ANNs on Co-occurrence Matrices for Mobile Malware Detection

Xi Xiao[1], Zhenlong Wang[1], Qi Li[1], Qing Li[1*], Yong Jiang[1]
[1]Graduate School At Shenzhen, Tsinghua University
518055 Shenzhen, China
[e-mail: xiaox@sz.tsinghua.edu.cn, wangzl@sz.tsinghua.edu.cn, qi.li@sz.tsinghua.edu.cn, li.qing@sz.tsinghua.edu.cn, jiangy@sz.tsinghua.edu.cn]
*Corresponding author: Qing Li

Received February 4, 2015; revised May 21, 2015; accepted June 12, 2015; published July 31, 2015

Abstract

Android dominates the mobile operating system market, which stimulates the rapid spread of mobile malware. It is quite challenging to detect mobile malware. System call sequence analysis is widely used to identify malware. However, the malware detection accuracy of existing approaches is not satisfactory since they do not consider correlation of system calls in the sequence. In this paper, we propose a new scheme called Artificial Neural Networks (ANNs) on Co-occurrence Matrices Droid (ANNCMDroid), using co-occurrence matrices to mine correlation of system calls. Our key observation is that correlation of system calls is significantly different between malware and benign software, which can be accurately expressed by co-occurrence matrices, and ANNs can effectively identify anomaly in the co-occurrence matrices. Thus at first we calculate co-occurrence matrices from the system call sequences and then convert them into vectors. Finally, these vectors are fed into ANN to detect malware. We demonstrate the effectiveness of ANNCMDroid by real experiments. Experimental results show that only 4 applications among 594 evaluated benign applications are falsely detected as malware, and only 18 applications among 614 evaluated malicious applications are not detected. As a result, ANNCMDroid achieved an F-Score of 0.981878, which is much higher than other methods.

Keywords: Android, mobile malware, malware detection, system call sequence, Artificial Neural Network, co-occurrence matrix

This work is supported by the NSFC project (61202358,61402255),the National Basic Research Program of China (2012CB315803), the NationalHigh-tech R&D Program of China(2014ZX03002004) and the Shenzhen KeyLaboratory of Software Defined Networking.

http://dx.doi.org/10.3837/tiis.2015.07.023 **ISSN : 1976-7277**

1. Introduction

Along with the significant growth in the popularity of smartphones and the number of available mobile applications, mobile security is increasingly prominent [1, 2].Android operating system has become the most prevalent mobile system because of its openness and freeness. As the 64-Bit Android 5.0 OS published, Android becomes even more popular [3]. However, as a result of its openness, an adversary can easily create malicious applications, i.e., malware, and spread them [13]. According to the JNPR report [4], the Android malware (hereinafter "malware") has increased very quickly in the Android market. Thus, it's urgent for us to find an effective method for Android malware detection.

Different from traditional operating systems, Android has more diversified communication interfaces, e.g., sending messages and making phone calls. The structure of the application in Android is very different from that in traditional operating systems. It is more challenging to detect malware in Android. There are two mechanisms for detecting malware. One is static analysis which recognizes signatures of the malicious applications without running them [5-8]. In static analysis, various binary forensic techniques can be used, such as de-compilation, decryption, pattern matching and static system call analysis. These methods usually take less time but are subjected to hardly detecting unknown malware. To surmount this disadvantage, dynamic analysis is proposed [9-15]. Dynamic analysis is deployed to monitor the application's behavior, including memory modifications, network access, reading and writing files, making phone calls, sending messages and so on [14, 16].

Many dynamic analysis methods have been put forward to detect Android malware [9-14]. Among these methods, the most prevalent methods are to utilize system calls [9-13]. Some of them only employ the frequencies of system calls such as [9,10]. These methods do not take into account the relevant relationship between two system calls in the system call sequence. So they get low detection rate. There are also some works that use common subsequences of system call sequences [13, 17].They recognize malicious applications by exactly matching the common subsequences of the malware with the system call sequence of one software. Thus these methods cost too much time and easily cause overfitting.

The application's behaviors are achieved through the invocation of system calls. System calls of the application are arranged in chronological order, forming one system call sequence. System call sequences are a good simple discriminator for attacks [22]. A process consists of several threads. Due to the CPU time rotation mechanism, one thread often invokes system calls not in chronological order. Thus the system calls of the same thread are not always adjacent in the system call sequence of one process. In fact, one event is generally conducted by one thread. Therefore, there exists the relevant relationship, i.e., correlation, between two adjacent and non-adjacent system calls in the system call sequence of one application.

Based on the correlation of system calls, we put forward a new malware detection method, ANNs on Co-occurrence Matrices Droid (ANNCMDroid). The co-occurrence matrix models the sequence of system calls by associating one system call with another call within a certain distance. It can extract the relevant properties hidden in the sequence of system calls, record the characteristics of not only adjacent system calls but also non-adjacent ones and adapt to the complex dynamic characteristics of system calls evoked by one application. We observe that the relevant relationships of system calls are significantly different between the system call sequences of malicious applications and benign ones, and consequently the co-occurrence

matrices are prominently different between them. According to this observation, we use the statistics technique to calculate the co-occurrence matrices from the system call sequences and transform them into vectors. Thereafter, based on the vectors the classifier, Artificial Neural Network (ANN), is employed to detect Android malware. The experiments are conducted on the malware obtained from the repository [18] provided by Zhou et al. and the benign applications downloaded from Google. The results indicate that, the F-Score of our method is higher than [13] and [19].

The innovation of this work primarily lies in the following aspects. First of all, our scheme considers the relevant properties for the first time, and gets good detection results. Second, we discover that the relevant relationships of system calls are significantly different between the system call sequences of malware and benign software, and then use co-occurrence matrices to mine these relationships. Third, we use ANNs to classify co-occurrence matrices of malicious applications and benign ones in malware detection for the first time.

The outline of this paper is as follows: Section 2 briefly surveys the recent literature. A brief introduction to some background is stated in Section 3. Section 4 describes the details of the method proposed in this paper. The experiment environment and results are described in Section 5. Finally, Section 6 gives the conclusions and the further work.

2. Related Work

There are many features used for Android malware detection, such as permissions, API calls, sensitive data, network spatial features, code instructions, and system calls, etc. Among these features, permissions and API calls are relatively simple. Based on permissions, Enck et al. [5] proposed the Kirin security service for Android, which performs lightweight certification of applications to mitigate malware at install time. This method only detects specific malware. Meanwhile, Fuchs et al. [6] disassembled the APK source file and checked permissions and data flow of application components in source code. However, the method suffers from the drawback of Enck et al. Zheng et al. [20] also disassembled the source file and extracted API call sequences to generate signatures. Nevertheless this method can not detect malicious applications that have unknown malicious signatures. Peiravian and Zhu [19] combined permissions and API calls as features to do the detection. Although their method utilized machine learning algorithms, it can merely get a detection rate of 94.8%.

