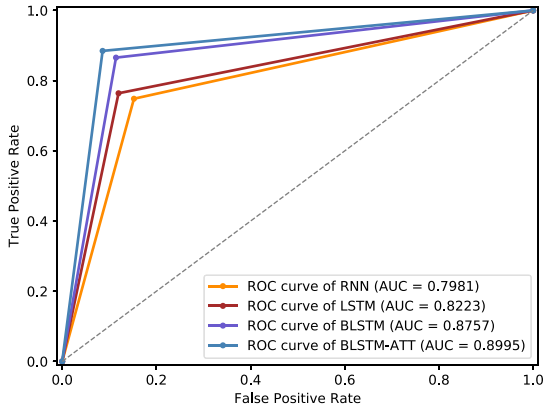
RQ2: How effective are our deep learning approaches when compared with state-of-the-art smart contract vulnerability detection tools? For answering this question, we compare our deep learning approaches with other existing detection methods, mainly including the formal analysis tools.

RQ3: How effective is our proposed representation method of *contract snippets* in our experiments? For answering this question, we conducted experiments on original and unprocessed smart contract source code (i.e., RSC dataset) compared with our smart *contract snippets* (i.e., RCS dataset).

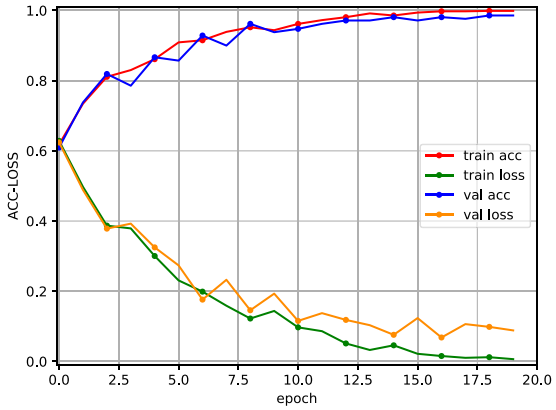
In what follows, we answer the above research questions one by one.

1) EMPIRICAL EVALUATION OF SEQUENTIAL MODELS (RQ1)

In section IV, we proposed using sequential models to detect smart contract reentrancy bugs, which is empirically evaluated in this section. The dataset contains over 2,000 *contract snippets* collected and normalized. We employ convincing metrics to explore the effect of deep learning-based



(a) ROC curves and AUC values of sequential models



(b) ACC-LOSS curves on BLSTM-ATT during training and validation

FIGURE 6. Performance visualization results of sequential models.

TABLE 2. Performance evaluation for Vanilla-RNN, LSTM, BLSTM, and BLSTM-ATT.

Models \ Metrics	ACC(%)	TPR(%)	FPR(%)	PRE(%)	F1(%)
Vanilla-RNN	79.81	74.85	15.24	82.92	78.56
LSTM	81.73	76.41	11.96	88.16	80.06
BLSTM	85.38	86.57	11.42	85.23	85.55
BLSTM-ATT	**88.47**	**88.48**	**8.57**	**88.50**	**88.26**

TABLE 3. Performance comparison with existing tools.

Models \ Metrics	ACC(%)	TPR(%)	FPR(%)	PRE(%)	F1(%)
Securify	53.50	37.41	**9.83**	**89.65**	52.79
Smartcheck	52.00	24.32	31.74	31.03	27.27
Mythril	60.00	39.21	18.36	68.96	50.00
Oyente	71.50	50.84	19.85	51.72	51.28
BLSTM-ATT	**89.52**	**92.38**	11.42	87.38	**89.81**

methods on the reentrancy detection task, in which the statistical results are shown in TABLE 2. As to each neural network model, we repeat experiments 10 times to calculate the average. It can be noted that almost sequential models achieve good performance and surpass the vanilla RNN, which can be attributed to its limitation at dealing with long sequences. In particular, BLSTM-ATT achieves a high F1-Score of 88.26% and a low false positive rate (FPR) of 8.57%, which means accurately identify the reentrancy bugs. The performance boosts are presumably from the effectiveness of the BLSTM architecture and the attention mechanism. The BLSTM-ATT model can not only capture the long-term dependencies in the past or future but also highlight the key point with the attention mechanism.

We further analyze the performance of our sequential models by ROC curve visualization as depicted in FIGURE 6(a) ROC curves plot the true positive rate (TPR) on the y-axis while the false positive rate (FPR) is shown on the x-axis, which usually used to study the performance of binary classifiers. The Area Under Curve (AUC) value of BLSTM-ATT reaches nearly 90%, which means a wonderful detection result. Additionally, as the training and validation process plotted in FIGURE 6(b), we report detection accuracy and loss of the sequential models on both the training and testing datasets. With the increase in the number of epochs, accuracy and loss corresponding increase or decrease.

