Semantic Learning Based Cross-Platform Binary Vulnerability Search For IoT Devices

Jian Gao, Xin Yang, Yu Jiang, Houbing Song, Kim-Kwang Raymond Choo, Jiaguang Sun

***Abstract*—The rapid development of Internet of things (IoT) has triggered more security requirements than ever, especially in detecting vulnerabilities in various IoT devices. The widely used clone-based vulnerability search methods are effective on source code, however their performance is limited in IoT binary search.**

In this paper, we present *IoTSeeker*, a function semantic learning based vulnerability search approach for cross-platform IoT binary. First, we construct the function semantic graph to capture both the data flow and control flow information, and encode lightweight semantic features of each basic block within the semantic graph as numerical vectors. Then, the embedding vector of the whole binary function is generated by feeding the numerical vectors of basic blocks to our customized semantics aware neural network model. Finally, the cosine distance of two embedding vectors is calculated to determine whether a binary function contains a known vulnerability. The experiments show that *IoTSeeker* outperforms the state-of-the-art approaches for identifying cross-platform IoT binary vulnerabilities. For example, compared to *Gemini*, *IoTSeeker* finds 12.68% more vulnerabilities in the top-50 candidates, and improves the value of AUC for 8.23%.

***Index Terms*—function semantic learning, cross-platform binary, vulnerability search, IoT devices, neural network.**

I. Introduction

OVER the past decade, the number of IoT devices around the world has increased dramatically. *Gartner*, Inc. forecasts that 26 billion connected things will be in use worldwide by the year 2020 [1], [2]. Embedded system software is the core of controlling IoT devices such as smart grid systems, industrial control systems and medical devices, and its security is particularly important [3]. Software in billions of IoT devices have similarly-functional code and widely use third-party libraries. This means that code clone plays an important role in the development of embedded software, and is widely used by programmers to improve the software production efficiency. Study shows that 22.3% code of Linux kernel of the IoT devices is from previous implementation [4].

For many IoT devices, security is an afterthought, leading to unknown vulnerabilities introduced by the code clone are increasingly common. For example, the Debian operating system of IoT devices confirms 145 unpatched clone vulnerabilities [5]. In addition, with the popularity of IoT devices, software programs on traditional X86 architecture are gradually being ported to other architectures (e.g., ARM, MIPS) with different compiler configurations. Correspondingly, vulnerabilities are also propagated to IoT devices of different platforms. It is envisaged that when attackers have mastered the exploitation of a specific vulnerability, they can make use of it to attack different platforms, which is very dangerous and may lead to system failure. For example, a compromised Haier SmartCare automation system containing vulnerabilities can lead to property damage, as an attacker can remotely hijack the functionality of the appliances driven by the IoT device [6].

Many methods have been proposed to solve the cross-platform IoT binary vulnerability search problem. For example, *Genius* [7] presents a learning-based approach to generate robust platform-independent function feature vector. The similarity of two functions is transformed into the distance between two function feature vectors. *BinGo* [8] first captures the input-output relations of length variant partial traces to create a signature model for the function, and then uses *Jaccard* containment similarity [9] to measure the similarity score between two different function signature models.

Although these methods have achieved certain results in vulnerability search, they show two limitations in complex search scenarios for large-scale code [10]. The first is that insufficient and inaccurate function semantic information is captured, leading to a high false positive rate. For example, *BinGo* [8] relies heavily on the control flow graphs (CFGs) to generate function signature model. However, the CFGs are significantly divergent across different implementation platforms, resulting in that its cross-platform clone detection accuracy is less than 60%. The second is that most methods require high computational overhead, making them difficult to apply to complex large-scale binaries. For example, *Genius* [7] requires the spectral clustering algorithm to generate a codebook and inefficient graph matching algorithms in search. *Gemini* [11] employs the deep learning model to reduce the time cost, but it only considers the CFGs of functions, and loses a large part of the function semantic information.

To overcome the above two limitations and strike a better balance between accuracy and efficiency, we present *IoTSeeker*, a function semantic learning based cross-platform binary vulnerability search approach for the IoT devices. Through integrating the DFG and the CFG of the function to a labeled semantic flow graph (LSFG), we adequately capture the semantic information of a function. We select 8 types

Manuscript received May 14, 2018; revised Sep 10, 2019; accepted Sep 20, 2019. This research is sponsored in part by the NSFC Program (No. 61527812), the National Science and Technology Major Project of China (No. 2016ZX01038101). Corresponding author: Yu Jiang (e-mail: jy1989@mail.tsinghua.edu.cn)

Jian Gao, Xin Yang, Yu Jiang and Jiaguang Sun are with the School of Software, Tsinghua University, China, also with Beijing National Research Center for Information Science and Technology, China, and also with Key Laboratory for Information System Security, Ministry of Education, China.

Houbing Song is with the Department of Electrical and Computer Engineering, West Virginia University, USA.

Kim-Kwang Raymond Choo is with the Department of Information Systems and Cyber Security, University of Texas at San Antonio, USA.

of features of basic blocks based on empirically study. Then the embedding vector of the function is generated by feeding features of basic blocks of the function to a customized neural network model. We determine the similarity of two functions by calculating the distance of their embedding vectors. This method supports training a generic model for code clone detection, and also supports the fine-tuning of the generic model for a specific vulnerability search to obtain a better precision.

For evaluation, we compare *IoTSeeker* with the state-of-the-art cross-platform binary clone vulnerability search tools on some widely used real world embedded system software in IoT devices. The experimental results show that *IoTSeeker* outperforms the most recent and related tools *BinGo* [8], *Genius* [7] and *Gemini* [11], and can accurately recognize the similar cross-platform binary functions and detect more vulnerabilities. In detail, during a complex search scenario, the AUC value of *IoTSeeker* is 0.8849, which is 11.62%, 14.71% and 8.23% higher than that of *BinGo*, *Genius* and *Gemini*, respectively. Furthermore, we use two vulnerabilities (CVE-2015-1791, CVE-2014-3508) to evaluate the vulnerability search capability of *IoTSeeker* in 4643 firmware images. For these two vulnerabilities, we found 39 and 41 real vulnerabilities respectively in the top-50 results, which are at least 3 and at most 31 more vulnerability instances than any of the comparison tools. The time cost of vulnerability search for *IoTSeeker* is about 0.2s per function, and 1.7s, 1.57s, 0.15s for *BinGo*, *Genius* and *Gemini*.