Besides these methods using permissions and API calls, there are many other approaches to recognize malicious applications. Enck et al. [11] proposed TaintDroid, an extension to Android, which tracks the flow of privacy sensitive data through thirty-party applications. This method could only identify the specific malware attempting to steal sensitive data. Isohara et al. [12] proposed a behavior analysis system which records all system calls of process management and file I/O operations and matches activities with signatures described by regular expressions. But, this system gets a low detection rate and a high false positive rate. Wei et al. [21] extracted network spatial features and used independent component analysis to identify malware. However, this method could only detect specific malicious applications which perform network activities. There are also some people who recognize malware through code instructions. Zhou et al. [7] proposed a method to detect the repackaged malware published in third-party marketplaces. It generates hash values from code instructions and calculates similarity scores between the original application and the repackaged malware. Nevertheless it can only detect repackaged malware, and it is difficult to get original official applications especially for those unpopular ones.

System calls are one of the most important features utilized in malware detection. Up to now, there are mainly two kinds of methods using system calls, one of which is to use system call frequencies. For example, Blasing et al. [9] presented AASandbox which performs both static and dynamic analysis on Android programs to detect suspicious applications. In the static part, the sandbox decompresses installation files and disassembles the corresponding executables. In the dynamic part, it records the system calls of a running application and calculates the frequencies of system calls. However, AASandbox can merely get a low detection rate. Burguera et al. [10] put forward CrowDroid, which also computes the frequencies of system calls, and suffers from the same drawback as AASandbox. The other kind of methods about system calls is to employ common system call sequences. For example, Rozenberg et al. [17] utilized the common system call subsequences to detect malware on the Windows platform. In the training phase, the method employs SPADE and Genetic Algorithm (GA) to extract the common subsequences that only exist in malware but not in benign software. In the testing phase, it checks whether there exists a match between a portion of the system call sequence invoked by the running software and one or more of the common subsequences. However, the time complexity of the algorithm to get common subsequences is too high, and the detection rate of this method is not higher than 90%. Lin et al. [13] proposed System Call Sequence Droid (SCSDroid), which adopts the thread-grained system call sequence activated by Android applications. Just like the method in [17], the common subsequences from the system call sequences of malware are extracted. However, this method suffers from two drawbacks: first, the time complexity to get common subsequences is too high; second, the sample set is too small, in which there are only 100 benign applications and 49 malicious ones.

In the above methods about system calls, these employing the frequencies do not consider the relevant relationship between two system calls in the system call sequence. So they get low detection rate. The other methods do the detection by exactly matching the common subsequence of the malware. These methods cost too much time and easily cause overfitting.

3. Background

3.1 Android Security Scheme

Different from the traditional operating system, Windows and Linux, Android is mainly used in the mobile terminals, which can deal with some other work, such as sending messages, making telephone calls, taking photos, etc.. Android's security mechanisms are different from the traditional operating systems too. Android is a "privilege separation" system. Before using limited system resources, such as network, telephone, SMS, Bluetooth, contacts and SD-card, an application must apply to the system in an XML file. Android adopts "sandbox" mechanism, which realizes the mutual isolation between different applications and processes. By default, the application has no access to system resources or other application resources. Each application runs in a separate Dalvik virtual machine, with separate address space and resources. The structure of the application in Android is very different from that in traditional operating systems. An Android application consists of four components, Activity, Service, Broadcast Receiver, and Content Provider. People almost hold mobile phones at every time and in every place. The information in the phone is closely related with personal life. Compared with traditional malware, Android malware on the phone can steal more personal privacy. It can cause a great loss, especially to the related people.

3.2 System Call

The main function of operating systems is to manage hardware resources and to provide a good environment for application developers. In order to achieve this goal, the kernel provides a series of scheduled kernel functions to developers by a set of interfaces, known as system calls. System calls transfer the application's request to the kernel, and call the corresponding kernel function to finish the required work, and then return the result to the application [22]. System calls embody user activity, incoming/outgoing traffic, files and memory access, energy consumption and etc.. Therefore, they are a good representative sample of the smartphone's behavior.

Android is a mobile operating system which runs on Linux kernel. In this work, Android 4.0.4 (Ice Cream Sandwich) version is used, which has 196 system calls as a part of its OS architecture [23]. There are many differences between the system call actions of Android and those of Linux. For example, there are no behaviors such as making calls, sending messages and using Bluetooth on traditional Linux. Here the **Table 1** shows several important system calls:

Table 1. Several important system calls

Name of system call	Function
Fork	Create a new process
Clone	Create a child process according to specific conditions
Getpid	Obtain the id of a process
Fcntl	File control
Read	Read file
Write	Write file
Ioctl	Master control function of I/O

3.3 Co-occurrence Matrix

The co-occurrence matrix is mainly used in the field of image recognition, such as texture recognition of wood surface, feature extraction of texture and defect detection of texture. Further more, the co-occurrence matrix has been used in the field of intrusion detection, such as [24, 25]. Yet to the best of our knowledge, it has not been used in Android malware detection before. The system call sequence looks chaotic, but there exists some relevant relationships between the system calls. The co-occurrence matrix can extract the relevant properties hidden in the sequence of system calls. It models the sequence of system calls by associating one system call with another call within a certain distance. The distance between the two related system calls and the frequency of this pair of system calls decide the correlation intensity among the system call sequences. In other words, the closer the distance between two system calls is or the more the frequency of this pair of system calls is, the greater intensity the system calls have. The element of the co-occurrence matrix represents the correlation intensity between two system calls within the distance. Thus the co-occurrence matrix records the characteristics of not only adjacent system calls but also non-adjacent calls. It can adapt to the complex dynamic characteristics of system calls evoked by one application.

3.4 Artificial Neural Network

A neural network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use. Artificial Neural Networks (ANNs) are computational models inspired by an animal's central nervous system. They resemble the brain in two respects [26]: firstly, knowledge is acquired by the network from its environment through a learning process; secondly, interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge. ANNs are generally presented as systems of interconnected "neurons" which can compute values from inputs [27]. Here is the model of a "neuron" labeled *k* in **Fig. 1**.

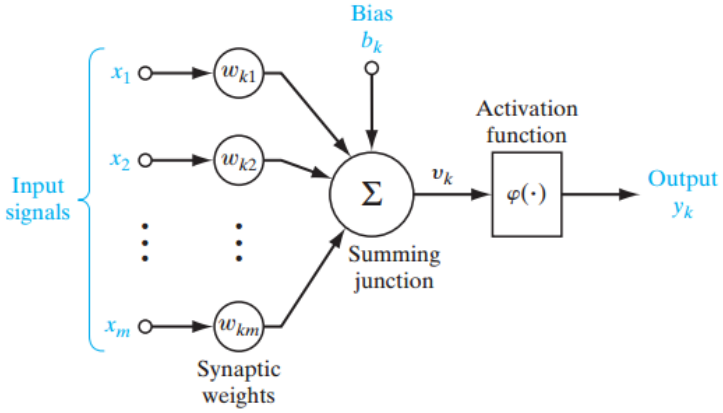


Fig. 1. The model of a "neuron" labeled *k*.

In mathematical terms, the neuron *k* depicted in the figure can be described by the following three equations:

$$u_k = \sum_{j=1}^{m} \omega_{kj} x_j \tag{1}$$

$$v_k = u_k + b_k \tag{2}$$

$$y_k = \varphi(v_k) \tag{3}$$

where $x_1, x_2, \ldots, x_m$ are the input signals; $\omega_{k1}, \omega_{k2}, \ldots, \omega_{km}$ are the respective synaptic weights of neuron *k*; u_k is the linear combiner output due to the input signals, b_k is the bias, y_k is the output signal of the neuron and $\varphi()$ is the activation function, such as

$$\varphi(v) = \frac{1}{1 + \exp(-av)} \tag{4}$$

where *a* is the slope parameter of the sigmoid function.

