Received December 30, 2018, accepted January 17, 2019, date of publication January 29, 2019, date of current version February 27, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2896003

A Combination Method for Android Malware Detection Based on Control Flow Graphs and Machine Learning Algorithms

ZHUO MA[1,2], HAORAN GE[1], YANG LIU[1], MENG ZHAO[1], AND JIANFENG MA[1,2]
[1]School of Cyber Engineering, Xidian University, Xi'an 710071, China
[2]Shaanxi Key Laboratory of Network and System Security, Xi'an 710071, China

Corresponding author: Zhuo Ma (mazhuo@mail.xidian.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant U1764263 and Grant 61872283, in part by the Natural Science Basic Research Plan in the Shaanxi Province of China under Grant 2016JM6074, and in part by the China 111 Project under Grant B16037.

ABSTRACT Android malware severely threaten system and user security in terms of privilege escalation, remote control, tariff theft, and privacy leakage. Therefore, it is of great importance and necessity to detect Android malware. In this paper, we present a combination method for Android malware detection based on the machine learning algorithm. First, we construct the control flow graph of the application to obtain API information. Based on the API information, we innovatively construct Boolean, frequency, and time-series data sets. Based on these three data sets, three detection models for Android malware detection regarding API calls, API frequency, and API sequence aspects are constructed. Ultimately, an ensemble model is constructed for conformity. We tested and compared the accuracy and stability of our detection models through a large number of experiments. The experiments were conducted on 10010 benign applications and 10683 malicious applications. The results show that our detection model achieves 98.98% detection precision and has high accuracy and stability. All of the results are consistent with the theoretical analysis in this paper.

INDEX TERMS Control flow graph, application programming interface, machine learning, malware detection.

I. INTRODUCTION

Android has become the first choice for many handset makers and other intelligent devices. According to Gartner [1], the global sales of Android-powered smart phones ranked first in 2017, accounting for 84.1% of all smart phone sales. However, the Android system is often attacked by malicious software. In 2017, Symantec [2] intercepted an average of 24,000 mobile phone malwares per day. In contrast to other platforms, Android allows for installing applications from unverified sources, such as third-party markets, which makes it easier for a malicious code developer to attack it. According to statistics, 17% of all Android applications are malware [3]. It is essential and of great importance to stop the proliferation of malwares on Android markets and smart phones.

The Android platform provides several security measures, most notably the Android permission system to prevent malicious software. To perform a certain task that needs Android permission, such as the location permission, each app has to explicitly request permission from the user during installation time or running time. However, without better understanding of Android permission, the users usually grant permission to unknown applications. As a consequence, the permission system can hardly guarantee realistic security.

Nowadays, a large number of researchers and companies began research on android malware. In industry, several companies have developed mobile anti-virus software and detection machines for Android malware detection. Among which Google introduced the latest machine learning modules and technologies which have significantly improved Google Play's security detection capabilities. While in academia, there are many articles on Android malware detection at conferences, in journals and on the Internet. Many are about an overview of Android security. For example, Enck *et al.* [4] elaborated on the Android system security model and the development of secure applications. Others are about Android malware. For example, Zhou and Jiang [5] systematically researched more than 1200 malicious software discovered

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

from August 2010 to October 2011 and established data sets about malware installation methods, malicious behavior activation method and attack payloads; DroidRanger [6] extracted malicious features through a manual analysis of known malwares and used heuristic rules to detect unknown malware. Many articles are about analysis and discussion of specific issues. For example, in order to solve the privacy leakage, McGill University designed a real-time monitoring system AppAudit [7] to monitor Android applications and detect whether there is a privacy leak in the application. As for vulnerabilities existing in Android systems, Felt *et al.* [8] analyzed the Android malware and found that there were loopholes in the Android platform, and conducted an in-depth study of the vulnerability; ComDroid [9] researched Android communication vulnerabilities, and initially solved the vulnerability of intentional information theft and forgery. Machine learning algorithms have been proved to be reliable and accurate in many problems [10], [11], and have also been used to detect Android malware. For example, McLaughlin *et al.* [12] constructed a malware detection system by using a convolutional neural network algorithm to analyze the application's byte code sequence.