The following summarizes main contributions of this article:

- As far as we know, *IoTSeeker* is the first that integrates both the CFG and the DFG together and applies neural network to automatically capture more binary function semantics than existing state-of-the-art approaches.
- The high search accuracy and low time cost of *IoTSeeker* demonstrate its potential on effectively identifying vulnerabilities from large scale binaries of IoT devices.

The rest of this paper is organized as follows. Section II details the overall design of our IoT binary search approach, including semantic flow graph construction, semantic feature extraction and customized neural network model suitable for IoT binary. Section III describes our experimental results compared to the state-of-the-art approaches and studies optimal hyper-parameters. And Section IV presents the conclusion.

II. Design of *IoTSeeker*

Our objective is to design a vulnerability search approach that can automatically tell whether a given binary program from IoT devices contains clone vulnerabilities or not. Lots of vulnerability-related databases are open to the public, such as *Common Vulnerabilities and Exposures* (CVE) [12]. There are more than 120,000 vulnerabilities in CVE, some of which occurs within functions and provides corresponding source code. As a result, the clone vulnerability search method at the function-level granularity will help to find the known vulnerabilities in target cross-platform binaries.

In this section, we describe the design of *IoTSeeker*. It consists of three phases, the ***learning*** phase, the ***fine-tuning*** phase and the ***vulnerability search*** phase. Each phase is completed by three identical steps: labeled semantic flow graph construction, semantic feature extraction and *IoTSeeker* deep neural network. The overall architecture is shown in Fig. 1.

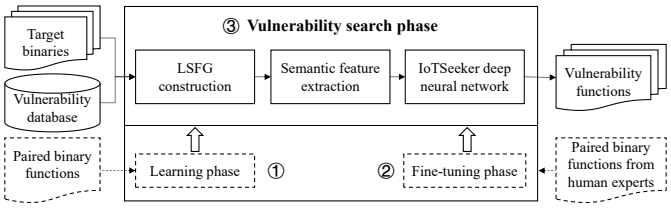
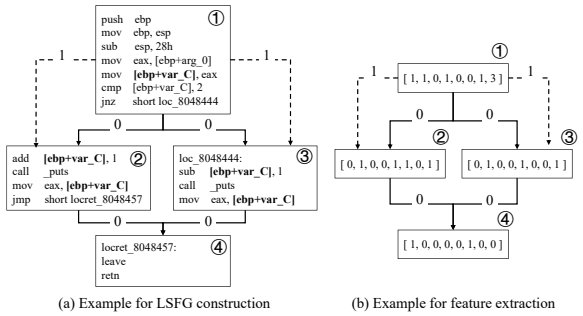


Fig. 1. Overall workflow of *IoTSeeker*.

The inputs to the *learning* phase are similar or dissimilar function pairs, which come from binary programs compiled under different platforms (e.g., architectures, compiling configurations). The output of the *learning* phase is a generic neural network model for binary function clone detection, which is the basis for subsequent model fine-tuning. Although the generic model can be used to search for the clone vulnerabilities directly, its effectiveness is not satisfactory because the training set may not contain vulnerability samples. So in the *fine-tuning* phase, some vulnerability samples from human experts are fed to the generic model to retrain a fine-tuned model for the specific vulnerability search. The generic model can be flexibly retrained multiple times for different types of vulnerabilities. The *vulnerability search* phase uses the fine-tuned model to search vulnerable functions in binaries of IoT devices. The following is a detailed introduction to the three identical steps for each phase.

A. Labeled Semantic Flow Graph Construction

Function semantics cannot be obtained by simply parsing functions into syntax trees or CFGs [13], [14]. It usually requires a higher level of abstract computation. Most methods compute a set of symbolic formulas representing the input-output relations of a basic block based on the CFGs [15]–[17]. Function semantics obtained through these methods can be significantly different in the face of changes in CFGs across compilation options [18].



(a) Example for LSFG construction (b) Example for feature extraction

Fig. 2. An example of labeled semantic flow graph construction.

We propose the labeled semantic flow graph (LSFG) which combines the control flow graph (CFG) and the data flow graph

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.2019.2947432, IEEE Transactions on Industrial Informatics

TABLE I
BASIC-BLOCK LEVEL FEATURES USED BY *IoTSeeker*.

Feature Name	Example
No. of stack operation instructions	push, pop
No. of arithmetic instructions	add, sub
No. of logical instructions	and, or
No. of comparative instructions	test
No. of library function calls	call prinf
No. of unconditional jump instructions	jmp
No. of conditional jump instructions	jne, jb
No. of generic instructions	mov, lea

(DFG) to extract the function semantics. The idea is based on the fact that the CFG determines the possible execution sequences of basic blocks and the DFG depicts the transfer and use of data within the function. By combining two kinds of dependency relations together, we can get more complete function semantics, which will be more immune to structural and syntactic differences introduced by the CFGs under d-ifferent architectures and compilation optimization strategies. Fig. 2(a) illustrates an example of the LSFG. The solid lines marked as 0 represent the control dependency, while the dotted lines marked as 1 represent the data dependency.

For a compiled binary function, the structures of the CFG obtained by different methods are almost the same, while the DFG differs according to different data dependency rules. We construct the DFGs on top of CFGs by leveraging the *define-use* rules and traversing all function paths. Specifically, for two instructions i and j from two different basic blocks which meet the CFG topology, if the instruction i writes to a memory location and the instruction j reads from the same memory location, we create a data dependent edge for these two basic blocks. It is worth noting that there is at most one data dependent edge between two different basic blocks. Therefore, the presence of memory address "$[ebp + var\_C]$" forms a data dependent edge between the block ① and ② in Fig. 2(a).