4. ANNCMDroid

The prior schemes that detect malware with system calls utilize either the frequencies of system calls or the common subsequences. The correlation properties between two system calls in the system call sequence of one application are ignored. Considering the relevant relationships, we propose ANNs on Co-occurrence Matrices Droid (ANNCMDroid). In our method, the correlation properties in system call sequences are extracted by co-occurrence matrices and ANNs are used to do the classification. As shown in **Fig. 2**, there are four steps in our work: (1) preprocessing, generate the system call sequences of applications; (2) calculating co-occurrence matrices, calculate co-occurrence matrices from the system call

sequences; (3) training, train the network with half of the samples; (4) testing, test the performance of the network with the other half of the samples.

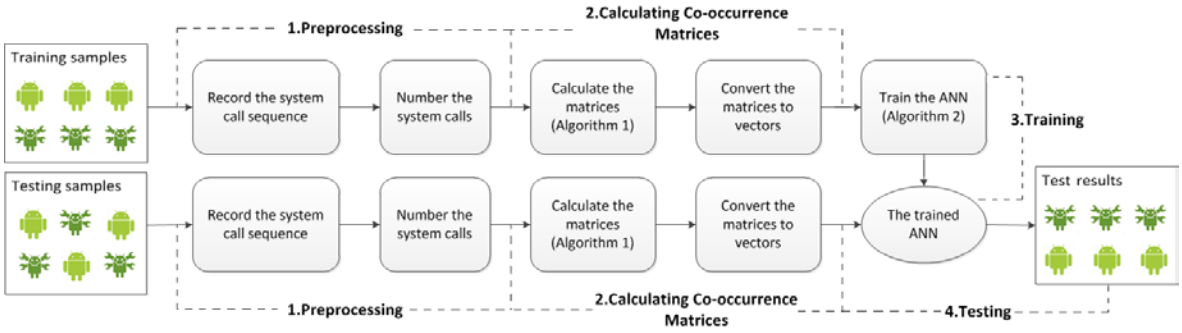


Fig. 2. The workflow diagram of our work.

4.1 Preprocessing

On the Android platform, the user process can't directly access hardware devices. When a user process needs to access hardware devices, such as reading disk files and receiving network data, it has to switch from the user mode to the kernel mode by system calls. During the preprocessing phase, system call sequences of applications are recorded by the *strace* command of Android. *strace* is a kind of tool software which can track and record the system call sequence invoked by a process, including the parameters, the returned value and the consumption of execution time. Besides *strace*, this work also needs *monkey*, another command-line tool of Android. *monkey* can send the stream of random and pseudo user events to applications. In this work, *monkey* is used to simulate the mobile phone user and trigger 1000 user events. After obtaining system calls, we convert the system call sequences from the character string sequence form to the number sequence form by numbering all the 196 system calls from S_0 to S_{195}.

4.2 Calculating Co-occurrence Matrices

The co-occurrence matrix is calculated by counting the times of transition from one system call to another one within the distance of *k*. The element of the co-occurrence matrix represents the correlation intensity between two system calls within the distance. Here is the pseudo-code of calculating the co-occurrence matrix:

Algorithm 1

Algorithm of Calculating the Co-occurrence Matrix

Input: the system call sequence $R = (r_1, r_2, \ldots, r_M)$ and the distance *k*

Output: the co-occurrence matrix $P = [p_{ij}]_{196\times196}$

1: $[c_{ij}]_{196\times196} = [0]_{196\times196}$ //initialize the matrix to zero.
2: **for** *i* = 1to *M*-*k* **do**
3: $x = r_i$;
4: **for** *j* =1 to *k* **do**
5: $y = r_{i+j}$;
6: $c_{xy} = c_{xy} + 1$; //count transition times from one system call to

7: **end for** //another system call within the distance of *k*.
8: **end for**
9: **for** i = 0 to 195 **do**
10: $c_i = c_{i\,0} + c_{i\,1} + c_{i\,2} + ... + c_{i\,195}$; //calculate transition times from one system call to
12: **end for** // all the other system calls within the distance of *k*.
13: **for** each p_{ij} **in** P
14: $p_{ij} = c_{ij} / c_i$; //normalize transition times.
15: **end for**

For example, suppose there are only five system calls, numbered as S_0,S_1,…,S_4, and the system call sequence is as flows:

S_1, S_0,S_3,S_0, S_2, S_4, S_3, S_2, S_1, S_4, S_0, S_3, S_4, S_2, S_3, S_1, S_0, S_3, S_1, S_4, S_2, S_3, S_4, S_2, S_1, S_3, S_0, S_2.

The times of transition from the system call S_i to the system call S_j within the distance of k is noted as c_{ij} and the times of transition from the system call S_i to any other system calls is noted as c_i. According to the above system call sequence, when k is 3, C_{00} =1, C_{01} =1, C_{02} = 4, C_{03} =4, C_{04} =3, C_0 =13, etc. By this means we can calculate all the c_{ij} and C_i. The results are as follows:

$$\left[c_{ij}\right]_{5\times5} = \begin{bmatrix} 1 & 1 & 4 & 4 & 3 \\ 5 & 1 & 2 & 5 & 2 \\ 3 & 2 & 2 & 4 & 3 \\ 3 & 4 & 6 & 2 & 5 \\ 1 & 3 & 3 & 5 & 2 \end{bmatrix} \qquad \left[c_i\right]_{1\times5} = \begin{bmatrix} 13 \\ 15 \\ 14 \\ 20 \\ 14 \end{bmatrix}$$

Then the transition probability from the system call S_i to the system call S_j within the distance of k is noted as p_{ij}, and it can be calculated in the following way:

$$p_{ij} = \frac{c_{ij}}{c_i} \tag{5}$$

According to the formula (5), we can get the co-occurrence matrix (where the distance k=3):

$$P = \left[p_{ij}\right]_{5\times5} = \begin{bmatrix} 1/13 & 1/13 & 4/13 & 4/13 & 3/13 \\ 5/15 & 1/15 & 2/15 & 5/15 & 2/15 \\ 3/14 & 2/14 & 2/14 & 4/14 & 3/14 \\ 3/20 & 4/20 & 6/20 & 2/20 & 5/20 \\ 1/14 & 3/14 & 3/14 & 5/14 & 2/14 \end{bmatrix}$$

After getting all the co-occurrence matrices of the applications in the training data, these matrices will be converted to vectors. Here the following case illustrates how to convert a matrix to a vector:

$$\begin{pmatrix} p_{1\,1} & \cdots & p_{1\,196} \\ \vdots & \ddots & \vdots \\ p_{196\,1} & \cdots & p_{196\,196} \end{pmatrix} \longrightarrow \left(p_{1\,1}, p_{1\,2}, \ldots, p_{1\,196}, p_{2\,1}, p_{2\,2}, \ldots, p_{2\,196}, \ldots, p_{196\,1}, p_{196\,2}, \ldots, p_{196\,196} \right)$$

$$= \left(x_1, x_2, x_3, \ldots, x_{38416} \right)$$

4.3 Training

During the training phase and the testing phase, we use Artificial Neural Networks as the classifier. ANNs are generally presented as systems of interconnected "neurons" which can compute values from inputs [28]. The input of the training phase is ($x(n)$,$d(n)$), where $x(n)$ is the converted vector and $d(n)$ is the label of the nth sample in the training data. $d(n)$ is defined as follows: when the application is malicious, $d(n) = 1$, otherwise, $d(n) = 0$. There are 196×196 input-layer nodes and only one output-layer node. The primary work of training is to train the networks so that the networks have the ability to recognize malicious and benign applications. Let N be the size of training data and L be the depth of the network. Here the Back-Propagation algorithm is adopted to train the networks [26, 29]:

Algorithm 2

Algorithm of Back-Propagation

Input: the training data
Output: the weights and thresholds of the network

1: **for** each weight // initialize every weight.
2: $\omega_{ij}^{(l)}(n) = r$; //r is a random number chosen from a uniform
3: **end for** //distribution whose means is 1.
4: *error* = 1;
5: **while** *error* >= 0.1
6: **for** n=1 to N **do** //N is the size of training data.
7: Forward Computation (n); //**Remark.1**
8: Backward Computation (n); //**Remark.2**
9: **end for**
10: $error = \sum_{n=1}^{N} \left(d(n) - y_1^{(L)}(n) \right)^2 / 2N$
11: **end while**

Remark.1. Forward Computation (n). The nth sample in the training data is denoted as ($x(n)$,$d(n)$), where $x(n)$ denotes the input vector acquired from the co-occurrence matrix and $d(n)$ denotes the label of nth sample. The induced local field $v_j^{(l)}(n)$ for neuron j in layer l is

$$v_j^{(l)}(n) = \sum_i \omega_{ji}^{(l)}(n) y_i^{(l-1)}(n) \tag{6}$$

where $y_i^{(l-1)}(n)$ is the output signal of neuron i in the previous layer l-1 at iteration n, and $\omega_{ji}^{(l)}(n)$ is the synaptic weight of neuron j in layer l that is fed from neuron i in layer l-1.

$\omega_{j0}^{(l)}(n)$ is the threshold of the network. Assuming the use of a sigmoid function, the output signal of neuron *j* in layer *l* is $y_j^{(l)}(n) = \varphi_j(v_j^{(l)}(n))$. If neuron j is in the first hidden-layer (i.e., *l*=1), set $y_j^{(0)}(n) = x_j(n)$, where $x_j(n)$ is the jth element of the input vector $\boldsymbol{x}(n)$. If neuron j is in the output-layer (i.e., *l*=*L*), then compute the error signal

$$e(n) = d(n) - y_1^{(L)}(n) \tag{7}$$

Note that there is only one node in the output-layer in our network.

Remark.2. Backward Computation (*n*). Local gradients of the neural in output-layer *L* is computed by

$$\delta^{(L)}(n) = e(n)\varphi'(v_1^{(L)}(n)) \tag{8}$$

where the prime in $\varphi'_j()$ denotes differentiation with respect to the argument. And local gradients of neural *j* in hidden-layer *l* are computed by

$$\delta_j^{(l)}(n) = \varphi_j'\left(v_j^{(l)}(n)\right)\sum_s \delta_s^{(l+1)}(n)\omega_{sj}^{(l+1)}(n) \tag{9}$$

where *s* refers to the neuron in layer *l*+1 that is fed from neuron *j* in layer *l*. Then the synaptic weights of the network in layer *l* are adjusted according to the generalized delta rule

$$\omega_{ji}^{(l)}(n+1) = \omega_{ji}^{(l)}(n) + \alpha\left[\Delta\omega_{ji}^{(l)}(n-1)\right] + \eta\delta_j^{(l)}(n)\, y_i^{(l-1)}(n) \tag{10}$$

where η is the learning rate and α is the momentum constant.

Note that in our algorithm, the sigmoid functions, $\varphi()$ and $\varphi_j()$, are all defined by the formula (4).

4.4 Testing

After the training phase, the neural network has already acquired the ability to recognize benign and malicious applications. During the testing phase, it only needs forward computations. Using the methods in Section 4.1 and 4.2, the co-occurrence matrices of the applications in the testing data are calculated and transformed into vectors. During the testing phase, the vectors are input to the trained neural network. Thereafter according to the formulas (1),(2),(3),(4), the network obtains the final results by computing the output of each node in the network layer by layer. The value of the output-layer node is between 0 and 1. When the value is greater than 0.5, the application is recognized as malicious one, otherwise is recognized as benign one.

5. Experiments and Results

5.1 Methodology

5.1.1 Datasets

The experimental datasets are composed of 1227 malicious applications and 1189 benign ones. The malicious applications, including 49 malware families, are offered by Zhou et al. [18]. By modifying the Chrome APK-downloader plugin, 1189 applications are downloaded from the Google Play as the benign applications of this work. Half of the malicious application dataset and half of the benign application dataset are used for training and the rest of datasets for testing. The programs of Artificial Neural Networks were written in C++ and executed on Ubuntu. In our experiments, the momentum constant of networks, α, is set to zero. We choose the activation function as the formula (4), where a=1. The learning rate, η, is set to 0.9 in all our experiments except those in Section 5.2.3. In order to speed up the execution, the programs utilized OpenMp, which is an API that supports multiprocessing programming.

5.1.2 Metric

There are some criteria to evaluate the experiment results, such as True Positive Rate (TPR), False Positive Rate (FPR), precision, and F-Score. Let TP denote the number of malicious applications that are correctly detected, and FP denote the number of benign applications that are falsely detected as malware. On the contrary, TN denotes the number of benign applications that are correctly detected, and FN denotes the number of malicious applications that are falsely detected as benign ones. Then True Positive Rate, i.e., detection rate, is defined as

$$TPR = \frac{TP}{TP + FN} \tag{11}$$

False Positive Rate is defined as

$$FPR = \frac{FP}{FP + TN} \tag{12}$$

Precision is defined as

$$precision = \frac{TP}{TP + FP} \tag{13}$$

F-Score [30] is the harmonic mean of the precision and the TPR, defined as

$$F - Score = \frac{2 \times Precision \times TPR}{Precision + TPR} \tag{14}$$

5.1.3 Comparison with Other Schemes

When the distance, k, of the co-occurrence matrices is 2 and the hidden-layer node number is 2, ANNCMDroid achieves the TPR of 0.970684, the FPR of 0.00673401 and thus the F-Score of 0.981878. In order to evaluate the experiment result, we compare ANNCMDroid with Peiravian [19] and SCSDroid [13] in the F-Score, the TPR and the FPR. Peiravian [19] combined permissions and API calls as features and utilized machine learning algorithms to detect Android malware. SCSDroid [13] adopted the thread-grained system call sequence and used the common subsequences to identify Android malware. The results are shown in **Table 2**. Peiravian did not calculate the F-Score, we derive it from the TPR and precision of

Peiravian. SCSDroid did not give the F-Score too, we obtain it from the TP, FP, TN and FN of SCSDroid.

Table 2. Comparison with Other Methods

	F-Score	TPR	FPR
Peiravian	0.9525	0.948	0.0312
SCSDroid	0.969697	0.979592	0.02
ANNCMDroid	0.981878	0.970684	0.00673401

Compared with Peiravian, the F-Score and the TPR of ANNCMDroid are much higher and the FPR is much lower. Compared with SCSDroid, the TPR of ANNCMDroid is a little lower, but the FPR is much lower and thus the F-Score is much higher. In general, the result of ANNCMDroid is better than those of both Peiravian and SCSDroid.