In this paper, we present a machine learning method for Android malware detection which can automatically detect known and unknown types of malware if it belongs to the malware types that we have analyzed. In our methods, we first de-compile the Android application and construct the control flow graph (CFG) from the source code. Then we extract Application Program Interface (API) calls from the CFG and build three different types of API data sets:Boolean data sets, frequency data sets and chronological data sets. We build 3 detection models: API Usage Detection Model, API Frequency Detection Model and API Sequence Detection Model based on the data sets using machine learning methods.

The experiment with 10010 benign applications from AndroZoo [13] and 10683 malwares from Android Malware Dataset (AMD) [14], [15] shows that our methods are effective: the API Usage Detection Model detects 95% of the malware samples with a false-positive rate of 6.2%. The API Frequency Detection Model detects 97% of the malware samples with a false-positive rate of 9.1%. The API Sequence Detection Model detects 99% of the malware samples with a false-positive rate of 2.9%. So far, our methods are the first to construct chronological datasets of the Android application using static detection methods and the Long Short-Term Memory (LSTM) algorithm to build the detection model.

In summary, the contributions of this paper are as follows:

1) We extracted system APIs from the application's control flow graph and built 3 different data sets. Compared to extracting APIs directly from the application source code, this method reduces the number of redundant APIs which will reduce the analysis time and improve detection efficiency.

2) We build 3 different detection models using machine learning algorithms based on each data sets. Machine learning methods can predict the type of data according to its training model. Thus, not only can we detect known malwares, but unknown malwares can also be detected.

3) We propose a method to build chronological API data sets using statical analysis methods and use LSTM to analyze them. Compared to traditional data sets: Boolean data sets, frequency data sets, and chronological API data sets are more accurate in characterizing an application.

The rest of this paper is organized as follows. Section 2 introduces the control flow graph construction and gives an overview of our detection architecture. Section 3 presents our three detection models: API usage detection model, API frequency detection model and API sequence detection model. Evaluation and comparison of these three detection models are presented in section 4. Section 5 discusses related works and future works. Section 6 concludes the paper.

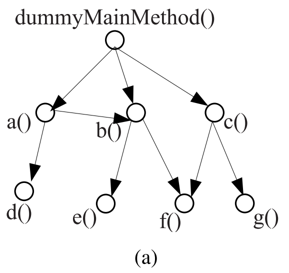
II. PRELIMINARIES

A. CFG CONSTRUCTION

We use FLOWDROID to do this job. FLOWDROID is an open source framework which performs the statical taint analysis for Android applications. It builds a precise model of the Android's lifecycle, allowing the analysis to properly handle callbacks invoked by the Android framework, while context, flow, field and object-sensitivity allow the analysis to reduce the number of false alarms [16]. The CFG builds the android application for accomplishing the taint analysis based on our method, but the taint analysis is useless for us as our aim is not at privacy leak. So, we modified the source code of FLOWDROID, replacing the taint analysis with our method.

A control flow graph represents all the flows of control that may arise during program execution [17]. In our methods, CFG is a directed graph $G = (N, E, entry)$ presented in figure 1(a), while figure 1(b) represents the corresponding source code, where N is a node set, and each API in the program (including system API and user-defined API) represents one node; E is the edge set and $E = \{< n_1, n_2 > |n1, n2 \in N\ and\ n2\ may\ be\ executed\ immediately\ after\ n1\}$; and entry is the entry node of the program. Each java application has an entry function *main ()*, so, we can treat *main()* as an entry of a Java application. Each corresponding Android application contains *onCreate ()* in a package *android.app.Application* which can be used as an entry function of an Android application.