B. Semantic Feature Extraction

The LSFG constructed above is not suitable to be directly input into the function semantic learning model. We should choose and extract some robust initial numerical features that should change little under various implementation platforms with different microprocessor architectures and various compilation optimization configurations. By empirically analyzing the binaries from various platforms and referring to features used in previous works [7], [11], [19], we have finally determined to use those 8 types of features shown in Table I as the initial semantic representation of each basic block.

We first count the number of each feature in each basic block, then arrange them into numerical vectors in order, and finally put these numerical vectors to the corresponding vertex of LSFG. Fig. 2(b) is the numerical vectors of each basic block corresponding to the function in Fig. 2(a). We denote the LSFG with numerical vectors as $g = \langle X, C, D \rangle$, where X, C and D are the sets of (basic block) vertices, control dependent edges and data dependent edges, respectively. And each vertex $x_i \in X$ represents the initial numerical feature vector. The LSFG mentioned later refers to the LSFG with initial numerical feature vectors unless otherwise specified. Paired LSFGs g_1, g_2 are the input of *IoTSeeker* neural network.

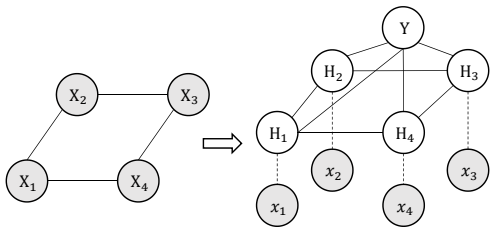


Fig. 3. The graph model of the *structure2vec* approach. There would be multiple hidden layers on the right, but only one layer is drawn for illustration.

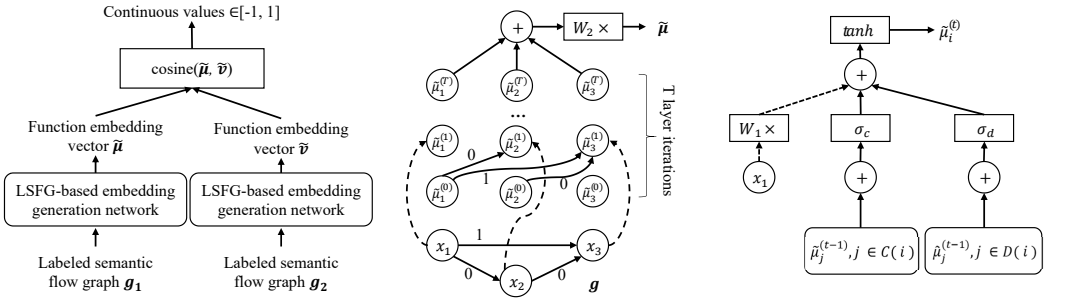
C. IoTSeeker Neural Network

The neural network model is the most important part to complete feature integration and vulnerability search. Deep learning models such as convolutional neural network (CNN) for image recognition and recurrent neural network (RNN) for natural language processing cannot be used directly for vulnerability search, because their input is usually a picture or a textual description. However, our input should be a pair of functions (more specifically, a pair of LSFGs) and the output is their degree of similarity. To apply to our vulnerability search scenario, the neural network that is dedicated to graph topology is required. We will first introduce two basic neural networks, and then present how to coalesce them to get the *IoTSeeker* neural network model.

1) Basic Siamese Neural Network: The *Siamese* neural network [20] solves the dual input problem of verification of signatures written on a pen-input tablet. It mainly consists of two identical sub-networks. Each sub-network takes a processed signature as its input, then output the feature of the signature. The joining neuron is used to measure the distance between these two output features of the two sub-networks, and output the similarity value ranging from -1 to 1. In *IoTSeeker*, *Siamese* is used as a network framework to train two identical embedded sub-networks in an end-to-end manner, which ensures that the two sub-networks share the same parameters. So, the final effect is that the vector representations of similar functions are close to each other.

2) Basic structure2vec Neural Network: Since functions are represented as graphs, it is critical to learn accurate function semantics from the graphs. We notice that the vertices in the graph are not isolated, but are connected to each other by the edges. Dai *et al.* [21] proposed *structure2vec* graph neural network which is an effective and scalable approach for structured data representation. It can encode vertex features and connection relationship of the edges in the graph as the embedding vector to represent function semantics. Fig. 3 shows how *structure2vec* works to generate such embedding vector based on the graph topology on the left. The example graph contains four vertices $X_i (i \in \{1, 2, 3, 4\})$, and each vertex is appended with an initial numerical vector x_i. The right side of Fig. 3 is the corresponding *structure2vec* network

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.2019.2947432, IEEE Transactions on Industrial Informatics



(a) The network model of IoTSeeker for vulnerability search

(b) LSFG-based embedding generation network

(c) A schematic diagram for generating the embedding vector of each vertex during t-layer iteration.

Fig. 4. The deep neural network of *IoTSeeker* for vulnerability search.

structure that consists of one input layer, T hidden layers (only one hidden layer is drawn for illustration) and one output layer. In *structure2vec*, the number of neuron nodes in the input layer and each hidden layer is the number of vertices in the original graph g.

In the input layer, the input of each neuron node is the initial numerical vector x_i of the corresponding vertex. In the t^{th} $(1 \leq t \leq T)$ hidden layer, each hidden neuron H_i is responsible for generating a new feature representation called the embedding vector $\widetilde{\mu}_i^{(t)}$, which encodes the initial numerical vector x_i and t-hop interactions between vertex features. We use formula $\widetilde{\mu}_i^{(t)} = F(x_i, \Sigma_{k \in E(i)} \widetilde{\mu}_k^{(t-1)})$ to represent this mapping relationship, where $E(i)$ refers to the set of vertices adjacent to vertex X_i. Through the mapping function F, the feature of each vertex is propagated to other vertices based on connected edges. This feature update strategy takes into account the graph topology and ensure that each vertex within the graph has corresponding T-hop context information after T hidden layer iterations. Finally, the output layer neuron Y aggregates the output embedding vectors $\widetilde{\mu}_i^{(T)}$ of the T^{th} hidden layer neurons to form the embedding vector (function semantic feature representation) of the entire graph.