From the results, we can conclude that the deep learning-based detection methods (i.e., sequential models) obviously achieve an impressive detection performance in terms of reentrancy bug. This implies that deep learning can be applied to vulnerability detection in smart contracts. In addition, due to the capture of the semantic information of sequential models and the highlight of the attention mechanism, smart contract vulnerability detection can achieve high accuracy and a low false positive rate.

2) COMPARISON WITH EXISTING METHODS (RQ2)

After empirically analyzing the effectiveness of sequential models, we now focus on the performance comparison of our detector w.r.t. existing state-of-the-art methods. Among the possible comparison methods, we consider one of the first tools for analyzing smart contracts (*Oyente*), a static analysis tool based on the bytecode level (*Securify*), a static analyzer based on source code (*Smartcheck*), and an automated analysis tool (*Mythril*). These methods are selected because they are automatic smart contract vulnerability detection tools that are closely related to our work.

To report the performance comparison results, we scan 200 subject smart contract instances (including 58 vulnerable reentrancy contracts) by these contract analyzers. Note that we have already classified all reentrancy warnings as true positive (TP) since *Securify* gives the report on vulnerability detection containing warnings. Additionally, we consider the situation that cannot be detected by current analyzers as false positive (FP). TABLE 3 has summarized the comparison results, in which we make the following observations. First, BLSTM-ATT substantially outperforms the other state-of-the-art smart contract vulnerability detection tools,

TABLE 4. Performance comparison for the experiment on datasets RSC and RCS.

RSC	ACC(%)	TPR(%)	FPR(%)	PRE(%)	F1(%)
Vanilla-RNN	58.57	54.28	42.85	55.88	55.07
LSTM	67.14	55.23	20.95	72.50	62.70
BLSTM	70.14	61.90	27.61	69.14	65.32
BLSTM-ATT	73.83	65.48	10.18	70.50	67.89
RCS	**ACC(%)**	**TPR(%)**	**FPR(%)**	**PRE(%)**	**F1(%)**
Vanilla-RNN	79.81	74.85	15.24	82.92	78.56
LSTM	81.73	76.41	11.96	88.16	80.06
BLSTM	85.38	86.57	11.42	85.23	85.55
BLSTM-ATT	**88.47**	**88.48**	**8.57**	**88.50**	**88.26**

because BLSTM-ATT incurs an FPR of 8.57% and an F1 of 88.26%. Second, we know that *Smartcheck* fails to detect certain vulnerabilities as it only depends on logic rules defined by human-experts, which lead to very low TPR and PRE (24.32% and 31.03%, respectively). As to *Mythril*, it reports slightly better than *Smartcheck*, but only 39.21% TPR and 68.96% PRE. Although *Oyente* also requires rules defined by human experts for recognizing reentrancy and is far from perfect, it utilizes data flow analysis so that it shows a TPR of 50.84% and a PRE of 51.72% in our experiments.

Unlike the other three tools, the results reported by *Securify* contain three parts, violations (i.e., ''1''), compliances (i.e., ''0''), and warnings. As such, after warnings treated as ''0'', this naturally leads to the improvement of its PRE. In spite of this, it still catches a low TPR of 37.41%, which leads to a low F1 of 52.39%.

Therefore, it is fair to say that our proposed BLSTM-ATT obviously outperforms four state-of-the-art formal analysis-based vulnerability detection tools. This leads that our sequential models achieve a more effective detection effect, which indicates deep learning-based methods not only is able to apply to smart contract vulnerability detection but also achieves good performance. Overall, in terms of experimental results, we use the sequential model far better than the state-of-the-art smart contract detection tools.

3) STUDY IN CONTRACT SNIPPETS (RQ3)

In this section, we mainly to verify that our proposed representation method of *contract snippets* plays a significant and valuable role. For this purpose, we make a comparison with the RSC dataset, which consists of the original and unprocessed smart contracts. Under the same background of model parameter settings, we conduct comparative experiments with different neural networks on the two datasets (i.e., RSC and RCS). TABLE 4 shows the specific experimental results of all sequential models.

Obviously, the results on RCS are far better than on RSC as shown in TABLE 4. Taking the results of BLSTM-ATT as an example, all evaluation metrics on RCS present much better results than those on RSC. Turn to the results on the RSC dataset, it only achieves the accuracy of 73.83% and the F1 of 67.89%. In particular, the results indicate a 14.64% higher ACC and 20.37% higher F1, respectively.

