Vulnerable Code Clone Detection for Operating System through Correlation Induced Learning

Heyuan Shi, Runzhe Wang, Ying Fu, Yu Jiang, Jian Dong, Kun Tang and Jiaguang Sun

***Abstract*—Vulnerable code clones in the operating system (OS) threaten the safety of smart industrial environment, and most vulnerable OS code clone detection approaches neglect correlations between functions that limits the detection effectiveness. In this paper, we propose a two-phase framework to find vulnerable OS code clones by learning on correlations between functions. On the training phase, functions as the training set are extracted from the latest code repository and function features are derived by their AST structure. Then, external and internal correlations are explored by graph modeling of functions. Finally, the graph convolutional network for code clone detection (GCN-CC) is trained using function features and correlations. On the detection phase, functions in the to-be-detected OS code repository are extracted and the vulnerable OS code clones are detected by trained GCN-CC. We conduct experiments on 5 real OS code repositories, and experimental results show that our framework outperforms the state-of-the-art approaches.**

***Index Terms*—Cyber-physical security, vulnerable code clone detection, graph convolutional network, correlation modeling**

I. Introduction

At present, numerous industrial applications with operating system (OS) are deployed on smart industrial devices. And the smart industrial environments is projected to soon interconnect tens of billions of cyber-physical devices including both high-end devices and low-end devices with resource constraints, which use a general LinuxOS distributions, e.g., Ubuntu, openSUSE and FreeBSD, or embedded real-time operating system (RTOS) distributions, e.g., uCos and freeRTOS [1].

Though the deployment of OS reduces the costs of managing various applications which become more complex with the development of industrial devices, vulnerable code clones in OS threaten the safety of smart industrial environment [2]–[5]. Code clones, the exact or similar copies of code fragments within or between programs, is one of the serious code smells which can lead to latent bugs, vulnerabilities and security-critical problems [6]–[9]. And the patches to buggy code are often not cover all of the code clones in real OS distributions [10], and code clones appear more common in the industrial OS because its version is often stable but old. Even worse, the wide deployment of numerous cyber-physical devices in production makes the vulnerable code clones not only suffer the OS distribution itself but also make the whole industrial environments vulnerable since the operating systems and their utilities running on top of industrial devices as the key components. Therefore, it is important to find vulnerable code clones in OS code to ensure the security of cyber-physical devices in industrial environments.

Many researchers have proposed the vulnerable code clone detection approaches for the "big code" such as OS code. And the most of these approaches are based on the code abstraction comparison [10]–[13] or neural networks [9], [14], [15]. The former methods have three steps. The first step is to split a program into some kind of code fragments. The second step is to represent each code fragment in the abstract fashion. The third step is to compute the similarity between code fragments via abstract representations obtained in the second step. And the latter methods regard the vulnerable code clone detection as a binary classification (whether a program has code clone or not) or to determine which category of vulnerability a program belongs to. In this way, the raw feature of code is firstly extracted as the training set. And then, a classifier is trained to learn the end-to-end feature based on the training set. Finally, the vulnerable code clones are detected by the pre-trained classifier. However, these methods only focus on the feature of code fragment itself without considering the correlation between functions in OS code repository. As the results, missing of correlation consideration limits the performance and scope of these methods, e.g., it is difficult to detect the complex code clones such as code clones in Type III (e.g., deletion, insertion, and rearrangement of statements) and Type IV (semantic clones which are syntactically different but convey the same functionality) [16].

The numerous function calls in each version of OS distribution leads to the complex correlations between OS functions. And the correlations between functions in the code repository can help us to find vulnerable code clones, especially for the OS code because there are numerous functions in its various versions. In this paper, we propose a framework of vulnerable code clone detection to integrate correlations between functions and function features. The graph convolutional network is used as the detector, which is widely used in semi-supervised learning on the graph structure [17], [18] and shows the effectiveness in various classification tasks [19]–[21]. In particular, the framework contains two phases, i.e., training phase to train a detector of vulnerable code clone detection, and detection phase to find the vulnerable code clone. In the training phase, the OS functions related to the vulnerabilities are first extracted from the commits in OS code repository, which as the training set of the vulnerable functions and its fixed versions. And the OS functions in training

Manuscript received May 15, 2019; revised Jun 17, 2019; accepted Jun 27, 2019. (Corresponding author: Yu Jiang)

Heyuan Shi, Runzhe Wang, Ying Fu, Yu Jiang and Jiaguang Sun are with the School of Software, Tsinghua university, China.

Jian Dong is with the School of Computer Science and Engineering, Central South University, China.

Kun Tang is with the School of Electronic and Information Engineering, South China University of Technology, China.

set is abstracted and vectorized based on its AST structure. Then, the correlation is explored based on graph modeling of OS functions calls, and the similarity estimation based on correlations between functions is derived. Finally, the graph convolutional network for vulnerable code clone detection (GCN-CC) is trained by both function features and correlations to optimize the network parameters. In the detection phase, the network-parameters-optimized detector is used to find vulnerable code clone in to-be-detected functions. We conduct experiments on 5 code repositories of OS distribution. And the experimental results demonstrate the superior performance of the proposed framework in vulnerable code clone detection, compared to many state-of-the-art methods. The contributions are summarized as follows.

- A two-phase framework of vulnerable code clone detection for OS code is proposed, with the consideration of both function features and correlations between functions.
- The similarity estimation based on correlations between functions is derived by graph modeling of function calls.
- Experimental results show the superior performance of the proposed framework.

The rest of this paper is organized as follows. Section II introduce the existing methods related to code clone detection of OS code. The proposed vulnerable code clone framework is described in Section III. And the experimental datasets, settings of the proposed framework, compared approaches, results and discussions are reported in Section IV. Finally, we conclude this paper in Section V.