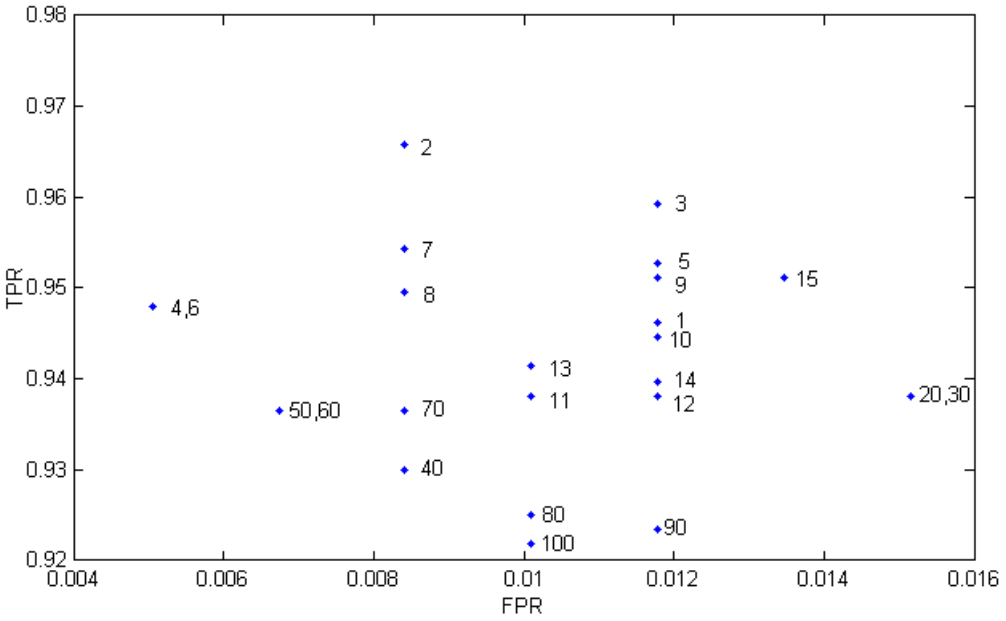
5.2 Experimental Results

The experiment results depend on the distance of co-occurrence matrices and the structure of networks. The structure of the networks involves two aspects, the layer number and the node number in every layer. In this work, we use Android 4.0.4 (Ice Cream Sandwich), which has 196 system calls as a part of its OS architecture [23]. There are 196×196 nodes in the input layer (the first layer). We adopt the three-layer networks with only 1 node in the output layer (the third layer) to do the classification. The layer number is 3 and the node number in the first layer and the third layer is 196×196 and 1 respectively in our networks. So the hidden-layer node number, i.e., the node number in the second layer, decides the structure of the networks.

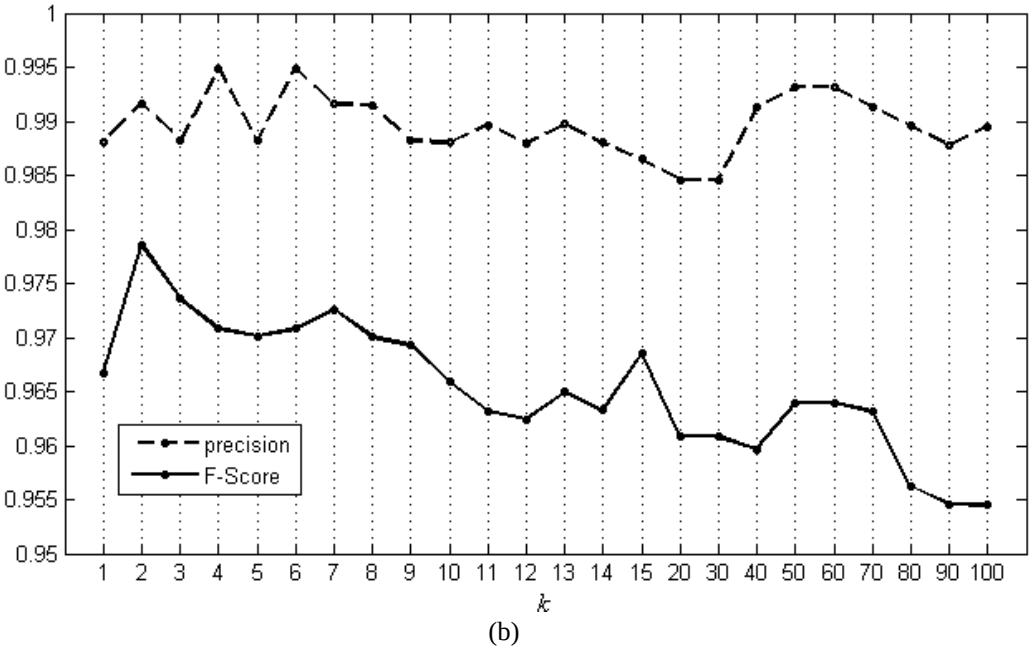
5.2.1 Effect of the Distance

In order to investigate the effect of the distance, k, of the co-occurrence matrices, we first let the hidden-layer node number, H, fixed, and then adjust the value of k. **Fig. 3(a)** illustrates the TPR and the FPR of the networks, of which the hidden-layer node number is 10 and k ranges from 1 to 100. The number on the right side of one point is the distance, k, corresponding to this point. In our experiment, the FP fluctuates within a small range, 3-9, and the value of FP+TN is a constant number of 594. So according to the formula (12), the variation of the FPR only depends on the fluctuation of the FP. Because the FP could only be 7 different values. The FPR could only achieve 7 different values. Thus there are many points have the same FPR. As shown in **Fig. 3(a)**, these points are on the same vertical line.

Fig. 3(b) shows the F-Score and the precision of the networks with different k. As descripted in **Fig. 3(b)**, when k is 2, the network gets the highest F-Score of 0.978548. Meanwhile, the FPR is 0.00841751 and the TPR is 0.965798, i.e., only 5 applications among 594 evaluated benign applications are falsely detected as malware, and only 21 applications among 614 evaluated malicious applications are not detected. It can be seen that the network gets a better performance when k is between 2 and 7.



(a)



(b)

Fig. 3. The experiment results of the networks where H is 10 and k ranges from 1 to 100. The result on TPR and FPR is labeled by (a). The result on F-Score and precision is labeled by (b).

Fig. 4 illustrates the F-Score and the precision of the networks, of which the hidden-layer node number is 28 and k ranges from 1 to 100. As descripted in **Fig. 4**, when k is 2, the network gets the highest F-Score of 0.979253. At this time, the FPR is 0.0016835 and the TPR is 0.960912, i.e., only 1 applications among 594 evaluated benign applications are falsely detected as malware, and 590 applications among 614 evaluated malicious applications are

correctly detected. It is obvious that the network achieves a better performance when *k* is between 2 and 7.