As Figure 1(a) shows, one circle represents an element of node set N; an arrow represents the calling relation. For example, an arrow from *a()* to *b()* means *a()* calls *b()* in the source code shown in Figure 1(b). So, corresponding to this graph, CFG will be represented as (N, E, entry) where the node set $N = \{dummyMainMethod(), a(), b(), c(), d(), e(), f(), g()\}$, the edge set $E = \{< dummyMainMethod(), a() >, < dummyMainMethod(), b() >, < dummyMainMethod(), c() >, < a(), b() >, < a(), d() >, < b(), e() >, < b(), f() >, < c(), g() >, < c(), f() >\}$ and the entry is



```
public class MainActivity extends ActionBarActivity{
    protected void conCreate(Bundle savedInstance State){
      a(); b();c();}
    public void a(){
      b();d();   }
    public void b(){
      e();f();   }
    public void c(){
      g();  f();   }
      ...
}
```

(b)

FIGURE 1. A simple CFG and its source code. (a) a CFG. (b) source code.

dummyMainMethod() which is an alias for *onCreate()* in order to relate to the Java *main()* method.

B. INFORMATION EXTRACTION

Usually, malicious applications use dangerous APIs to achieve certain targets. For example, privacy leaking will happen by using SMS APIs to get private information and sending them though a network with network APIs. This kind of malware can be detected by analyzing the usage of APIs. However, some malicious behavior will use normal APIs many times. For example, DOS attack will happen by constantly using network APIS. Thus, by analyzing the frequency of APIs, these kinds of malwares will be detected. Although this type of information helps to detect malware, it is not accurate enough. A benign application will use the same APIs intermittently during the application lifetime which makes it difficult to detect by analyzing the frequency. However, by analyzing the order of APIs, we can easily solve this problem. What's more, by analyzing the chronological API sequence, we will obtain its behavior.

Therefore, in our methods, we focus on three kinds of information in an application: the usage of APIs (which APIs the application uses), the API frequencies (how many times the application uses APIs) and API sequence (the order the application uses APIs). All of this information can be extracted from CFG and will be presented in the following forms.

1) API USAGE FORM

In this form, the information will be presented as a set $K = \{API_1, API_2, API_3 \ldots API_n \ldots\}$.

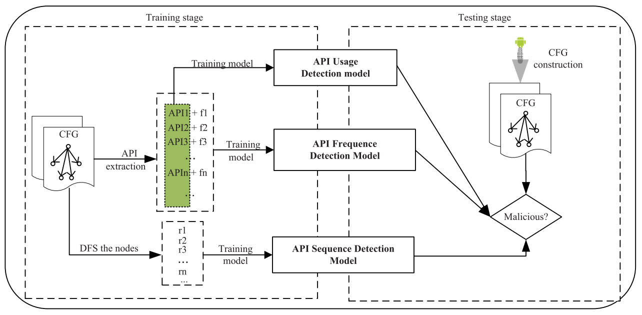


FIGURE 2. Detection architecture.

2) API FREQUENCIES FORM

In this form, we use a set of key value pairs $F = \{< API_1, f_1 >, < API_2, f_2 > \ldots < API_n, f_n > \ldots\}$,where API_n indicates one API in set K, and f_n indicates the times that API_n appears in CFG to express the information.

3) API SEQUENCE FORM

A chronological API sequence $R = \{r_1, r_2, \ldots, r_n \ldots\}$ (each r represents an API) which means if $i < j$, API_i will be called earlier than API_j in the executed path of the application used to express the information.

C. DETECTION ARCHITECTURE AND OVERVIEW

The architecture of our detection architecture is shown in Figure 2. In our methods, we train 3 detection model based on the features in CFG (section 2.2): API Usage Detection Model, API Frequency Detection Model and API Sequence Detection Model.

1) TRAINING STAGE

The data we train for this stage are the key set $K = \{API_1, API_2, API_3 \ldots API_n \ldots\}$. The key value pair $F = \{< API_1, f_1 >, < API_2, f_2 > \ldots < API_n, f_n > \ldots\}$ and API sequence $R = \{r_1, r_2, \ldots, r_n \ldots\}$ of CFGs are extracted from malicious applications and benign applications. We train a 2-class classification model for different data sets with different algorithms.