II. Related works

A. Code clone detection of big code

The code clone detection of OS code is related to detect code clones in "big code". Sajnani et. al. proposed a scalable clone detector called SourcerCC to detect Type I, II and III of code clones in large inter-project, with the granularity of the whole code repositories [13]. Based on the SourcerCC, Nishi et. al.proposed a token-based code clone detection approach which integrates filtering and verification, to optimize the runtime cost of code clone detection and extend the scalability [22]. Jiang et. al.proposed Deckard to find code clone as the granularity of file [23]. Based on Deckard, Gharehyazie et. al.detected code clone of the whole project repositories in GitHub, which verifies the scalability of Deckard [24]. Though these methods can detect code clones by computing the similarity of two abstracted code fragments, they cannot detect the vulnerable code clones since two code fragments with identical syntax abstract data structures do not necessarily contain the same vulnerability. Therefore, these methods are insensitive to judge patched and unpatched, or fixed and vulnerable code fragments.

B. Vulnerable code clone detection

Since the above methods of discovering code fragments with similar syntactic patterns cannot handle the vulnerable code clone detection, many researchers have proposed vulnerable code clone detection approaches for the big code based on code syntactic similarity comparison and neural networks.

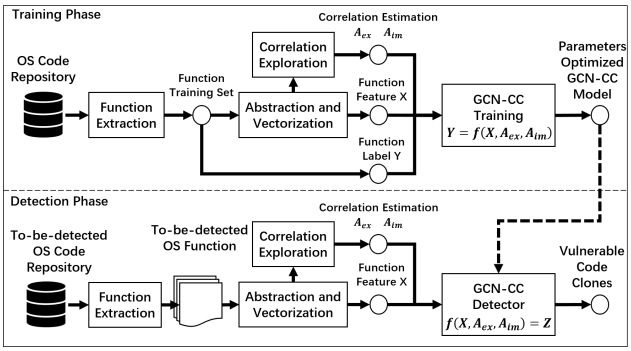


Fig. 1. The overview of the proposed framework.

As for the abstracted code fragments comparison, Jang et. al.proposed the first tool of vulnerable code clone detection specific to the OS distributions called ReDeBug [10], which uses the syntax-based normalization and tokenization to generate abstracted code fragments and a sliced window for similarity comparison. Kim et. al. proposed an approach for the scalable detection of vulnerable code clones called VUDDY, which exploits security-awared function abstraction with four levels and a hash function for fingerprint generation [11]. For the application of VUDDY, an automated analysis platform called IoTcube is designed to find code similarity and security vulnerabilities in the IoT devices [25]. Liu et. al. proposed a system called VFDETECT that can detect code clone of variable renaming, code sequence changing and redundancy inserting, which is also based on the fingerprints generation and abstraction [26].

For the neural-network-based approaches, Li et. al. proposed deep learning-based vulnerability detection system called VulDeePecker [14], which uses code gadgets to represent programs and the bidirectional long short term memory (BLSTM) neural network as the detector for vulnerable code clone detection. Lin et. al. also used the BLSTM to obtain a representation indicative of software vulnerability for vulnerability detection [27], while the code feature is derived by the serialized abstract syntax trees (ASTs) and vectorized by the Continuous Bag-of-Words neural embeddings. However, the neural-network-based methods neglect the correlation between code fragments, while both the code feature and correlations between code fragments are considered in the proposed framework.

III. The proposed framework

In this section, we propose the system overview and details of the proposed framework.

A. System overview

The overview of our framework is shown in Fig. 1. The framework contains two phases, i.e., the training phase and the detection phase. The training phase is to optimize the detector for vulnerable code clones and the detection phase is to detect vulnerable code clones in the to-be-detected OS functions by the optimized detector. In particular, on the training phase, the training set of OS functions is first extracted with their

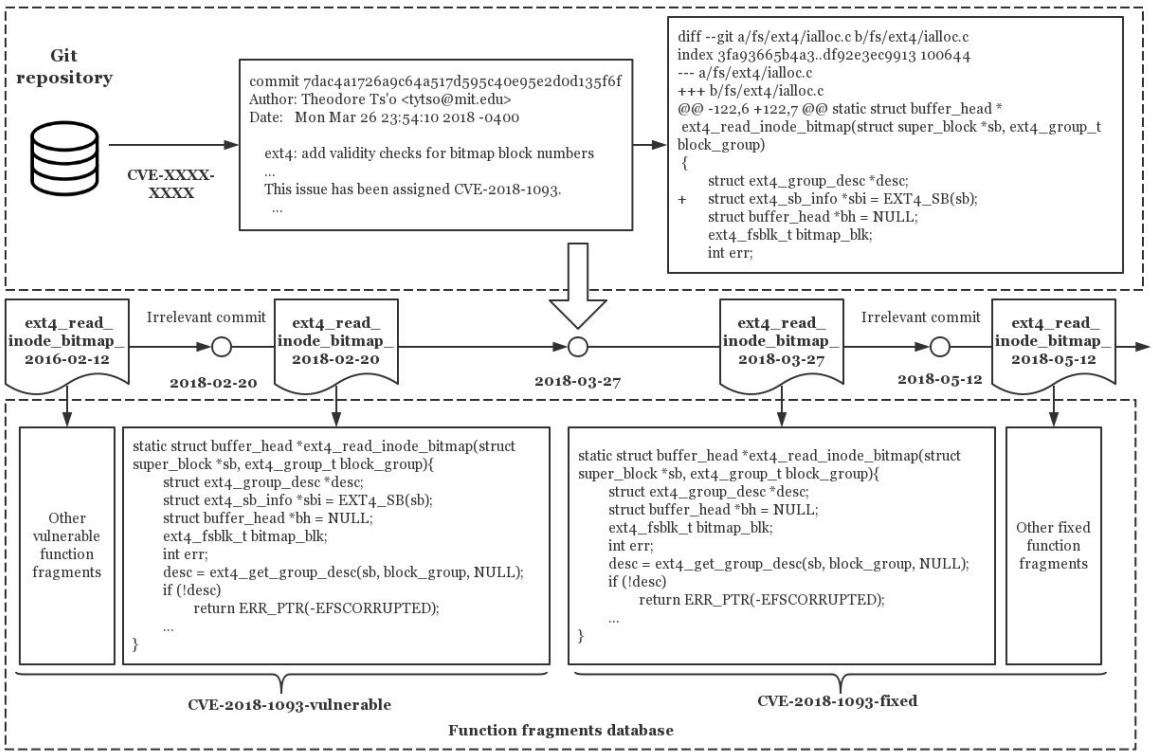


Fig. 2. Example of function extraction from code repository as training set.