Conceptually, RSC dataset contains the original smart contracts, containing much irrelevant information, which prevents the sequential models adapting to the semantic information of smart contracts. Therefore, according to the experimental results on datasets RSC and RCS, we can attribute the good performance to the *contract snippets*, which actually abandon the useless information (e.g., code explanatory comments and blank lines) and capture key points (e.g., control flow dependency, keywords, and semantic inheritance information). With the highly-expressive *contract snippets*, our proposed sequential models tend to be well-adapted and well-trained to achieve precisely vulnerability identification.

VI. CONCLUSION AND FUTURE WORK

In this work, we put forward using sequential models for the reentrancy detection task, which applies the effectiveness of deep neural networks to smart contract vulnerability detection. The key *contract snippet* representations and sequential model (i.e., BLSTM-ATT) are proposed to learning more informative features at the source code level, even a *contract snippet* level. Extensive experiments on a real-world smart contract dataset have shown good results for the reentrancy vulnerability task. Furthermore, this is a successful practice of applying deep learning technology to smart contract vulnerability detection, which is able to promote future research interests in this area.

Besides, it needs to point out that our current work is limited to the detection task of reentrancy vulnerability. We tend to apply sequential models to more detection tasks of smart contract vulnerabilities in the follow-up work, such as integer overflow, unhandled exceptions, and so on. Furthermore, towards the dynamic detection of smart contract vulnerability related to the runtime state, we consider designing a general attack scenario to interact with each smart contract on runtime state, aiming to analyze the dynamic execution. We also hope that there will be more breakthroughs in this field in the future.

REFERENCES

[1] R. Schollmeier, ''A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications,'' in *Proc. 1st Int. Conf. Peer–Peer Comput.*, Aug. 2001, pp. 101–102.
[2] (2018). *Slowmist Hacked*. [Online]. Available: https://hacked.slowmist.io/en/
[3] (2015). *Ethereum*. [Online]. Available: https://github.com/ethereum/go-ethereum
[4] (2015). *Etherscan*. [Online]. Available: https://etherscan.io/
[5] (2018). *EOS*. [Online]. Available: https://eos.io/
[6] (2017). *TRON*. [Online]. Available: https://tron.network/
[7] (2016). *The Dao Attack*. [Online]. Available: https://en.wikipedia.org/wiki/TheDAO(organization).
[8] (2017). *Parity Multisig Bug*. [Online]. Available: http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/
[9] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Büenzli, and M. Vechev, ''Securify: Practical security analysis of smart contracts,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 67–82.