2) DETECTION STAGE

For each new arrived APK, the CFG will be constructed and K, F and R will be obtained based on the CFG. We use the three models to classify the K, F and R. If the result of the classification is malicious, we consider that the APK is malicious.

III. MALWARE DETECTION

We use the three detection models to detect malware. In the rest of this section, we introduce the methods to build the three models in detail and how to detect malware with the models.

A. API USAGE DETECTION MODEL

1) DATASET Construction

In our CFG, the node set N contains all of the nodes of CFG and each node is presented as an API. So, we can get the API usage of CFG simplify by traversing its node set. For example, in the CFG shown in Figure 1, the node set $N = \{dummyMainMethod(), a(), b(), c(), d(), e(), f(), g()\}$. After we traverse this set, we obtain the API's usage set $K = \{dummyMainMethod, a, b, c, d, e, f, g\}$ (we only take the API name into considerarion).

As the number of user-defined APIs are infinite and the user-defined APIs will always call the system API to accomplish a certain task, we remove the user-defined APIs during analysis and only the system APIs remain (here the system API in our methods is the API that Java Developer's Kit (JDK) and Software Development Kit (SDK) contain). The total number of system APIs is a constant. Let $S = \{API_1, API_2, \ldots API_m\}$ be the total system API set (suppose the total number of system APIs is m). There are two methods to obtain S : first, looking it up in the JDK and SDK documents and collecting the system APIs manually; second, using the self-learning methods. In our methods, we choose the second method.

2) SELF-LEARNING METHODS

We build a Boolean data set based on S and K. In the Boolean data sets, the attributes are the APIs in set S plus a classification label, and the value is a matrix of 0 and 1 (we use 0 to indicate that the API does not appear in set K while 1 indicates that it appears. As for the classification value, 0 represents benign and 1 represents malicious). For example, assuming $S = \{a, b, c, d, e, f, type\}$, $K_1 = \{a, c, f\}$ and is extracted from a malicious CFG, $K_2 = \{b, d, e\}$ and is extracted from a benign CFG, then the vector of K_1 will be $\{1, 0, 1, 0, 0, 1, 1\}$ and the vector of K_2 will be $\{0, 1, 0, 1, 1, 0, 0\}$.

In each CFG, we can obtain a $K = \{API_1, API_2, \ldots API_n\}$; obviously, $K \subseteq S$. With the number of CFG increasing, we will get numerous K. We combine all of these K together and remove the duplicated APIs. In this way, the total system API set S is obtained.

3) TRAINING STAGE

Discrete and with each attribute having only 2 values, our data sets can be analyzed with a decision tree algorithm. C4.5 is an algorithm used to generate a decision tree which can be used for classification. In the top ten algorithms of data mining, the C4.5 algorithm ranks first [18]. So, we choose the C4.5 algorithm to train a 2-class classification model. In the training stage, we build a decision tree based on the Boolean data sets. There are 2 classes in our data sets (malicious and benign), and each attribute has 2 kinds of values (0 or 1). Assume the data set is D, and the class set is $C (C = \{malicious, benign\})$. In the first step, we calculate the information gain ratio of each attribute in S and choose the attribute with the largest value as the root of the tree. Then, we partition D into corresponding subsets D_1, D_2 based on the value of the chosen attribute. We apply the same procedure recursively to each subset until each subset belongs to the same class. After the recursion is done, a decision tree is built.

4) TESTING STAGE

To test whether an incoming K is malicious or benign, we use the decision tree built during the training stage to test: Let $K = \{API_1, API_2 \ldots API_n, \ldots\}$. Suppose the root of the tree is attribute API_k, we choose one branch of the tree root according to the value of API_k in set K. We apply the same procedure recursively to each subset and finally the classification result of K will come out.