3) Extended IoTSeeker Neural Network: The embedding vector generation strategy of the *structure2vec* is especially suitable for our LSFG, which simplifies the vulnerability search problem to calculate the similarity of function embedding vector. In our vulnerability search scenario, our goal is to determine whether given two LSFGs g_1, g_2 are similar. The label y in the training sample $\langle g_1, g_2, y \rangle$ indicates whether the two graphs are similar. This dual-input and single-output sample requires us to train two *structure2vec* networks with shared parameters, each of which is responsible for one LSFG. So, we embed two identical *structure2vec* networks into *Siamese* framework to achieve this. Since the *structure2vec* is implemented to support the LSFG, we call it LSFG-based embedding generation network.

Network structure: The overall architecture of *IoTSeeker* neural network model is showed in Fig. 4(a). Each LSFG-based embedding generation network takes a LSFG g as input and output its embedding vector $\widetilde{\mu}$ which captures function semantics by taking into account the structured graph information. The *Cosine* function is used to calculate the similarity of two embedding vectors (e.g., $\widetilde{\mu}$, $\widetilde{\nu}$), which represents the similarity of two binary functions.

Fig. 4(b) is a detailed description of the LSFG-based embedding generation network. This process is similar to the basic *structure2vec* except for the graph type. We extend *structure2vec* to deal with labeled semantic flow graph (LSFG) containing both data flow (labeled 1 on the solid arrow) and control flow (labeled 0 on the solid arrow), while basic *structure2vec* works with an ordinary undirected graph. The example LSFG $g = \langle X, C, D \rangle$ consists of three vertices X_i, $i \in \{1, 2, 3\}$, each of which represents a basic block of the function and contains the initial feature vector x_i as the network input. $C(i)$ and $D(i)$ represent the control dependent edge set and data dependent edge set of vertex i, respectively.

The extended *structure2vec* network contains a total of T hidden layer responsible for transforming graph information into function semantic embedding vector. Each hidden layer node is represented as the updated embedding vector $\widetilde{\mu}_i^{(t)}$, where different t values correspond to different hidden layer. During the t^{th} hidden layer iteration, the updated $\widetilde{\mu}_i^{(t)}$ consists of three different inputs: initial feature vector x_i of the corresponding vertex X_i (the dotted arrow in Fig. 4), the sum $l_c^{(t-1)} = \Sigma_{j \in C(i)} \widetilde{\mu}_j^{(t-1)}$ of previous embedding vectors of vertices pointing to X_i through the control dependency $C(i)$, and the sum $l_d^{(t-1)} = \Sigma_{j \in D(i)} \widetilde{\mu}_j^{(t-1)}$ of previous embedding vectors of vertices pointing to X_i through the data dependency $D(i)$. The function mapping F can be expressed by the formula: $\widetilde{\mu}_i^{(t)} = F(x_i, l_c^{(t-1)}, l_d^{(t-1)}) = tanh(W_1 x_i + \sigma_c(l_c^{(t-1)}) + \sigma_d(l_d^{(t-1)}))$. Fig. 4(c) illustrates the schematic diagram for generating the embedding vector $\widetilde{\mu}_i^{(t)}$ of each hidden layer node, where σ_c, σ_d are two non-linear transformation functions which are responsible for calculating an embedding vector with more powerful representation capability. Similar to [11], we define them as two n layer fully-connected networks with the following forms:

$$\begin{cases} \sigma_c(l_c) = P_1 \times ReLU(P_2 \times \cdots ReLU(P_n \times l_c)) \\ \sigma_d(l_d) = Q_1 \times ReLU(Q_2 \times \cdots ReLU(Q_n \times l_d)) \end{cases} \quad (1)$$

where n is the embedding depth of each vertex, P_i and Q_i are $p \times p$ dimensional parameter matrixes for every layer of the two fully-connected networks.

The overall procedure for generating function semantic representation described above is implemented and integrated in Algorithm 1. W_1 and W_2 are $d \times p$ and $p \times p$ dimensional

parameter matrixes respectively. Through the T-layer iteration in Algorithm 1 from line 5 to 11, a new feature representation of each vertex is generated, which not only follows the topology structure of LSFG, but also integrates the T-hop interaction among vertices. In other words, the features of the vertices are propagated to other vertices as the iteration progresses, ensuring that each basic block within the function has corresponding context information. Finally, the binary function semantics, including the data flow dependency and the control flow dependency, is aggregated into the corresponding embedding vector $\widetilde{\mu}$ in Line 12.

Algorithm 1: Algorithm for generating function semantics

Input: LSFG $g = \langle V, E, x \rangle$
Hidden layer iteration number T
Output: Binary function semantics embedding vector $\widetilde{\mu}$

1 $C(i)$ as the set of parent nodes that are the control dependency of vertex i; $D(i)$ as the set of parent nodes that are the data dependency of vertex i.
2 **for** $i \in V$ **do**
3 $\quad \widetilde{\mu}_i^{(0)} = 0$
4 **end**
5 **for** $t = 1$ *to* T **do**
6 $\quad$ **for** $i \in V$ **do**
7 $\quad\quad l_c^{(t-1)} = \Sigma_{j \in C(i)} \widetilde{\mu}_j^{(t-1)}$
8 $\quad\quad l_d^{(t-1)} = \Sigma_{j \in D(i)} \widetilde{\mu}_j^{(t-1)}$
9 $\quad\quad \widetilde{\mu}_i{}^{(t)} = tanh(W_1 x_i + \sigma_c(l_c^{(t-1)}) + \sigma_d(l_d^{(t-1)}))$
10 $\quad$ **end**
11 **end**
12 return $\widetilde{\mu} = W_2(\Sigma_{i \in V} \widetilde{\mu}_i^{(T)})$