[10] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.
[11] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: Finding reentrancy bugs in smart contracts," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng., Companion*, May 2018, pp. 65–68.
[12] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, Sep. 2018, pp. 259–269.
[13] J. Wang, L.-C. Yu, K. R. Lai, and X. Zhang, "Dimensional sentiment analysis using a regional CNN-LSTM model," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, vol. 2, 2016, pp. 225–230.
[14] Y. Fan, X. Lu, D. Li, and Y. Liu, "Video-based emotion recognition using CNN-RNN and C3D hybrid networks," in *Proc. 18th ACM Int. Conf. Multimodal Interact.*, 2016, pp. 445–450.
[15] R. Messina and J. Louradour, "Segmentation-free handwritten Chinese text recognition with LSTM-RNN," in *Proc. 13th Int. Conf. Document Anal. Recognit. (ICDAR)*, Aug. 2015, pp. 171–175.
[16] A. Graves, A.-R. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 6645–6649.
[17] J. Schmidhuber and S. Hochreiter, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[18] K. Choi, G. Fazekas, and M. Sandler, "Text-based LSTM networks for automatic music composition," 2018, *arXiv:1604.05358*. [Online]. Available: https://arxiv.org/abs/1604.05358
[19] Y. Wu, C. Ting, and Z. Dabin, "Hierarchical attention mechanism and bidirectional long short-term memory based neural network model for smart contract automatic classification," *J. Comput. Appl.*, vol. 1, no. 1, pp. 1–9, 2019.
[20] P. Zhou, W. Shi, J. Tian, Z. Qi, B. Li, H. Hao, and B. Xu, "Attention-based bidirectional long short-term memory networks for relation classification," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, vol. 2, 2016, pp. 207–212.
[21] B. Mueller. (2017). *A Framework for Bug Hunting on the Ethereum Blockchain*. [Online]. Available: https://github.com/ConsenSys/mythril
[22] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of Ethereum smart contracts," in *Proc. IEEE/ACM 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, May/Jun. 2018, pp. 9–16.
[23] T. H. Kim, "A study of digital currency cryptography for business marketing and finance security," *Asia-Pacific J. Multimedia Services Convergent Art, Hum., Sociol.*, vol. 6, no. 1, pp. 365–376, 2016.
[24] Z. Huang, X. Su, Y. Zhang, C. Shi, H. Zhang, and L. Xie, "A decentralized solution for IoT data trusted exchange based-on blockchain," in *Proc. 3rd IEEE Int. Conf. Comput. Commun. (ICCC)*, Dec. 2017, pp. 1180–1184.
[25] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *25th Annu. Netw. Distrib. System Security Symp.*, San Diego, CA, USA, Feb. 2018, pp. 1–15.
[26] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 1345–1351.
[27] H. K. Dam, T. Tran, and T. M. Pham, "A deep language model for software code," in *Proc. Found. Softw. Eng. Int. Symp.*, 2016, pp. 1–4.
[28] X. Huo, M. Li, and Z.-H. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 1606–1612.
[29] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.
[30] A. Søgaard and Y. Goldberg, "Deep multi-task learning with low level tasks supervised at lower layers," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 231–235.
[31] (2015). *Keras*. [Online]. Available: https://keras.io/
[32] A. Graves, S. Fernández, and J. Schmidhuber, "Bidirectional LSTM networks for improved phoneme classification and recognition," in *Proc. Int. Conf. Artif. Neural Netw.*, 2005, pp. 799–804.
[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.
[34] M. Dhawan, "Analyzing Safety of Smart Contracts," in *Proc. Conf., Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2017, pp. 16–17.
[35] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 653–663.
[36] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "TBCNN: Tree-based convolutional neural network for programming language processing," 2014, *arXiv:1409.5718*. [Online]. Available: https://arxiv.org/abs/1409.5718
[37] J. Harer, O. Ozdemir, T. Lazovich, C. Reale, R. Russell, and L. Kim, "Learning to repair software vulnerabilities with generative adversarial networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 7933–7943.
[38] W. Yuxin, C. Ting, "Application of blockchain technology in the Internet of Things," *Mod. Comput.*, pp. 16–21, 2019.
[39] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the Internet of Things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.
[40] Z. Meng, T. Morizumi, S. Miyata, and H. Kinoshita, "Design scheme of copyright management system based on digital watermarking and blockchain," in *Proc. IEEE 42nd Annu. Comput. Softw. Appl. Conf.*, Jul. 2018, pp. 359–364.

PENG QIAN received the B.S. degree in software engineering from Yangtze University, in 2018. He is currently pursuing the master's degree with the School of Computer and Information Engineering, Zhejiang Gongshang University. His research interests include blockchain security and multimedia data analysis.

ZHENGUANG LIU received the B.E. degree from Shandong University and the Ph.D. degree from Zhejiang University, China, in 2010 and 2015, respectively. He was a Research Fellow with the National University of Singapore and the Singapore Agency for Science, Technology, and Research (A* STAR). He is currently a Professor of computer science with the School of Computer and Information Engineering, Zhejiang Gongshang University. His research interests include multimedia data analysis, data mining, and blockchain security.

QINMING HE received the B.S., M.S., and Ph.D. degrees in computer science from Zhejiang University, in 1985, 1988, and 2000, respectively. He is currently a Professor with the College of Computer Science, Zhejiang University, China. His research interests include data mining, computing system virtualization, and blockchain technology. His research has been supported by the National Natural Science Foundation of China (NSFC), the National Key Technology Research and Development Program, and so on.

ROGER ZIMMERMANN (Senior Member, IEEE) received the M.S. and Ph.D. degrees from the University of Southern California, in 1994 and 1998, respectively. He is currently an Associate Professor of computer science with the School of Computing, National University of Singapore (NUS), where he is also the Deputy Director with the Smart Systems Institute (SSI). He was Co-Director of the Centre of Social Media Innovations for Communities (COSMIC) and the Research Institute Funded by the National Research Foundation (NRF) of Singapore. Among his research interests are mobile video management, streaming media architectures, distributed and peer-to-peer systems, spatio-temporal data management, and location-based services.

XUN WANG (Member, IEEE) received the B.Sc. degree in mechanics and the Ph.D. degree in computer science from Zhejiang University, Hangzhou, China, in 1990 and 2006, respectively. He is currently a Professor with the School of Computer Science and Information Engineering, Zhejiang Gongshang University, China. His current research interests include mobile graphics computing, image/video processing, pattern recognition, intelligent information processing, and visualization. He is also a member of ACM and a Senior Member of CCF.