B. API FREQUENCY DETECTION MODEL

1) DATA SET CONSTRUCTION

Identically, we only construct the data sets of system APIs. Since our CFG is a directed graph [19], we use the BFS (breadth-first search) algorithm [20] to traverse its node. The result of each traversal is stored in a key-value pair form where the key is the node represented as API and the value includes the frequencies where the node appears. As an example in figure 1, suppose the frequencies set is $F = \{<API_1, f_1>, <API_2, f_2> \ldots <API_n, f_n> \ldots\}$. After first traversal, we got a(), b(), c(), so $F = \{<a, 1>, <b, 1>, <c, 1>\}$; after the second traversal, d(), b(), e(), f(), f() and g() come out, therefore F is updated to $\{<a, 1>, <b, 2>, <c, 1>, <d, 1>, <e, 1>, <f, 2>\}$. We take the same methods to update F until the traversal finishes and finally $F = \{<a, 1>, <b, 2>, <c, 1>, <d, 1>, <e, 1>, <f, 2>, <g, 1>\}$.

We built frequencies datasets based on F and S (the whole system API set). In the data sets, the attributes are the same as Boolean data sets but the attribute values are no longer 0 and 1. The attribute value is a matrix of 0 and an integer number indicates the frequencies of the API. For example, assuming $S = \{a, b, c, d, e, f, type\}$, $F_1 = \{<a, 10>, <c, 50>, <f, 25>\}$ and is extracted from a malicious CFG; $F_2 = \{<b, 20>, <d, 15>, <e, 22>\}$ and is extracted from a benign CFG, then the corresponding vector of V_1 will be $\{10, 0, 50, 0, 0, 25, 1\}$ and the vector of V_2 will be $\{0, 20, 0, 15, 22, 0, 0\}$(here 0 indicates benign CFG and 1 indicates malicious CFG).

2) TRAINING STAGE

As each attribute value in our data set is a continuous integer, it will be complicated for the decision tree algorithm to build the model. However, DNN (deep neural network) is able to learn patterns in the numerical vectors form and can classify or cluster raw input. Therefore, we use the DNN algorithm instead. In this stage, a 2-class classification model is trained with the DNN algorithm based on the training sets. That is, for each vector $V = \{f_1, f_2, \ldots f_n, label\}$. The algorithm will calculate and update the coefficients based on input f_k to make the results of the label equal to the input label.

3) TESTING STAGE

For each vector V, we calculate the classification result with input f_k and the coefficients trained in the training stage, and compare the result with the input label. We use

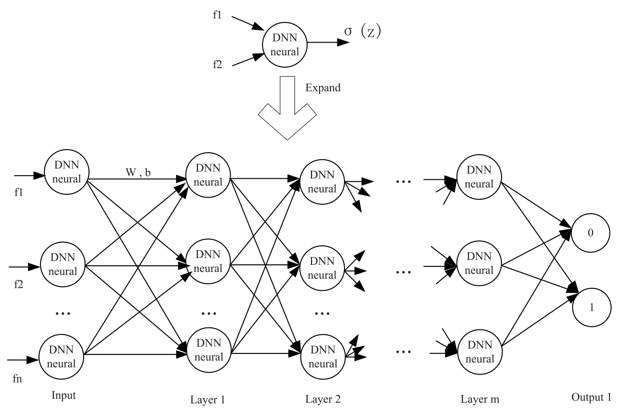


FIGURE 3. A DNN neural and network.