Learning parameters: Paired embedding vectors (e.g., $\widetilde{\mu}$, $\widetilde{\nu}$) are obtained through two identical LSFG-based embedding generation networks with two LSFGs (e.g., g_1, g_2) as inputs. The output of the whole network represents the similarity of the two functions and is measured by the *Cosine* function denoted as $\widehat{y} = cos(\widetilde{u}, \widetilde{v}) = (\widetilde{u} \cdot \widetilde{v})/(\|\widetilde{u}\| \cdot \|\widetilde{v}\|)$, where $\widehat{y}$ is the prediction output of the similarity of two functions, ranging from -1 to 1. Given the ground truth $y \in \{1, -1\}$ of embedding vectors $\widetilde{u}$ and $\widetilde{v}$, where $y = 1$ indicates they are similar functions, otherwise dissimilar. Suppose that the training data set has M pairs of labeled samples $\langle f_1, f_2, y \rangle$, then our training objective is to minimize the training errors. We can use stochastic gradient descent algorithm to minimize error function $min\, E(W_1, W_2, P_1 \cdots Pn, Q_1 \cdots Q_n) = \sum_{m=1}^{M} (\widehat{y} - y)^2$ and obtain the most appropriate model parameters (e.g., W_1, P_1). After the generic model is trained, specific vulnerability samples from human experts can be optionally fed to the generic neural network model to retrain a fine-tuned neural network model for the specific vulnerability search.

III. Experimental Evaluation

IoTSeeker is based on code similarity detection to complete binary code vulnerability search in IoT devices. Therefore, our experiments are centered at the following two questions:

- RQ1. What accuracy can *IoTSeeker* achieve in predicting similar cross-platform binary functions of IoT devices when compared with other approaches?
- RQ2. How many real known vulnerabilities can *IoTSeeker* identify in existing large code bases from embedded system software of IoT devices when compared with other approaches?

A. Experiment Setup

All experiments were performed on a server with an Intel Xeon E5 running at 2.2GHz, 64GB memory and 2 NVIDIA GeForce Titan X GPUs.

Data Preparation. We prepare two datasets to complete different evaluation tasks shown in Table II. The following is a brief description:

- **Dataset I**. We choose five open-source projects commonly used on IoT devices. Then we use two compilers to compile these five projects into six target processor architecture binaries under four optimization options. As a result, we get a total of 240 binary versions containing 735,540 functions and more than 9,345K basic blocks. There are 1,222,950 pairs of samples constructed in this dataset.
- **Dataset II**. We adopt the same firmware image dataset from IoT devices (such as IP cameras, routers and access points) used by [7], [11]. We use this dataset to evaluate the effectiveness and the efficiency of *IoTSeeker* on real vulnerability search. It consists of 4,643 successfully disassembled firmware images.

TABLE II
DESCRIPTIONS OF THE DATASET.

Type	#Function	#Project	Source
Dataset I	735,540	240	Programs:OpenSSL(v1.0.1f, v1.0.1u), BusyBox(v1.21.0), Coreutils(v6.5, v6.7) Compilers: GCC(v4.9),GCC(v5.5) Optimization levels: O0-O3 Processors: X86, X64, MIPS32, MIPS64, ARM32 and ARM64
Dataset II	25,202,314	4,643	Firmware images from IoT devices (such as IP cameras, routers and access points)

Comparison Tools. We compare *IoTSeeker* with three state-of-the-art tools: *BinGo* [8], *Genius* [7] and *Gemini* [11]. *BinGo* is a traditional CFG-based technique that generates function model by extracting length variant partial trace and compares them to detect similar functions. *Genius* is a spectral clustering based technique that uses bipartite graph matching algorithm to generate function vectors and then perform function comparison. *Gemini* is a learning-based technique that compares embedding vectors which are learned from CFGs to predict function similarity. For fair comparison, we reimplement the other three tools, and train *Genius* and *Gemini* on our dataset according to their configuration.

Ground Truth. We use the following strategies to automatically label the training samples in dataset I for the generic neural network model of the learning phase. With the source code of function f, we compile it into a set of binary functions denoted as $set(f) = \{f_1, f_2, \cdots, f_n\}$ across different compilation options. For each function f_i in $set(f)$, we randomly select a different function $f_j, i \neq j$ to make

up a similar sample, and label them as $\langle f_i, f_j, +1\rangle$. We also randomly select another binary function s_k that is not in $set(f)$ to construct a dissimilar sample, and label them as $\langle f_i, s_k, -1\rangle$. A total of 1,222,950 pairs of samples are constructed. We apply 10-fold cross-validation to train and evaluate *IoTSeeker*. Namely, we partition the samples into 10 subsets, each time nine subsets are used to train model, and one subset is chosen as test set. We repeat this 10 times and each time the picked test subset is different. The reported result is averaged over the 10 times.

For a specific vulnerability search task, to get a higher prediction accuracy, we can optionally provide a small number of additional samples from human experts to fine-tune the generic neural network model in the fine-tuning phase. In this paper, we compile each vulnerability f into 240 binary versions and construct 240 pairs of similar samples $\langle f_i, f_j, 1\rangle$ and 240 pairs of dissimilar samples $\langle f_i, s_k, -1\rangle$. We use the augmented dataset which consists of the 480 samples and 1% randomly selected samples of dataset I to retrain a fine-tuned model for 5 epochs. The advantage of this is that we only need a model to accurately predict whether a particular vulnerability is included in any compiled version of the binary.

Default Training Configuration. Following those related previous works, we set up the hyper-parameters of the model as follows: the training epoch is 100, the learning rate is 0.0001, the embedding depth n is 2, the embedding size p is 64, the number of iterations T for each basic block is 6 and the size of mini-batch is 10. After finishing each epoch, we randomly shuffle the training set.

B. Accuracy of Code Clone

We answer RQ1 about what accuracy our approach can achieve in identifying similar binary functions of IoT devices across different platforms. We use *BinGo*, *Genius* and the generic models of both *IoTSeeker* and *Gemini* to perform this experiment on dataset I. Since we adopt 10-fold cross-validation method to train models and there are 10 different test sets in total, we conduct experiments on each test set separately and take the average of 10 experiments finally.