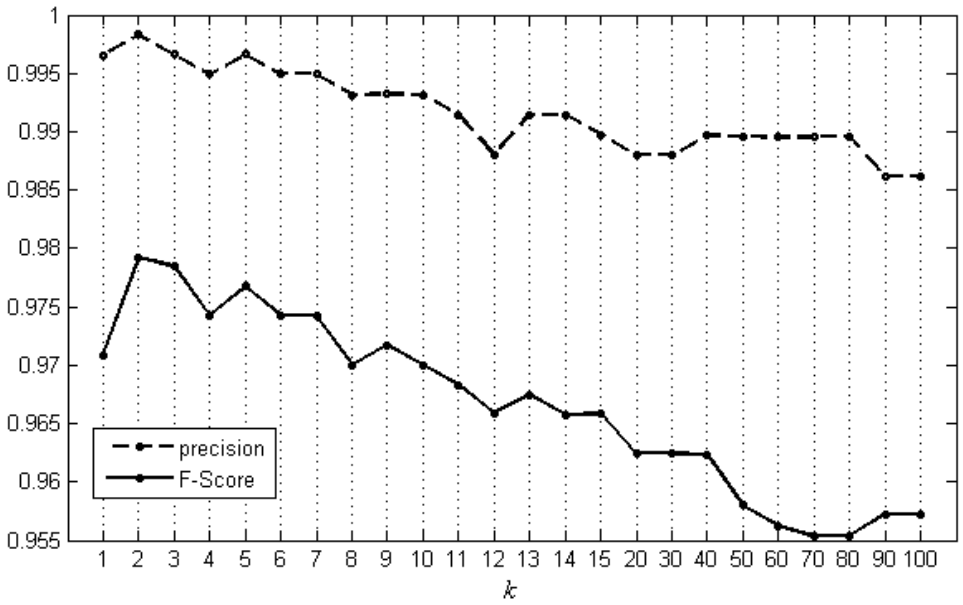


Fig. 4. The F-Score and precision of the networks where *H* is 28 and *k* ranges from 1 to 100.

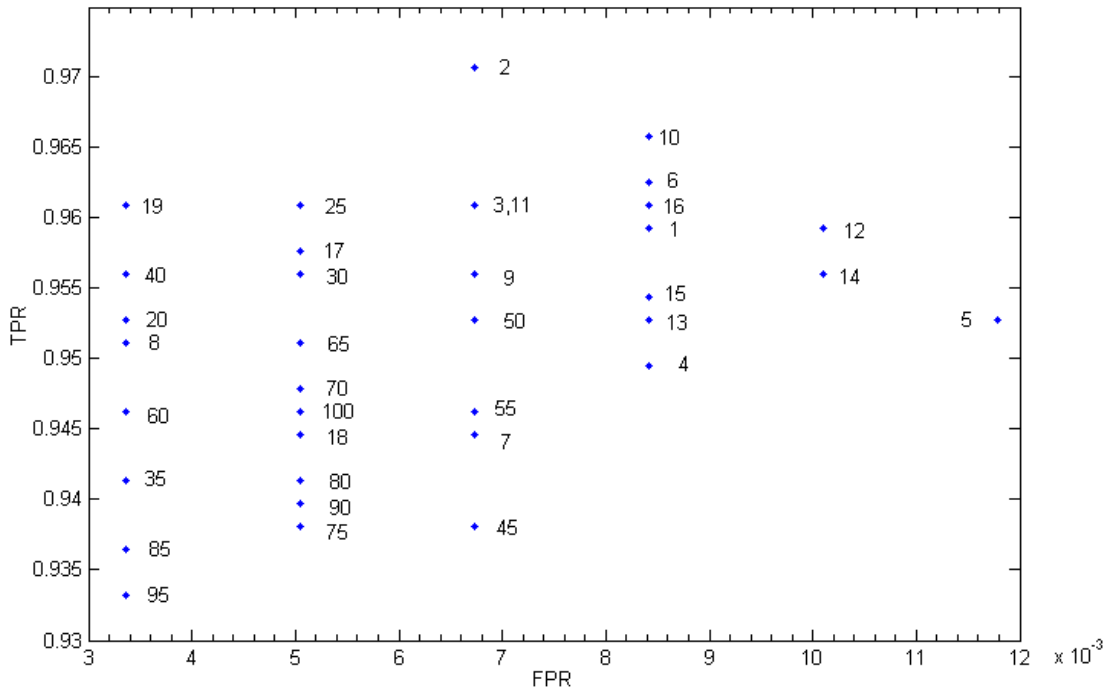
From **Fig. 3** and **Fig. 4**, we observe that F-score shows a decreasing trend with increase in *k* when *k* is more than 2. The reason is that the farther the distance between two system calls is, the weaker the intensity of the two system calls is, which means that there will be a weak relationship betweeen the two system calls. Moreover, if *k* is set to 1, F-Score is lower as well since there will be not enough association information in the co-occurrence matrices. Therefore, when *k* is set to 2, the network gets the highest F-Score.

A Markov chain is a random process, which possess a property that the probability distribution of the next state depends only on the current state and not on the sequence of events that preceded it. In our special case, when the distance *k*=1, only the relationship between the adjacent system calls is considered. The system call sequence can be regarded as a Markov chain. Because the Markov chain takes some correlation in the system call sequence into account, the case of *k*=1 can yield a good result (the F-Score above 0.965). It merely extracts the relationship between the adjacent system calls, neglecting the correlation between the non-adjacent system calls. However, when *k*=2, all the correlation between the adjacent and non-adjacent system calls is taken into consideration. The association information is more than that in the case of *k*=1. Thus the performance is better than that of *k*=1.

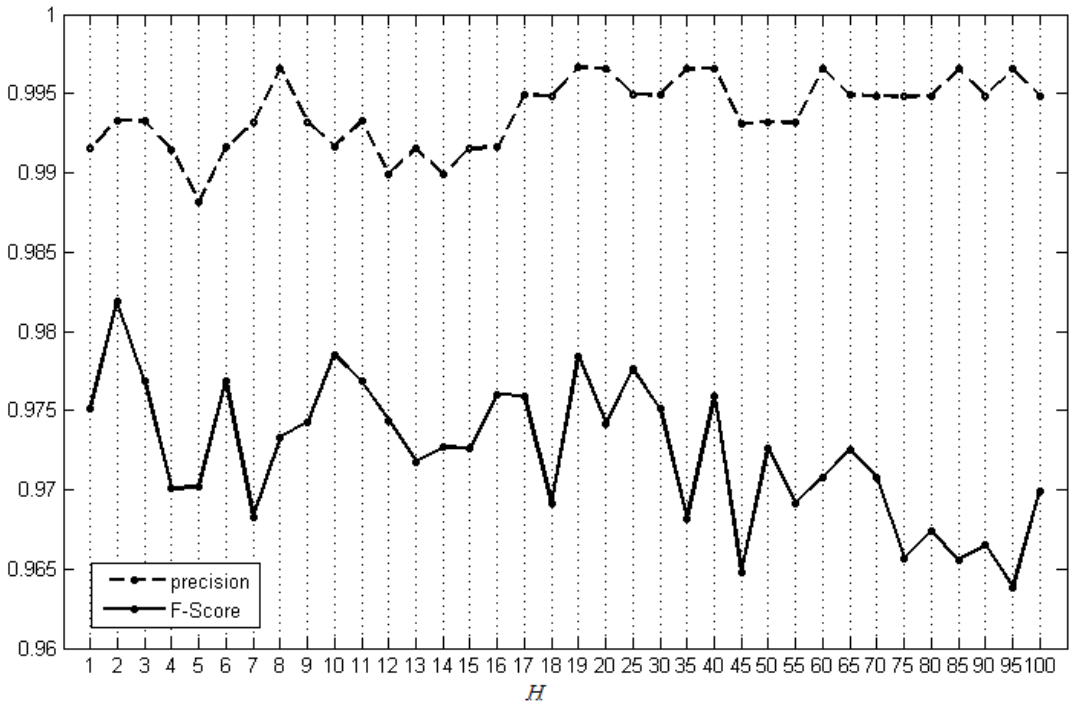
5.2.2 Effect of the Hidden-layer Node Number