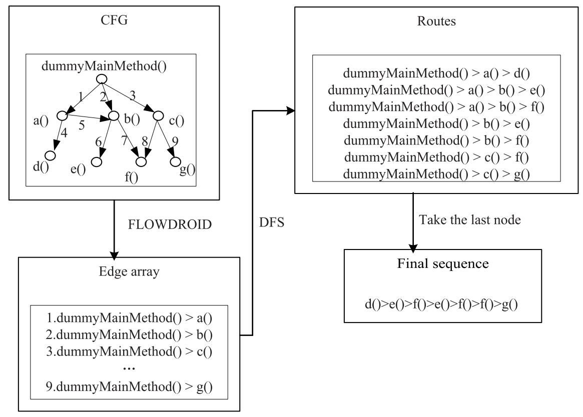

CFG
dummyMainMethod()
a()
b()
c()
d()
e()
f()
g()
FLOWDROID
Edge array
1.dummyMainMethod() > a()
2.dummyMainMethod() > b()
3.dummyMainMethod() > c()
...
9.dummyMainMethod() > g()
DFS
Routes
dummyMainMethod() > a() > d()
dummyMainMethod() > a() > b() > e()
dummyMainMethod() > a() > b() > f()
dummyMainMethod() > b() > e()
dummyMainMethod() > b() > f()
dummyMainMethod() > c() > f()
dummyMainMethod() > c() > g()
Take the last node
Final sequence
d()>e()>f()>e()>f()>f()>g()

FIGURE 4. Route construction process.

standard classification metrics such as Precision, Recall and F-measure to test the performance of the model.

4) THE DNN APPROACH

DNN is part of the broad field of AI, which is the science and engineering of creating intelligent machines that have the ability to achieve goals like humans do [21]. Figure 4 shows a single DNN neural and the expanding network.

The top of Figure 3 shows a single DNN neural. In the DNN network, each DNN neuron maintains state and passes it to the next neural. The output state is calculated with the input, weights values and bias which satisfy $z = \sum W_i \times f_i + b$ plus an activation function $\sigma(z)$ where f_i is the input and $\sigma(z)$ is the output state. An h-layer DNN network is shown in the button of Figure 4. DNN network will calculate a proper W and b to make the output equal to or very close to the sample output as much as possible during training. In other words, given a $V = \{f_1, f_2, \ldots f_n, label\}$. DNN will calculate the label based on $\{f_1, f_2, \ldots f_n\}$ using some mathematical formula expressed with W and b, i.e. $z = \sum W_i \times f_i + b$. After each calculation step, W and b will be updated to a proper value to make the calculated label equal to or very close to the input label.

After the training is done, we can predict the output for an input $\{f_1, f_2, \ldots f_n, label\}$. By comparing the output with the input label, we know whether the prediction is correct.

C. API SEQUENCE DETECTION MODEL

In fact, an application will call a set of APIs sequentially to accomplish a certain function. These chronological API sequences are the best for characterizing an application. In the rest of the manuscript, we introduce how we construct and analyze the chronological API sequences.

Figure 4 shows the process how we construct an API sequence. In FLOWDROID, each API is represented as a class MethodOrMethodContext. The edge of CFG is presented as two APIs: source API and target API, which are stored in an array. When FLOWDROID analyzes an application, it will store the edge in the order that the API appears in the application. So, the edges in the array are chronological. We use DFS (depth-first search) [22] to traverse the array. In each traversal, we obtain a route of APIs call a sequence. After all the traversala are done, a set of the route is obtained. We take the last API of each route in order and combine them to form a chronological route. Let us take the CFG in figure 1 as an example; after the traversal is done, we get 7 routes:

- $dummyMainMethod() > a() > d()$
- $dummyMainMethod() > a() > b() > e()$
- $dummyMainMethod() > a() > b() > f()$
- $dummyMainMethod() > b() > e()$
- $dummyMainMethod() > b() > f()$
- $dummyMainMethod() > c() > f()$
- $dummyMainMethod() > c() > g()$

1) ROUTE CONSTRUCTION

Then, we take the last API in each route in order, combine them together and we get a chronological route of the CFG i.e. $d() > e() > f() > e() > f() > f()$. Here, we need to point out that:

1) In general, the last API in each route is the system API, as user-defined APIs will always call the system APIs. This means the chronological route of the CFG is formed from system APIs.

2) The edges in the array are chronological, and when we use the DFS algorithm to traverse the array, the earlier-called API should be traversed earlier. The final route is chronological, and corresponding to our example, a() is called earlier than b(), b() is called earlier than c(), and d() is called earlier than b()...