TABLE I
NOTATIONS AND DEFINITIONS

Notation	Definition
$\mathcal{O}$	OS functions in the training set.
N	The number of functions in the training set.
$\mathbf{Y}$	Function labels.
M	The number of labels for OS functions in training set.
$\mathbf{X}$	Features of functions in training set.
$\mathcal{G}$	Graph of functions $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, W\}$
v	Vertices set which indicates functions on the graph.
$\mathcal{E}$	Edges set indicates function calls on the graph.
W	The weight of edges.
$\mathbf{A}_{ex}$	The adjacent matrix of explicit correlations.
$\mathbf{A}_{im}$	The adjacent matrix of implicit correlations.
$\mathbf{H}_w^{im}$	The matrix of input-to-fuse parameters.
$\mathbf{H}_w^{ex}$	The matrix of fuse-to-output parameters.
$\mathbf{Z}$	The outputs of GCN-CC.

vulnerability label from the commits in the OS code repository. And the AST structure of OS function in the training set is formulated. Then, the feature of functions is derived while the correlation estimation is explored on graph modeling of functions calls, based on the derived AST structure of OS functions. Finally, the derived feature, label and correlation estimation are integrated as the input to optimize the network parameters in the graph convolutional neural networks, which is used as the code clone detector. On the detection phase, the to-be-detected OS functions are also vectorized and correlation explored firstly. Then, the vulnerable code clone detection is performed by the detector which is optimized on the training phase. In the following, we give the details of each step with the formal description. The important notations and the corresponding definitions are summarized in Table I.

B. Function extraction from code repository

The code repository is regarded as the input of the proposed framework, e.g., git repository, since the OS code is usually maintained by the version control tools due to the complexity and scalability of programs. The example of function extraction is shown in Fig. 2.

We first extract the code fragment as the granularity of function from the commits in the code repository, by filtering the keywords of vulnerable code, i.e. CVE id. In this way, we first list all commits in code repository and select with the keyword "CVE-" to extract the CVE related commits. Then, for each keyword related commit, we check out all versions of the OS function and stored. For example, the patch for CVE-2018-1093 in the commit "7dac4a1726a9c64a517d595c40e95e2d0d135f6f" changes function $ext4\_read\_inode\_bitmap$, so all the revised versions of this function are stored, i.e., $ext4\_read\_inode\_bitmap$ changed in 12 Feb 2016, 20 Feb 2018, 27 Mar 2018 and 12 May 2018. Finally, the training set is extracted that consists of various versions of the vulnerability-related OS functions.

As for the label of functions in training set, the functions in the training set are labeled with the keyword and status

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2019.2929739, IEEE Transactions on Industrial Informatics

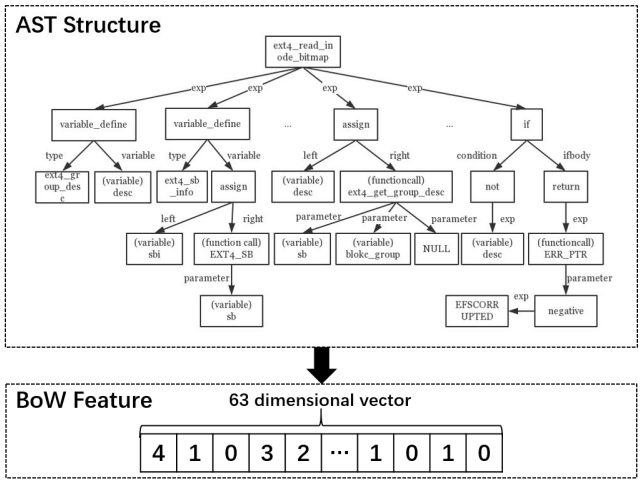


Fig. 3. Illustration of feature extraction based on AST structure.

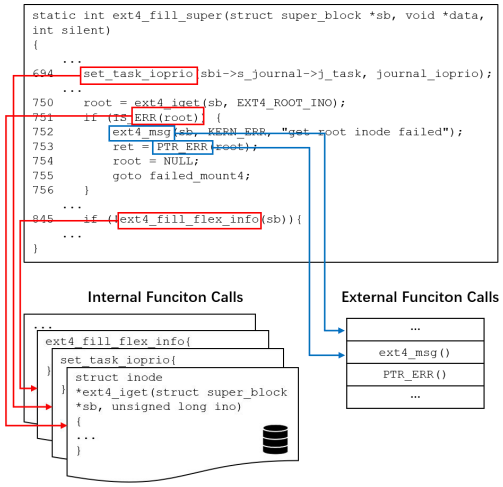


Fig. 4. Illustration of internal function calls and external function calls.

of vulnerability fix. In this way, the functions before the keyword related commits are tagged with "vulnerable" while the functions after that are tagged with "fixed". For example, the fragments of the function $ext4\_read\_inode\_bitmap$ are labeled as CVE-2018-1093 with "vulnerable" or "fixed" tag, and the versions before the patch for CVE-2018-1093 are labeled as "CVE-2018-1093-vulnerable" while the versions after that are labeled as "CVE-2018-1093-fixed".

By this step, the vulnerability related functions with labels are extracted from code repository as the training set for the proposed vulnerable code clone detection framework. For the convenience of descriptions in the following part, we denotes the training set contains N functions as $\mathcal{O} = \{\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_N\}$, where $\mathcal{O}_i$ is the i-th function in the training set. And we denote the label of training functions as $\mathbf{Y} = \{\mathbf{Y}_1, \mathbf{Y}_2, \cdots, \mathbf{Y}_M\}$ with one-hot encoding. Here, M is the number of labels for OS functions in training set, which equals 2 times (vulnerable and fixed tags) of the number of N_c keywords of vulnerability. And the keywords of vulnerability is denoted as $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_{N_c}\}$.