In order to investigate the effect of the hidden-layer node number, *H*, of the neural networks, we first let the distance, *k*, to be a fix value, and then adjust the value of *H*. **Fig. 5(a)** illustrates the TPR and the FPR of the networks, of which *k* is 2 and the hidden-layer node number ranges from 1 to 100. And **Fig. 5(b)** shows the F-Score and the precision of the networks with different *H*. As descripted in **Fig. 5(b)**, when *H* is 2, the network gets the highest F-Score of 0.981878. Meanwhile, the FPR is 0.00673401 and TPR is 0.970684. In fact, in this case only 4

applications among 594 evaluated benign applications are falsely detected as malware, and only 18 applications among 614 evaluated malicious applications are not detected.



(a)



(b)

Fig. 5. The experiment results of the networks where *k* is 2 and *H* ranges from 1 to 100. The result on TPR and FPR is labeled by (a). The result on F-Score and precision is labeled by (b).

Fig. 6 shows the F-Score and the precision of the networks, of which the value of *k* is 5 and the hidden-layer node number ranges from 1 to 100. From **Fig. 6**, we can see when *H* is 5, the network gets the highest F-Score of 0.977667. In this case, the FPR is 0.00673401 and the TPR is 0.962541, i.e., only 4 applications among 594 evaluated benign applications are falsely detected as malware, and 591 applications among 614 evaluated malicious applications are correctly detected.

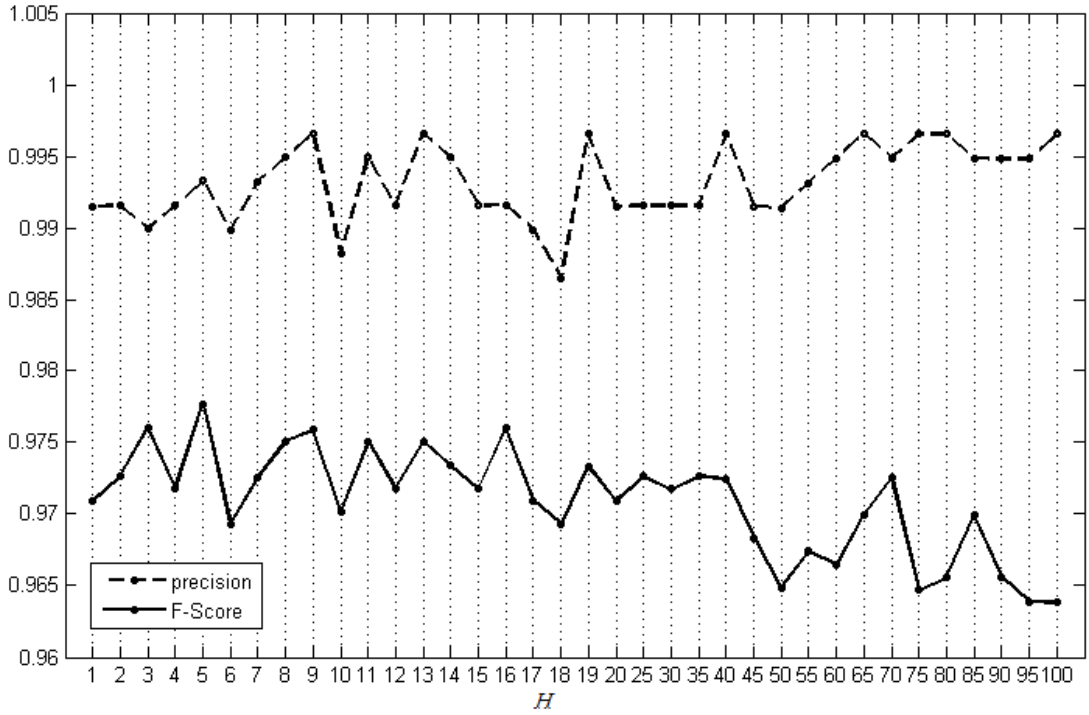


Fig. 6. The F-Score and precision of the networks where *k* is 5 and *H* ranges from 1 to 100.

5.2.3 Effect of Learning Rates

The learning rate, η decides the learning speed of neural networks [26]. **Table 3** shows the variation of the training times and the F-Score with the different learning rate from 0.1 to 0.9. In this table, we utilizes the three-layer network, of which the number of hidden-layer nodes is 28 and the distance, *k*, is 2. When the learning rate becomes smaller, it needs more times to train the network to convergence. And when the learning rate is 0.1, it even needs 209 times for the network to convergence. It can be seen from the table that the changes of the F-Score are relatively slight with the variation of learning rates.

Table 3. Effect of Learning Rates

Learning Rate	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Training Times	209	124	74	66	61	57	53	50	48
F-Score	0.98103	0.98100	0.98100	0.98016	0.98097	0.98013	0.98013	0.97925	0.97925

6 Conclusions and Further Work

Taking the relevant relationship between two system calls in the system call sequence of one application into consideration, we present a new Android malware detection method, ANNs on Co-occurrence Matrices Droid (ANNCMDroid), which abstracts the relevant properties in system call sequences by co-occurrence matrices and use ANNs to classify these matrices.

Based on the fact that the co-occurrence matrices are significantly different between malicious applications and benign ones, our method gets a high F-score and outperforms the other methods.

There are some directions for further study based on system call sequences to detect mobile malware. First, besides ANNs there are many other classifiers can be used in this work, such as SVM, K-NN and so on. It is meaningful for us to find which classifier is the most suitable. Second, the function of every system call is very important, but it is ignored in our method at present. In the future, we will incorporate the function to improve the detection.

Acknowledgments

This work is supported by the NSFC project (61202358,61402255), the National Basic Research Program of China (2012CB315803), the National High-tech R&D Program of China(2014ZX03002004) and the Shenzhen Key Laboratory of Software Defined Networking.

We would like to thank the anonymous reviewers for their helpful suggestions. We also would like to thank the authors in [18] to provide the malware dataset for us and thank XiangMing Li, XianniXiao, Yi He, and Peng Fu for the helpful discussion.

References

[1] H.T. Phuc, P. Chang-Woo, L. Minsik, C. Sang-Il, J. Sang-Hoon, J. Gu-Min, "Rapid Implementation of 3D Facial Reconstruction from a Single Image on an Android Mobile device," *KSII Transactions on Internet and Information Systems*, vol.8, no.5, pp.1690-1710, 2014. Article (CrossRef Link).

[2] X. Xiao, X. Xiao, Y. Jiang,Q. Li, "Detecting Mobile Malware with TMSVM," in *Proc. of 10th International Conference on Security and Privacy in Communication Networks*, 2014. Article (CrossRef Link).