2) ROUTE DIGITIZING

Let $R = \{API_1, API_2, API_3, \ldots, API_n\}$ be the chronological route of CFG, and let S be the whole system API set as any API_k in R must be an element of S. We use the index of API_k in S to present API_k. For example, assuming $R = \{a, e, c, d\}$ and $S = \{a, b, c, d, e, f\}$, then we can translate R into digital form $\{1, 5, 3, 4\}$ (the index of the first element in S is 1).

3) DATA SET CONSTRUCTION

We build a chronological data sets with R, S and the class the corresponding CFG belongs to. That is, assuming $R_1 = \{a, b, c, f\}$ and is extracted from a malicious CFG and $R_2 = \{e, e, a, b\}$ and is extracted from a benign CFG, $S = \{a, b, c, d, e, f\}$, then the vector of R_1 in our data set is $\{1, 2, 3, 6, 1\}$ and the vector of R_2 is $\{5, 5, 1, 2, 0\}$(here 0 indicates benign and 1 indicates malicious). We split the data sets into training sets and tests sets based on the split ratio.

4) TRAINING STAGE

We use the LSTM algorithm to train a 2-class classifier based on the training set. That is, for each input vector $R = \{r_1, r_2, \ldots, r_n, label\}$. The algorithm will calculate and update the coefficients based on the input rk to make the result equal to or close to the input label.

5) TESTING STAGE

For each input $R' = \{r_1, r_2, \ldots, r_n, label\}$ in the testing set, we calculate the classification result with input r_k and the coefficients trained during the training stage, and compare the result with the input label. We use standard classification metrics such as Precision, Recall and F-measure to test the performance of the model.

6) THE LSTM APPROACH

LSTM is a recurrent neural network (RNN) architecture that has been designed to address the vanishing and exploding gradient problems of conventional RNNs. Unlike feed forward neural networks, RNNs have cyclic connections making them powerful for modeling sequences [23]. Thus, our methods use LSTM for malware detection from an API sequence.

Our LSTM network is trained to maximize the probability of having a classification result 0 or 1 on the training data set. Which means, it learns a probability distribution $\{\Pr(label = 0|r_1, r_2, \ldots, r_n)\ \Pr(label = 1|r_1, r_2, \ldots, r_n)\}$ which maximizes the probability.

Figure 5 shows how we use LSTM to train our datasets. The left is a single LSTM cell. The state of an LSTM cell will be passed to the next cell together with the input data to calculate the new state of the next cell. This is the way that historical information is passed on and maintained in a single LSTM block and why LSTM can process sequential datasets.

After expanding a single LSTM cell by layer, a series of LSTM cells form a one layer LSTM network shown in the middle of Figure 5. A hidden vector Ht and a cell state vector St is maintained in each cell and will be passed to the next cell to calculate its state. In our methods, one LSTM cell represents an API in a route vector. Therefore, one layer of the LSTM network consists of l (where l is the length of the route) LSTM cell.

By unfolding a layer LSTM network, we obtain a multi-layer LSTM network shown in Figure 5. In our methods, we choose different numbers of layers to test its

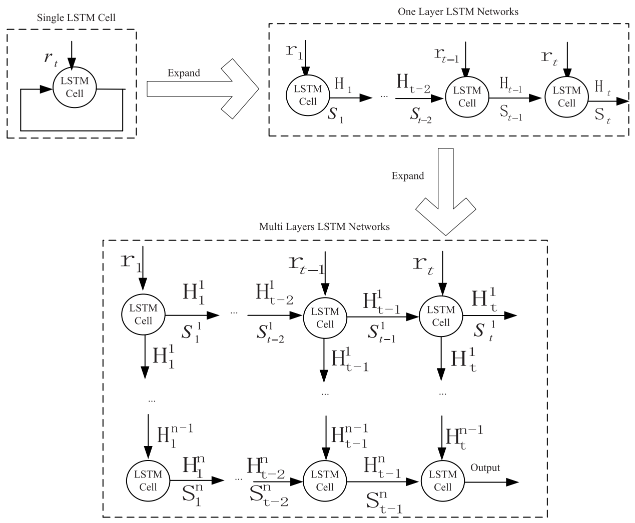


FIGURE 5. **API sequence detection model with LSTM.**

performance. We use the hidden state of the previous layer as the input of each corresponding LSTM cell in the next layer.