C. Function abstraction and vectorization

In this part, functions in the training set are parsed into the abstract syntax tree (AST) structure and vectorized with bag-of-words (BoW) model to generate the feature of functions.

We first exploit a lazy AST parser for our abstraction of function fragments, which includes 10 AST node types and 53 expression node types, and each node corresponds to a C code block or expression. The bag-of-words (BoW) model is used to generate the function feature based on its AST structure, which traverses each node of abstract syntax tree and count the number of AST and expression node types. As the result, we get the 63-dimensional vector of each function fragments by the bag-of-abstract-word based on AST structure, which ensures the robustness of function representation to common modifications in OS functions. For example, as illustrated in Fig. 3, the function $ext4\_read\_inode\_bitmap$ is parsed as the AST structure and vectorized by the BoW model.

By this step, for the training set of function $\mathcal{O} = \{\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_N\}$, the function feature is generated based on the AST structure of function and BoW model, which is denoted as $\mathbf{X} = \{\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_N\}$. Here, $\mathbf{X}_i \in \mathbf{X}$ is the feature of function $\mathcal{O}_i \in \mathcal{O}$, and the feature of function $\mathbf{X} \in \mathbb{R}^{N \times 63}$ is used as the one of the inputs for training the GCN-CC detector.

D. Function correlation exploration

In this step, the correlation between OS functions is explored based on graph modeling of function calls in the training set. In this way, both internal and external function calls in training set are extracted based on their AST structure firstly. And a graph is used to model the function calls. Then, two types of correlation called are explored on the graph with the form of adjacent matrix. Finally, the derived correlation estimations are used as the input for training GCN-CC model. In the following, we give the formal description of correlation exploration based on graph modeling of function calls.

We first extract function calls in the training set based on the AST structure of functions. And two types of function calls of internal function calls and external function calls are defined. The internal function call function shows which function in the training set is called by a function. And the external function call indicates the function out of the training set is called by a function. For example, as shown in Fig. 4, the function $ext4\_read\_inode\_bitmap$ calls 5 functions, with 3 internal function calls ($ext4\_fill\_flex\_info$, set_task_ioprio and $ext4\_iget$ are in the training set) and 2 external function calls ($ext4\_msg$ and PTR_ERR are out of training set).

Based on the two types of calls for functions in the training set, we use a graph model to formulate the correlation between functions. In this way, a graph of functions is denoted as $\mathcal{G} = \{\mathcal{V}, \mathcal{E}, W\}$, where $\mathcal{V}$ is a function set, $\mathcal{E}$ denotes an edge set that indicates calls between functions and W is the weight of edges. And for $v_i \in \mathcal{V}$, the internal function and external functions called by $v_i \in \mathcal{V}$ are denoted as $\mathcal{V}_{inl}(i)$ and $\mathcal{V}_{ext}(i)$, respectively.

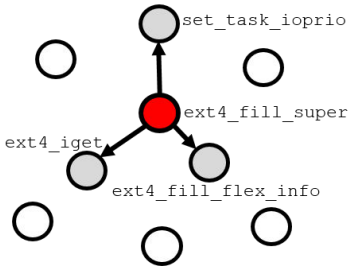


Fig. 5. Explicit correlation exploration between functions.

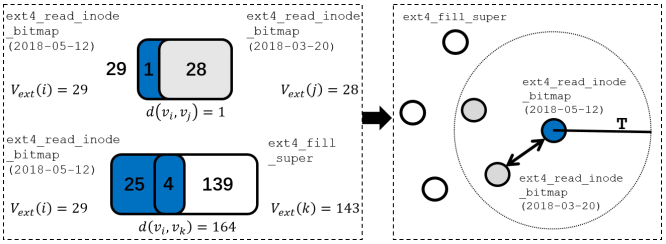


Fig. 6. Implicit correlation exploration between functions.

Based on the graph contains functions and their internal and external function calls, we explore the correlation between functions to derive the correlation estimation. The correlation estimation is formulated as an adjacent matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, and the correlation between two functions is denoted by the i, j-th entry of $\mathbf{A}$. Therefore, we have

$$\mathbf{A}(v_i, v_j) = \begin{cases} 1 & \text{if } v_i \in e, v_j \in e, \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

where $v_i, v_j \in \mathcal{V}$ and $e \in \mathcal{E}$. Moreover, we define the explicit correlation as the correlation estimation based on internal function calls, and the implicit correlation as the correlation estimation based on external function calls. Let $\mathbf{A}_{ex}$ and $\mathbf{A}_{im}$ denote the adjacent matrix that indicates the explicit and implicit correlation between functions. Now we introduce the correlation exploration on the graph model of function calls as follows.

To formulate the explicit correlation, for $v_i \in \mathcal{V}$, its internal function calls $\mathcal{V}_{inl}(i)$ are linked directly to the function v_i. As shown in Fig. 5, the internal functions of $ext4\_fill\_super$ are $ext4\_fill\_flex\_info$, set_task_ioprio and $ext4\_iget$, so $ext4\_fill\_super$ are linked to these internal functions, which indicates the explicit correlation corresponding to $ext4\_fill\_super$. Thus, for $v_i \in \mathcal{V}$, we have

$$\mathbf{A}_{ex}(v_i, v_j) = \begin{cases} 1 & \text{if } v_j \in \mathcal{V}_{inl}(i) \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

It is noted that the explicit correlation is unidirectional, which means for $v_i \in \mathcal{V}$ and $v_j \in \mathcal{V}_{inl}(i)$, the $\mathbf{A}_{exp}(i, j) = 1$ but $\mathbf{A}_{exp}(j, i) = 0$.