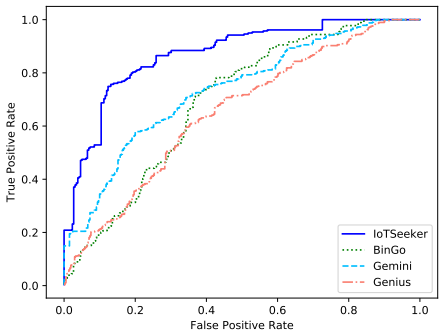


Fig. 5. ROC curves of code clone for different approaches.

Fig. 5 is the accuracy of code clone shown with the ROC (receiver operating characteristic) curves. The area enclosed by the ROC curve and the x-axis is expressed as AUC (area under the curve), which is equivalent to the probability that a randomly chosen positive example is ranked higher than a randomly chosen negative example [22]. Therefore, classifiers with larger AUC values have better generalization performance. We can see that the curve drawn by *IoTSeeker* is above the curves drawn by other three tools, which means that we can achieve a higher true positive rate with a lower false positive rate. The AUC values of *BinGo*, *Genius*, *Gemini* and *IoTSeeker* are 0.7928, 0.7714, 0.8176 and 0.8849, respectively. The main reason is that in addition to the CFG, we also construct the DFG for tracking the usages of variables between basic blocks. By the 6-layer iteration during generating the function embedding vector, features of basic blocks propagate 6-hops forward. Therefore, each basic block holds the semantic information hidden in the LSFG, which ultimately results in the effective identification of clone functions.

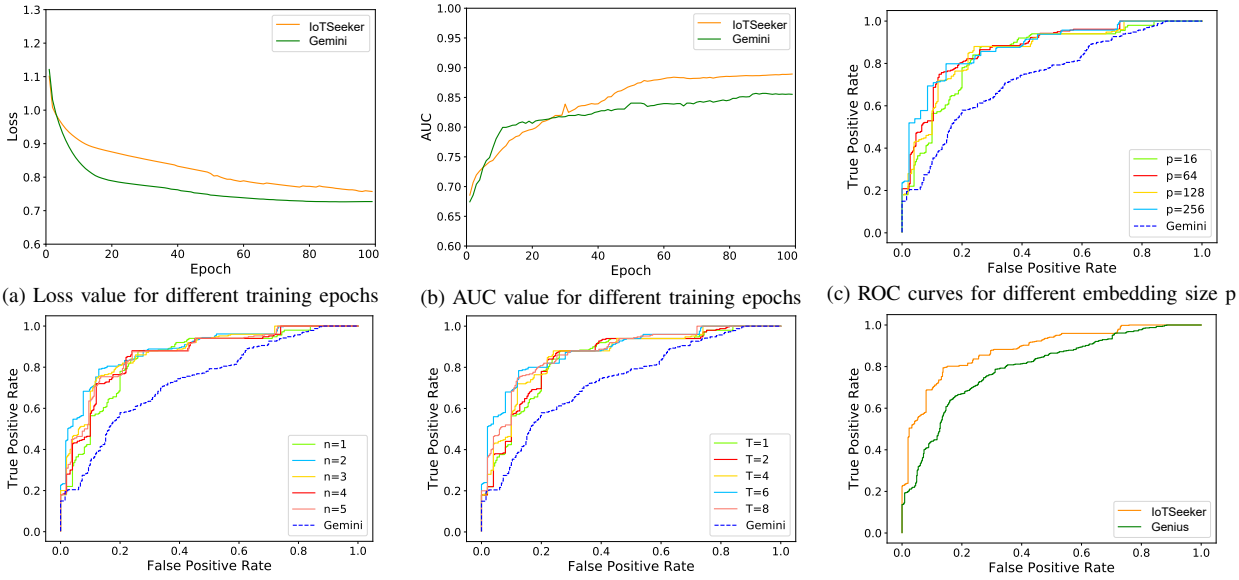
For *Gemini* and *IoTSeeker* in some less complex scenarios, the accuracy of *Gemini* and *IoTSeeker* can be much higher than the results presented in Fig. 5. For example, in the dataset settings of [11], *Gemini*'s AUC value of the pre-trained model is 0.971 and ours is about 0.984, which is much higher than the value of 0.8849 and 0.8176 tested in the presented experiments. Two reasons lead to this situation. One is that our dataset I contains 5 different programs, but the dataset of *Gemini* only contains 2 of them. The other is that we compile these programs into six architectures, including three 64-bit ones, and *Gemini* only compiles them into three 32-bit ones. The number of general-purpose registers used in 32-bit and 64-bit architectures is different, which affects the feature vectors of basic blocks, and its results drop accordingly. We found that more complex the dataset is, more improvements *IoTSeeker* would achieve.

C. Hyper-parameter Studies

We evaluate the impacts of different hyper-parameters on the accuracy of code clone with dataset I. Due to the time and resource limitation, we train each model with the training set for 50 epochs. All the hyper-parameters are evaluated on the test set. In the study of each hyper-parameter, except for the parameter being evaluated, other parameters take default settings as described in Section III-A. We summarize all the investigated hyper-parameters of *IoTSeeker* in Table III. Values that enable *IoTSeeker* to achieve the best predictive capability are bolded in Column **Values**. Column **Metric** represents the evaluation criteria used to select the most appropriate hyper-parameter values.