[3] W. Jackson, "The Future of Android: The 64-Bit Android 5.0 OS," *Android Apps for Absolute Beginners*, pp 591-650, 2014. Article (CrossRef Link).

[4] "Third Annual Mobile Threats Report: March 2012 through March 2013," *Juniper Networks Mobile Threat Center*. Article (CrossRef Link).

[5] W. Enck, M. Ongtang, P. McDaniel, "On lightweight mobile phone application certification," in *Proc. of the 16th ACM Conference on Computer and Communications Security*, ACM, 2009. Article (CrossRef Link).

[6] A.P. Fuchs, A. Chaudhuri, J.S. Foster, "SCanDroid: Automated security certification of Android applications," *Technical Report, University of Maryland, College Park*, 2009. Article (CrossRef Link).

[7] W. Zhou, Y. Zhou, X. Jiang, P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. of the 2nd ACM conference on data and application security and privacy*, pp. 317-326, 2012. Article (CrossRef Link).

[8] A.D. Schmidt, R. Bye, H.G. Schmidt, J. Clausen, O. Kiraz, K.A. Yuksel, S.A. Camtepe, S. Albayrak, "Static Analysis of Executables for CollaborativeMalware Detection on Android," in *Proc. of 2009 IEEE International Conference on Communications*, pp. 1-5, IEEE, 2009. Article (CrossRef Link).

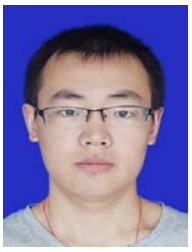
[9] T. Blasing, L. Batyuk, A.D. Schmidt, S.A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference*, pp. 55–62, IEEE, 2010. Article (CrossRef Link).

[10] I. Burguera, U. Zurutuza, S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proc. of the 1st ACM Workshop on Security and Privacy in Smartphones*

and Mobile Devices, pp. 15–26, ACM, 2011. Article (CrossRef Link).
[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel,and A. N. Sheth, "TaintDroid: An information-flow tracking system for real-time privacy monitoring on smartphones," in *Proc. of the 9th USENIX conference on Operating systems design and implementation*, pp. 1–6, 2010. Article (CrossRef Link).
[12] T. Isohara, K. Takemori, A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Proc. of the 7th international conference on computational intelligence and security*, pp. 1011-1015, 2011. Article (CrossRef Link).
[13] Y.D. Lin, Y.C. Lai, C.H. Chen, H.C. Tsai, "Identifying android malicious repackaged applications by thread-grained system call sequences," *Computers & Security*, vol. 39, pp. 340-350, 2013. Article (CrossRef Link).
[14] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. Weiss, " "Andromaly": a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38**,** no. 1, pp. 161-190, 2012. Article (CrossRef Link).
[15] A. Shabtai, L. Tenenboim-Chekina, D. Mimran, L. Rokach, B. Shapira, Y. Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Computers & Securit*, vol. 43, pp. 1-18, 2014. Article (CrossRef Link).
[16] A. Moser, C. Kruegel, E. Kirda, "Limits of static analysis for malware detection." in *Proc. of the 23th Annual Computer Security Applications Conference*, pp. 421-430, 2007. Article (CrossRef Link).
[17] B. Rozenberg, E. Gudes, Y. Elovici, Y. Fledel, "A method for detecting unknown malicious executables," in *Proc. of the 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 190-196, IEEE, 2011. Article (CrossRef Link).
[18] Y. Zhou, X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. of the 33rd IEEE symposium on security and privacy*, pp. 95-109, 2012. Article (CrossRef Link).
[19] N. Peiravian, X. Zhu, " Machine Learning for Android Malware Detection Using Permission and API Calls," in *Proc. of the 2013 IEEE 25th International ConferenceTools with Artificial Intelligence*, pp. 300-309, IEEE, 2013. Article (CrossRef Link).
[20] M. Zheng, M. Sun, J. Lui, "Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware," in *Proc. of the 2013 12th IEEE International Conference Trust, Security and Privacy in Computing and Communications*, pp. 163-171, 2013. Article (CrossRef Link).
[21] T.E. Wei, C.H. Mao, A.B. Jeng, H.M. Lee, H.T. Wang, D.J. Wu, "Android malware detection via a latent network behavior analysis," in *Proc. of the2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, pp. 1251-1258, 2012. Article (CrossRef Link).
[22] A.H. Steven, F. Stephanie and S. Anil, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151-180, 1998. Article (CrossRef Link).
[23] http://developer.android.com/about/versions/android-4.0-highlights.html#DeveloperApis
[24] O. Mizuki, O. Yoshihiro, A. Hirotake, K. Kazuhiko, "Anomaly Detection Using Layered Networks Based on Eigen Co-occurrence Matrix," in *Proc. Of Recent Advanced in Intrusion Detection*, pp. 223–237, 2004. Article (CrossRef Link).
[25] C. Liangwen, A. Masayoshi, "An SVM-Based Masquerade Detection Methodwith Online Update Using Co-occurrence Matrix," in *Proc. Of Detection of Intrusions and Malware & Vulnerability Assessment*, pp. 37-53, 2006. Article (CrossRef Link).
[26] S.S. Haykin, *Neural networks and learning machines*, 3rd Edition, Pearson Education, 2009. Article (CrossRef Link).
[27] M.T. Hagan, H.B. Demuth, M.H. Beale, *Neural network design*, 2nd Edition, 1996. Article (CrossRef Link).
[28] K. Anders, V. Jesper, " Neural network ensembles, cross validation, and active learning," in *Advances in neural information processing systems*, pp. 231-238, 1994. Article (CrossRef Link).
[29] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Proc. of 1989 International Joint Conference on Neural Networks*, pp. 593-605, IEEE,1989.

Article (CrossRef Link).
[30] Walid Magdy and Gareth J.F. Jones, "PRES: A Score Metric for Evaluating Recall-OrientedInformation Retrieval Applications," in *Proc. of Research and Development in Information Retrieval - SIGIR'10*, pp. 611-618, 2010. Article (CrossRef Link).

Xi Xiao is a lecturer in Graduate School At Shenzhen,Tsinghua University. He got his Ph.D. degree in 2011 in State Key Laboratory of Information Security, Graduate University of Chinese Academy of Sciences. His research interests focus on information security and the computer network.



Zhenlong Wang is a graduate student in Graduate School At Shenzhen, Tsinghua University. He got his B.S. degree from Jilin Universityin 2013. His research interests focus on information security.

Qi Li is an associate researcher in Graduate School At Shenzhen,Tsinghua University. He received the B.Sc. degree and the Ph.D. degree from Tsinghua University in 2003 and 2012. His research interest includes network architecture and protocol design, system and network security.

Qing Li is currently an assistant researcher of Graduate School at Shenzhen, Tsinghua University. He received his B.S. degree from Dalian University of Technology, Dalian, China, in 2008, the Ph.D. degree from Tsinghua University, Beijing, China, in 2013; all in computer science and technology. His research interests include reliable and scalablerouting of the Internet, software defined networking and information centric network.

Yong Jiang is a professor and a Ph.D. supervisor in Graduate School at Shenzhen, Tsinghua University. He got his Ph.D. degree in Tsinghua University 2002. He mainly engaged in computer network architecture, Internet applications, and Information security. He hosted and participated in a lot of national projects such as 863 project, 973 project and et al. He has published more than 50 papers in top international journals and international conferences.