To formulate the implicit correlation, we exploit the one-order distance between functions based on their external function calls. For $v_i, v_j \in \mathcal{V}$, their external functions are $\mathcal{V}_{ext}(i)$ and $\mathcal{V}_{ext}(j)$, respectively. Let $d(v_i, v_j)$ denote the one-order distance between v_i and v_j, and we have

$$d(v_i, v_j) = |\mathcal{V}_{ext}(i) - \mathcal{V}_{ext}(j)|. \tag{3}$$

Moreover, we set a threshold $\mathbf{T}$ to determine the correlation estimation $\mathbf{A}_{im}$. Thus, for $v_i \in \mathcal{V}$ and $v_j \in \mathcal{V}_{ext}(i)$, we have

$$\mathbf{A}_{im}(v_i, v_j) = \begin{cases} 1 & \text{if } d(v_i, v_j) \leq \mathbf{T} \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

As shown in Fig. 6, for example, there are 28 external function calls for the version of function $ext4\_read\_inode\_bitmap$ on 20 Mar 2018, 29 external function calls for the version of function $ext4\_read\_inode\_bitmap$ on 12 May 2018, and 143 external function calls for the function $ext4\_fill\_super$. And for $ext4\_read\_inode\_bitmap$ with the version of 12 May 2018, the one-order distance to its version on 20 Mar 2018 and the function $ext4\_fill\_super$ is 1 and 164, respectively. Therefore, the two versions of $ext4\_read\_inode\_bitmap$ will be linked and regarded as implicit related when the threshold is set as 5. It is noted that the implicit correlation is bilateral, which means for $v_i \in \mathcal{V}$ and $v_j \in \mathcal{V}_{ext}(i)$, both $\mathbf{A}_{im}(i, j)$ and $\mathbf{A}_{im}(j, i)$ equal 1.

By this step, two correlation estimations, i.e., $\mathbf{A}_{ex}$ and $\mathbf{A}_{im}$, are explored by the graph modeling of function calls. And both $\mathbf{A}_{ex}$ and $\mathbf{A}_{im}$ are integrated with function feature $\mathbf{X}$ to train the GCN-CC detector in the next step.

E. Graph convolutional networks for code clone detection

In this step, both correlation estimations derived by correlation exploration and features of function, are integrated by the proposed GCN-CC model. In the training phase, the label of function is used to train the GCN-CC model and outputs the GCN-CC model with fine-optimized network parameters. In the detection phase, the to-be-detected functions features and their correlation descriptions are fed into the optimized GCN-CC detector, and the GCN-CC model outputs the predicted label with its vulnerability keywords.

The proposed GCN-CC model contains three layers, i.e., input layer, fuse layer, and output layer. And two convolution operations between layers are designed to tune the network parameter between layers. More specifically, the input-to-fuse convolution is conducted to optimize the network parameters by function feature, label and implicit correlation, and the fuse-to-output convolution is conducted to optimize network parameters by the function feature, label and explicit correlation. The implementation of the GCN-CC network structure is described as

$$f(\mathbf{X}, \mathbf{A}_{ex}, \mathbf{A}_{im}) = \text{softmax}(\mathbf{A}_{ex}\text{ReLU}(\mathbf{A}_{im}\mathbf{X}\mathbf{H}_w^{im})\mathbf{H}_w^{ex}),$$

where $\mathbf{H}_w^{im}$ and $\mathbf{H}_w^{ex}$ are to-be-tuned network parameters. Specifically, $\mathbf{H}_w^{im} \in \mathbb{R}^{D \times U}$ is the input-to-fuse parameters matrix, where U is the number of fused feature. $\mathbf{H}_w^{ex} \in \mathbb{R}^{U \times N_c}$ is the fuse-to-output parameters matrix. Moreover, ReLU and softmax are the activation function used for the input-to-fuse convolution and fuse-to-output convolution in GCN-CC, respectively. Specifically, $\text{ReLU}(\cdot)$ is defined as

$$\text{ReLU}(\cdot) = max(0, \cdot). \tag{5}$$

And softmax$(\cdot)$ is described as

$$\text{softmax}(\mathbf{Z}_i) = \frac{exp(\mathbf{Z}_i)}{\sum_{\mathbf{Z}_i \in \mathbf{Z}} \exp(\mathbf{Z}_i)}. \quad (6)$$

Here, $\mathbf{Z}_i \in \mathbb{R}^{1\times N_c}$ is the outputs of the GCN-CC for the i-th function as inputs, and $\mathbf{Z}_{i,j}$ determines the possibility that a function $\mathcal{O}_i \in \mathcal{O}$ belongs to the labels $\mathbf{Y}_j \in \mathbf{Y}$.

In the training phase, to tune the network parameters $\mathbf{H}_w^{im}$ and $\mathbf{H}_w^{ex}$, the label of functions as training set is used and we have

$$\mathbf{Y} = f(\mathbf{X}, \mathbf{A}_{ex}, \mathbf{A}_{im}). \quad (7)$$

The batch gradient descent is exploited to optimize $\mathbf{H}_w^{im}$ and $\mathbf{H}_w^{ex}$ for each iteration in the training phase. And the stochasticity is also considered to avoid the overfitting in the training, by adding a dropout layer between each pair of consecutive layers of the proposed GCN-CC. Moreover, we use the cross-entropy error between the labeled OS functions in the training set and predicted labels in the training phase, to optimize the network parameters. Let Err denote the cross-entropy error and we have

$$Err = -\sum_{p\in\mathcal{V}}\sum_{q=1}^{N_c} \mathbf{Y}_{p,q} \ln(\mathbf{Z}_{p,q}), \quad (8)$$

where $\mathbf{Y}_{p,q}$ denotes the function label associated with the p-th function belonging to the q-th label, and $\mathbf{Z}_{p,q}$ represents the possibility that the p-th function belongs to the q-th label detected by GCN-CC.

F. Vulnerable code clone detection

On the detection phase, the vulnerable code clone detection is performed by the GCN-CC which is optimized on the training phase. The preprocessing, i.e., abstraction, vectorization and correlations exploration, of the to-be-detected function is the same as that for the training functions. Then, the GCN-CC model with the optimized network parameters is used as the detector to find vulnerable code clones. In this way, the correlation estimation and feature of to-be-detected functions are exploited as the inputs of GCN-CC detector, and the detector outputs the possibility of a certain vulnerability in a to-be-detected function.