1) Size of Training Epochs: Increasing training epochs results in more times of weight updating. However, it is pointless to increase the number of training epochs blindly, which may not substantially change the parameter values of the model. So we want to know when the performance of the model tends to be stable. In total, we train the model for 100 epochs and evaluate the loss value and the AUC value on the validation set for every epoch. Fig. 6(a) and Fig. 6(b) show the loss value and the AUC value respectively. We can see that our approach has achieved a good performance in about 50 epochs, the AUC value is 0.87, and the loss value is 0.83. The loss values for both approaches are below 1.2. As the

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.2019.2947432, IEEE Transactions on Industrial Informatics



(a) Loss value for different training epochs (b) AUC value for different training epochs (c) ROC curves for different embedding size p

(d) ROC curves for the embedding depth n (e) ROC curves for the number of iterations T (f) ROC curves for semantic feature selection

Fig. 6. Hyper-parameter studies results on dataset I. Fig. 6(a) and Fig. 6(b) show the effects of training epochs. Fig. 6(c-f) describe the predictive effects of the generic model with different choices of the embedding size, the embedding depth, the number of iterations and feature sets. When studying each hyper-parameter, the choices of other hyper-parameters are the optimal setting described in Section III-A. With the variation of embedding size, the embedding depth and the number of iterations, the AUC value of *IoTSeeker* remains higher than that of *Gemini*. The reason of temporary exception in Fig. 6(b) is that our model contains more parameters and needs more epochs for training in the early stage.

TABLE III
THE SUMMARY OF ALL INVESTIGATED HYPER-PARAMETERS IN *IoTSeeker*.

Parameter	Values	Metric	Description
Epoch	20,40,60, 80,**100**	Loss, AUC	The number of rounds for model training
embedding size p	16,**64**, 128,256	ROC	The dimension of the embedding vector used to represent the function semantics
embedding depth n	1,**2**,3, 4,5	ROC	The number of layers of two fully-connected networks in σ_c and σ_d
iteration T	1,2,4, **6**,8	ROC	The number of hidden layers in the LSFG-based embedding generation network
feature selection	**IoTSeeker**, Genius	ROC	Determine which initial feature set is suitable for vulnerability search

number of epochs increases we eventually have a higher AUC value than *Gemini*, even though our AUC value is lower in about the first 30 epochs. The main reason is that our model contains more parameters and needs more epochs for training in the early stage.

2) Embedding Size: The embedding size refers to the dimension of the embedding vector used to represent the function semantics. We use the ROC curves to evaluate which embedding size can achieve the best performance. Fig. 6(c) plots the experimental results. When the embedding size is greater than 64, their corresponding ROC curves are close to each other. We choose 64 as the default embedding size, because it can reduce the time cost of training and prediction. Whatever embedding size is set up for *IoTSeeker*, the AUC value is higher that that of *Gemini* with the optimal settings. This phenomenon also applies to embedding depth and the number of iterations.

3) Embedding Depth: The embedding depth refers to the number of layers of two fully-connected networks represented as σ_c, σ_d. Fig. 6(d) shows the effects of varying embedding depth. The relatively good AUC value is obtained when the embedding depth is 2. This means that by increasing a two-layer full-connected network, the generated embedding vectors have stronger representation capabilities and better capture the function semantics. However, when the embedding depth exceeds 2, there will be no benefit other than higher time cost of training and prediction.

4) Number of Iterations: It refers to the number of hidden layers in the LSFG-based embedding generation network in Fig. 4(b). We vary the number of iterations T, results of ROC curves are drawn in Fig. 6(e). When the number of iterations is 6, our approach achieves the best performance of code clone. This means that the feature vector of each vertex in LSFG can propagate 6-hops along the topological structure of LSFG.

5) Features Selection: Related researches have proposed several sets of binary function features for predicting code similarity in machine learning methods [7], [19]. By observing and analyzing the characteristics of different binaries compiled across different platforms, we propose a set of features that are suitable for performing code similarity prediction tasks. Section II-B introduces these features in detail. We use dataset I to verify the performance of different feature sets.

Based on the default configuration detailed in Section III-A, we use different feature sets to train our models on the train set, then evaluate the effects on the test set. Fig. 6(f) shows the performance of different feature sets on dataset I using ROC curves. We observe that the feature set we choose has the best performance compared to other feature sets on *IoTSeeker*. The AUC value for our feature set is 0.8849, which is higher than just using the feature set of *Genius*.

TABLE IV
ACCURACY OF VULNERABILITY SEARCH IN THE TOP-K REPORTED RESULTS. N@K MEANS #NUM@K HERE.

	CVE-2015-1791				CVE-2014-3508			
	#N@1	#N@10	#N@50	#N@100	#N@1	#N@10	#N@50	#N@100
BinGo	0	1	21	37	0	2	19	42
Genius	0	2	8	16	0	3	10	23
Gemini	1	4	36	75	1	7	35	73
IoTSeeker	1	6	39	83	1	7	41	82

D. Accuracy of Vulnerability Search

We answer RQ2 about how effective *IoTSeeker* is in identifying vulnerability functions in IoT devices when compared with other clone-based vulnerability search approaches. Referring to the vulnerability search experiments of other tools [7], [11], we use the same two vulnerabilities (CVE-2015-1791 and CVE-2014-3508) to perform comparative experiments. We use the fine-tuned models of *IoTSeeker* and *Gemini* in this experiment. By feeding a small amount of additional vulnerability function pairs, we can get fine-tuned models for a particular vulnerability search after a few rounds of fine-tunings. As described in Section III-A, we fine-tune the generic models of both *IoTSeeker* and *Gemini* in dataset I, and then search for vulnerabilities in dataset II consisting of 4,643 firmware images (such as IP cameras, routers and access points) of IoT devices. For each vulnerability, we carry out 5 epochs of fine-tuning.

For the search results of the *MIPS32* version vulnerability, we sort the functions in all firmware images in descending order of similarity scores. Table IV shows the effectiveness of vulnerability search on the top-K most similar results. In the comparison experiment, we take 4 different K values, namely 1,10,50 and 100. For each K value, we manually confirm the number of true positive instances in the top-K reported results and count them as ***#Num@K***.