IV. Experiments

In this section, we first introduce the experimental datasets, settings, compared methods and metrics. Then, the experimental results and discussions are reported.

A. Dataset

We collected 5 git repositories of OS distributions as the experimental datasets, i.e., openSUSE, ubuntu-trusty, FreeBSD, OpenBSD and Linux kernel. The vulnerable and fixed functions are extracted by the keyword of CVE id. Table II summarizes the experimental datasets, where "#CVE id" is the number of CVE ids, "#CVE Label." is the number of CVE-related function labels. "#Vul." is the number of vulnerable functions and "#Fix." is the number of fixed functions related to CVE ids. We use the above OS distributions since these OS distributions are widely used in industrial practice and also included in the vulnerable code database of the compared methods.

TABLE II
Datasets of OS code repository used in experiments

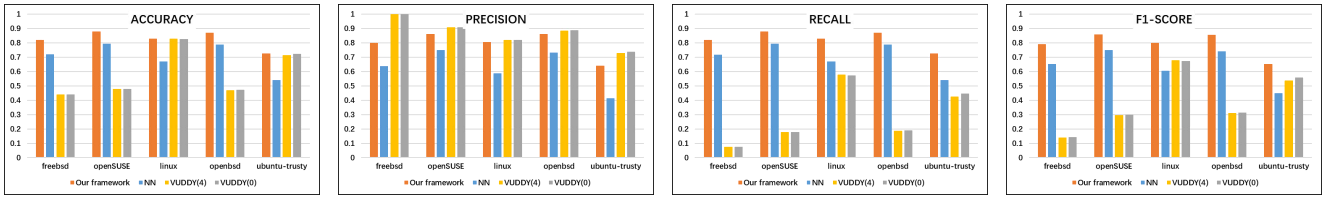
Repository	#CVE id	#CVE Label	#Vul.	#Fix.
FreeBSD	21	492	760	1127
openSUSE	206	717	3179	5415
Linuxkernel	171	540	2281	3684
OpenBSD	31	112	267	119
ubuntu-trusty	343	1187	2643	1650
Total	772	3048	9130	11995

B. Experimental Settings

1) Compared methods: We compared the proposed framework with VUDDY [11], [25] and LSTM neural network. VUDDY exploits the vulnerability-preserving abstraction to detect vulnerable code clones as the granularity of functions. Specifically, there are two stages of VUDDY, i.e., preprocessing and vulnerable clone detection. As for the preprocessing stage, vulnerable functions are firstly extracted from the code repository. And the syntactic abstraction with 4-level (formal parameter, local variable name, data type and function call) and normalization are performed to transform a function into a token series. Then, a 2-tuple fingerprint of function is generated and stored as the form of maps for retrieval. Then in the clone detection stage, VUDDY uses key and hash look-up between the vulnerable database and to-be-detected programs based on the generated fingerprint. It is noted that VUDDY with and without program abstraction schemes are both used in our comparison. LSTM neural network (NN) is widely used as the classifier for the vulnerable code clone detection [14], [27]. And the same function feature used in our framework is regarded as the inputs of NN. It is noted that other feature of function can be used for the comparison between the proposed framework and compared methods since the key of the proposed framework is to integrate function correlation and feature to improve the performance of vulnerable code clone detection.

2) Settings of the proposed framework: We now give the experimental settings of the proposed framework. For the correlation exploration, we set the threshold of implicit correlation determination as 1, which means the implicit correlation exists when the distance is less than 1 between any two OS functions. For the parameters GCN-CC model, the maps parameter to optimize the feature of function is set as the same as the number of labels $\mathbf{Y}$. The dropout layer is used between layers and the rate of dropout is set as 0.2. We train this network model for a maximum of 600 epochs (training iterations) using the Adam optimizer. The empirical learning rate is set as 0.01 for all datasets of code repository. Because the outputs of the proposed framework is a possibility distribution, we use the predicted label which possibility is maximal for each to-be-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2019.2929739, IEEE Transactions on Industrial Informatics



(a) Accuracy (b) Precision (c) Recall (d) F1-score

Fig. 7. Experimental results of vulnerable code clone detection of the compared methods.

detected function, as the predicted results with the following comparison in experiments.

3) Evaluation criteria: We use the widely used metrics, i.e., accuracy, precision, recall and F1-score, to evaluate the performance of the proposed framework compared with the state-of-the-art methods. To train the GCN-CC model, we randomly select 50% of functions per CVE-related label as the training set. And we select one more function per label as the validation set. The training will stop if the loss of validation set not decreases in 10 times in a row. We test the proposed framework using all the functions, because it is a common scenario to find both the unchanged vulnerable functions and the functions which are developed based on the vulnerable functions in practice.

C. Experimental results

The experimental results of the proposed framework compared with other methods are summarized in Fig. 7. And we make the following observations.

Firstly, the experimental results of accuracy show that the proposed framework outperforms the state-of-the-art approaches on most test cases of datasets. Moreover, the average accuracy on 5 datasets is 58.82%, 58.61%, 70.16% and 82.54%, for VUDDY without and with abstraction, LSTM methods and our framework, respectively. And the improvements of the proposed framework is 40.32% compared to VUDDY without abstraction, 40.81% compared to VUDDY with abstraction, and 17.64% compared with NN. The results indicate that the proposed framework is more effective in vulnerable code clone detection than other methods, and the better performance of the proposed framework is summarized as the following reasons. Compared with the state-of-the-art methods, the proposed framework finds the vulnerable code clones with the consideration of the correlation between OS functions, not only the function feature. Specifically, both the explicit and implicit correlation are integrated with GCN-CC model, whereas the NN only uses function feature to train the neural network, and VUDDY only takes the token series based on the parameter or statement abstraction within the function for comparison.