From Table IV, we can see that the fine-tuned model of *IoTSeeker* has a great improvement on the search accuracy. For four different K values, *IoTSeeker* always identifies more real vulnerabilities than other tools. *BinGo* and *Genius* have average search accuracy lower than 45% and 25%, respectively. When the report results are given by these two tools, a large amount of manual effort has to be required to filter out false positives. Only *IoTSeeker* and *Gemini* find real vulnerabilities in top-1 reported results. But for the CVE-2015-1791 vulnerability, *IoTSeeker* finds 2, 3 and 8 more vulnerabilities than *Gemini* in the top-10, top-50 and top-100 results, which are 50%, 8.33% and 10.67% higher accuracy, respectively. For the CVE-2014-3508 vulnerability, *IoTSeeker* has 17.14% and 12.33% higher search accuracy than *Gemini* in the top-50 and top-100 results. Since our approach not only considers the control dependency but also constructs the transfer and use of the data, which captures more accurate and sufficient function semantic information. As a result, *IoTSeeker* achieve a higher accuracy than other three tools, and more vulnerabilities are detected. In terms of time cost, *BinGo*, *Genius*, *Gemini* and *IoTSeeker* need an average of 1.7s, 1.57s, 0.15s and 0.2s to calculate the similarity between a target function and a vulnerable function.

IV. CONCLUSION

In this paper, we present *IoTSeeker*, a cross-platform binary vulnerability search approach for IoT devices based on function semantic learning. With integrating both the data flow dependency and the control flow dependency of the binary function, we capture more function semantics. Experiments show that *IoTSeeker* achieves 88.49% AUC value, which improves 11.62%, 14.71% and 8.23% than that of *BinGo*, *Genius* and *Gemini*, respectively. For the case studies of two vulnerability searches, we discover at least 3 and at most 31 more vulnerability instances compared with any of three state-of-the-art tools in the top-50 candidates. *IoTSeeker* only needs about 0.2s to determine whether a function has a known vulnerability or not. The high predictive accuracy and low time cost indicate *IoTSeeker* is suitable for effectively identifying vulnerable functions from the embedded system software of IoT devices.

REFERENCES

[1] P. Middleton, P. Kjeldsen, and J. Tully, "Forecast: The internet of things, worldwide, 2013," 2013.
[2] S. Jeschke, C. Brecher, and H. Song, "Industrial internet of things: cyber manufacturing systems," in *Industrial internet of things: cyber manufacturing systems*.
[3] A. Humayed, J. Lin, F. Li, and B. Luo, "Cyber-physical systems security – a survey," *IEEE Internet of Things Journal*, vol. PP, no. 99, pp. 1–1, 2017.
[4] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 176–192, 2006.
[5] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire OS distributions," in *IEEE Symposium on Security and Privacy, SP 2012, San Francisco, California, USA*, 2012, pp. 48–62.
[6] J. Wurm, K. Hoang, O. Arias, A. Sadeghi, and Y. Jin, "Security analysis on consumer and industrial iot devices," in *21st Asia and South Pacific Design Automation Conference, ASP-DAC 2016, Macao, Macao, January 25-28, 2016*, 2016, pp. 519–524.
[7] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 480–491.
[8] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: cross-architecture cross-os binary search," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 678–689.
[9] P. Agrawal, A. Arasu, and R. Kaushik, "On indexing error-tolerant set containment," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, 2010, pp. 927–938.
[10] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.
[11] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 363–376.
[12] CVE, "Common vulnerabilities and exposures," http://cve.mitre.org/, accessed Apri 4, 2018.
[13] H. Flake, "Structural comparison of executable objects," in *Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop, DIMVA 2004, Dortmund, Germany, July 6.7, 2004, Proceedings*, 2004, pp. 161–173.
[14] 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.

[15] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2013, PPREW@POPL 2013, Rome, Italy*, 2013, pp. 5:1–5:6.

[16] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 709–724.

[17] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *SIGSOFT FSE*, 2014.

[18] Y. David and E. Yahav, "Tracelet-based code search in executables," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 349–360.

[19] S. Alrabaee, L. Wang, and M. Debbabi, "Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs)," *Digital Investigation*, vol. 18, pp. S11–S22, 2016.

[20] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a siamese time delay neural network," in *Advances in Neural Information Processing Systems 6, [7th NIPS Conference, Denver, Colorado, USA, 1993]*, 1993, pp. 737–744.

[21] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2016, pp. 2702–2711.

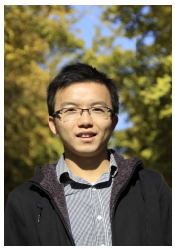
[22] T. Fawcett, "An introduction to roc analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2005.

Jian Gao received the BS degree in software engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2016. He is currently working toward the Ph.D. degree in software engineering at Tsinghua University, Beijing, China.

His research interests include binary vulnerability search, binary clone detection, machine learning in program analysis and their applications to industry.

Xin Yang received the BS degree in software engineering from Beijing Jiaotong University, Beijing, China, in 2016. She is currently working toward the M.S. degree in software engineering at Tsinghua University, Beijing, China.

Her research interests include binary vulnerability search, machine learning in program analysis and their applications to industry.



Yu Jiang received the BS degree in software engineering from Beijing University of Posts and Telecommunications in 2010, and the PhD degree in computer science from Tsinghua University in 2015. He worked as a Postdoc researcher in the department of computer science of University of Illinois at Urbana-Champaign, IL, USA, in 2016, and is now an assistant professor in Tsinghua University. His current research interests include domain specific modeling, formal computation model, formal verification and their applications in embedded systems.

Houbing Song received the M.S. degree in civil engineering from the University of Texas at ElPaso, USA, in 2006 and the Ph.D. degree in electrical engineering from the University of Virginia at Charlottesville, USA, in 2012. He joined the Department of Electrical and Computer Engineering, West Virginia University, USA, where he is an Assistant Professor. His research interests include cyber-physical systems, intelligent transportation systems, and wireless communications and networking.

Kim-Kwang Raymond Choo received the Ph.D. degree in information security from Queensland University of Technology, Australia, in 2006. He currently holds the Cloud Technology Endowed Professorship at The University of Texas at San Antonio. He is also an adjunct Associate Professor at the University of South Australia, and a Fellow of the Australian Computer Society. His research interests include cyber security and forensics.

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.