Second, the experimental results of precision and recall show that the proposed framework achieves relative balanced performance compared with VUDDY on most test cases of datasets. The methods of VUDDY without and with abstraction trade high precision for low recall. For example, the precision is 1 but the recall is 0.08 for VUDDY on the FreeBSD dataset, while the precision and recall is 0.79 and 0.82 for the proposed framework, respectively. It is noted that a vulnerable code clone detector with high false positive rates is not usable, and a detector with high false negative rates is not useful [14]. Therefore, VUDDY achieves the high false negative rates by the emphasis on low false positive rates, which limits the usability of vulnerable code clone detection. In this case, the proposed framework achieves both relatively low false negative rates and low false positive rates, which ensures useful and usable results of vulnerable code clone detection. Moreover, the imbalance between precision and recall leads to very low F1-measures. And for the average F1-score on 5 datasets, the proposed framework is 0.7921, while the VUDDY with abstraction is 0.3922, the VUDDY without abstraction is 0.3970, and the NN is 0.6389.

In summary, the proposed framework improves the performance of vulnerable code clone detection for OS by the consideration of the correlation between functions and feature of function.

TABLE III
AVERAGE RANKING COMPARISON WITH DIFFERENT CORRELATION INPUTS.

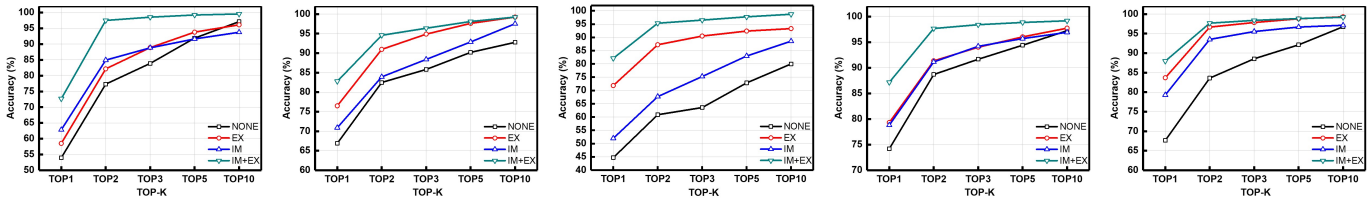
	FreeBSD	openSUSE	linux	OpenBSD	ubuntu-trusty
NONE	18.85	8.57	7.62	8.59	10.48
IM	13.34	8.27	7.03	2.54	7.51
EX	8.09	3.45	2.67	1.98	4.07
IM+EX	**2.13**	**1.63**	**1.90**	**1.66**	**1.78**

D. Discussions

In this part, we discuss the impact of correlation exploration and the form of outputs. To show the improvements by integrating the correlation consideration, we compared the proposed framework (IM+EX) with different correlation inputs, i.e., no correlation (NONE), explicit correlation only(EX) and implicit correlation only(IM). For this purpose, we change the inputs of correlation descriptions as two identity matrices, two explicit matrices and two implicit matrices for NONE, EX and IM, respectively. We report the average ranking and top-K accuracy of vulnerable code clone detection. The average ranking results of different inputs on 5 datasets are reported in TABLE III. And Fig. 8 shows the top-K accuracy of the proposed framework with various correlation inputs on 5 datasets.

1) On correlation exploration: From the reported results in TABLE III and Fig. 8, we can observe that the proposed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TII.2019.2929739, IEEE Transactions on Industrial Informatics



(a) FreeBSD (b) openSUSE (c) linux (d) OpenBSD (e) ubuntu-trusty

Fig. 8. Top K accuracy of the proposed framework on 5 datasets.

framework with both explicit and implicit correlation achieves the best performance compared with other correlation inputs, and the performance degrades when correlations are not integrated into the inputs. And the experimental results indicate that the correlation exploration and the integration of both explicit correlations, implicit correlation and feature is the key to improve the performance of the proposed framework. Moreover, the average ranking of the proposed framework considering both explicit and implicit correlations is less than 2. It indicates that the real vulnerable code clones can be expected to be found in the first two candidates from results on average, which is efficient for OS code developer and maintainer to reduce the costs of confirming whether a function has vulnerable code clones or not.

2) On probability distributional outputs: From Fig. 8, we make the following observation. First, the results indicate that the accuracy increases with K when it less than 3, and then the accuracy reaches a stable value. Second, the top-2 accuracy of vulnerable code clone detection by the proposed framework with both correlations is over 95%, which is usable and useful for developers and maintainer to find vulnerable code clones in OS code. Moreover, it indicates that the proposed framework can improve the performance by its outputs with the form of the possibility distribution, while the outputs of other token comparison based methods, e.g., VUDDY, cannot provide the candidate list for developer to confirm vulnerable code clones.

V. Conclusion

In this paper, we proposed a two-phase vulnerable code clone framework for OS in industrial environments. Specifically, in the training phase, we first extract the functions as the training set from the commits of the latest OS code repository. To derive the function feature, the OS functions are abstracted by their AST structure and vectorized based on the BoW model. To explore the correlation between functions, we explore the explicit and implicit correlation by graph modeling of function calls. Both function feature and correlation are integrated to optimize the network parameters of GCN-CC model, and the optimized model is used to detect the vulnerable code clone of OS code in the detection phase. The experiments conducted on 5 real code repository of OS distributions show that the proposed framework outperforms the state-of-the-art approaches. For future work, the more effective inputs of GCN-CC will be studied. The current function feature is based on the BoW model and AST structure, and correlations are explored by function calls. And more various function features and correlations will be added in our future work. Moreover, approaches such as target kernel fuzzing will integrate into the process of code clones confirmation to increase the automatic degree of vulnerable OS code clone detection.

References

[1] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the internet of things: A survey," *IEEE Internet of Things Journal*, vol. 3, pp. 720–734, 2016.

[2] H. Shi, R. Wang, Y. Fu, M. Wang, X. Shi, X. Jiao, H. Song, Y. Jiang, and J. Sun, "Industry practice of coverage-guided enterprise linux kernel fuzzing," in *ESEC/FSE*, 2019.

[3] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "Pafl: extend fuzzing optimizations of single mode to industrial parallel mode," in *ESEC/SIGSOFT FSE*, 2018.

[4] Y. Chen, Y. S. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, Z. Su, and X. Jiao, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[5] J. Liang, M. Wang, Y. Chen, Y. Jiang, and R. Zhang, "Fuzz testing in practice: Obstacles and solutions," *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 562–566, 2018.

[6] H. Liu, Z. Yang, Y. Jiang, W. Zhao, and J. Sun, "Enabling clone detection for ethereum via smart contract birthmarks," in *ICPC*, 2019.

[7] H. Liu, Z. Yang, C. Liu, Y. Jiang, W. Zhao, and J. Sun, "Eclone: detect semantic clones in ethereum via symbolic transaction sketch," in *ESEC/SIGSOFT FSE*, 2018.

[8] M. S. Rahman and C. K. Roy, "On the relationships between stability and bug-proneness of code clones: An empirical study," *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 131–140, 2017.

[9] J. Gao, X. Yang, Y. Fu, Y. Jiang, H. Shi, and J. Sun, "Vulseeker-pro: enhanced semantic learning based binary vulnerability seeker with emulation," in *ESEC/SIGSOFT FSE*, 2018.

[10] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," *2012 IEEE Symposium on Security and Privacy*, pp. 48–62, 2012.

[11] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 595–614, 2017.

[12] V. Saini, H. Sajnani, J. Kim, and C. V. Lopes, "Sourcerercc and sourcerercc-i: Tools to detect clones in batch mode and during software development," *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 597–600, 2016.

[13] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 1157–1168, 2016.

[14] 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," *CoRR*, vol. abs/1801.01681, 2018.

[15] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary," in *ASE*, 2018.

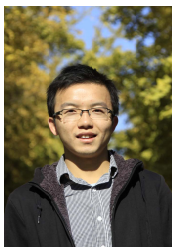
[16] D. Rattan, R. K. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, pp. 1165–1199, 2013.

[17] W. Wang, Y. Yan, F. Nie, S. Yan, and N. Sebe, "Flexible manifold learning with optimal graph for image and video representation," *IEEE Transactions on Image Processing*, vol. 27, pp. 2664–2675, 2018.
[18] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2017.
[19] M. S. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *ESWC*, 2018.
[20] H. Shi, X. Zhao, H. Wan, H. Wang, J. Dong, K. W. Tang, and A. Liu, "Multi-model induced network for participatory-sensing-based classification tasks in intelligent and connected transportation systems," *Computer Networks*, vol. 141, pp. 157–165, 2018.
[21] H. Shi, Y. Zhang, Z. Zhang, N. Ma, X. Zhao, Y. Gao, and J. Sun, "Hypergraph-induced convolutional networks for visual classification." *IEEE transactions on neural networks and learning systems*, 2018.
[22] M. A. Nishi and K. Damevski, "Scalable code clone detection and search based on adaptive prefix filtering," *Journal of Systems and Software*, vol. 137, pp. 130–142, 2018.
[23] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105, 2007.
[24] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github," *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 291–301, 2017.
[25] S. Kim, S. Woo, H. Lee, and H. Oh, "Iotcube: An automated analysis platform for finding security vulnerabilities," 2017.
[26] Z. Liu, Q. Wei, and Y. Cao, "Vfdetect: A vulnerable code clone detection system based on vulnerability fingerprint," *2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEC)*, pp. 548–553, 2017.
[27] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. D. Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, pp. 3289–3297, 2018.

Heyuan Shi received the BS degree in school of information science and engineering, Central South University, Changsha, China, in 2015. He is currently working toward the Ph.D degree in with School of Software, Tsinghua University, Beijing, China. His current research interests include software safety and security of embedded system, Internet of things and operating systems.

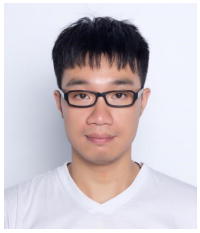
Runzhe Wang received the BS degree in school of software engineering, Beijing University of Posts and Telecommunications, Beijing, China, in 2017. He is currently working toward the Master degree in with school of software, Tsinghua University, Beijing, China. His current research interests include software safety and security of embedded system, Internet of things and operating systems.

Ying Fu received the BS degree in school of software engineering, Beijing University of Posts and Telecommunications, Beijing, China, in 2017. She is currently working toward the Master degree in with school of software, Tsinghua University, Beijing, China. Her current research interests include software safety and security of binary program, blockchain systems.



Yu Jiang received the BS degree in software engineering from Beijing University of post and telecommunication, Beijing, China, in 2010, and the PhD degree in computer science from Tsinghua University, Beijing, China, in 2015. He is currently an Assistant Professor in School of Software, Tsinghua University, China. His current research interests include domain specific modeling, formal computation model, formal verification and their applications in embedded systems, safety analysis and assurance of cyber-physical system.

Jian Dong received the B.S. degree in electrical engineering from Hunan University, Changsha, China, in 2004, and the Ph.D. degree in electrical engineering from Huazhong University of Science and Technology (HUST), Wuhan, China, in 2010. Since September 2010, He has been an Associate Professor with the School of Information Science and Engineering, Central South University, Changsha, China. His current research interests include wireless communications, sensor networks, and antennas.



Kun Tang received the B.S. degree in telecommunications from Wuhan University of Technology, Wuhan, China, in 2006, and M.S. degree in The University of New South Wales, Sydney, Australia, in 2011, and Ph.D. degree in Telecommunications from the Central South University, Changsha, China, in 2018. He is now a postdoctor with the school of Electronic and Information at South China University of Technology. His research interests are in the areas of cognitive radio networks, sensor networks and Massive MIMO.

Jiaguang Sun received the BS degree in automation science from Tsinghua University in 1970. He is currently a professor in Tsinghua University. He is dedicated in teaching and R&D activities in computer graphics, computer-aided design, formal verification of software, and system architecture. He is currently the director of the School of Information Science & Technology and the School of Software in Tsinghua University.