In the multi-layer LSTM network, each LSTM cell will do the following things with the input data(r_t) and the output of previous cells(H_{t-1}) during training: 1) calculate and maintain its states S_t; 2) calculate output H_t. The calculation is done by using some gate function (in the LSTM network, there are three gates in each cell, i.e. the input, output and forget gates that provide continuous analogues of write, read and reset operations for the cell). Each gating function is parameterized by a set of weights to be learned.

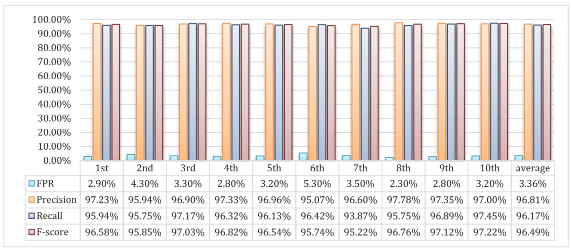
The whole process of the training stage is to calculate proper weight values to make the output equal to or very close to the sample output as much as possible with the input data of the training data set. In our methods, the input data is the API sequence and the output is the class (0 or 1) that the API sequence belongs to.

After training is done, we can predict whether the API sequence is malicious or benign.

D. COMBINATION METHOD

The most widely used method for model combination is Boosting, a machine learning ensemble meta-algorithm for improving accuracy, which converts weak learners to strong ones. As is discussed above, each of the detection models can respectively extract API features and give prediction results. However, the performance in Sec.IV shows that the three detection models are far better than weak learners. Thus, we adopt soft voting to implement the combination method.

To implement the combination detection, we extract three kinds of dataset, the API usage data, the API frequency data and the API sequence data, from Android package files. The result generated by a model is either 1 or 0 which represents a malicious or benign prediction. The classification accuracy of each model is used as its weight and the weighted sum of three models serve as the final detection result. The package



	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	average
FPR	2.90%	4.30%	3.30%	2.80%	3.20%	5.30%	3.50%	2.30%	2.80%	3.20%	3.36%
Precision	97.23%	95.94%	96.90%	97.33%	96.96%	95.07%	96.60%	97.78%	97.35%	97.00%	96.81%
Recall	95.94%	95.75%	97.17%	96.32%	96.13%	96.42%	93.87%	95.75%	96.89%	97.45%	96.17%
F-score	96.58%	95.85%	97.03%	96.82%	96.54%	95.74%	95.22%	96.76%	97.12%	97.22%	96.49%

FIGURE 6. Accuracy of API usage detection model.

file whose weighted sum is more than half of the total weight will be predicted malicious, otherwise benign.

IV. PERFORMANCE EVALUATION

Our experiment is carried out with 10010 benign applications from AndroZoo [13] and 10683 malwares from AMD [14], [15]. In order to ensure the accuracy of malicious and benign applications, we send each sample application to the common 4 anti-virus scanners (McAfee, 360 Security Guard, Kingsoft Antivirus, Norton). We flag the application as malicious when it is detected by one or more of these scanners. In the rest of this section, we show evaluations of each detection model. Since all of our detection models are classification models, we use standard classification metrics such as False Positive Rate (FPR), Precision, Recall and F-measure. $FPR = \frac{FP}{FP+TN}$ (FP stands for false positive) shows the ratio between the number of wrongly detected benign applications and the total number of the benign applications; $Precision = \frac{TP}{TP+FP}$ (TP stands for true positive) shows the percentage of malware among all of the applications detected; $Recall = \frac{TP}{TP+FN}$ being detected; and $F-score = \frac{2\times Precision\times Recall}{Precision+Recall}$ is the harmonic mean of the two.

A. API USAGE DETECTION MODEL

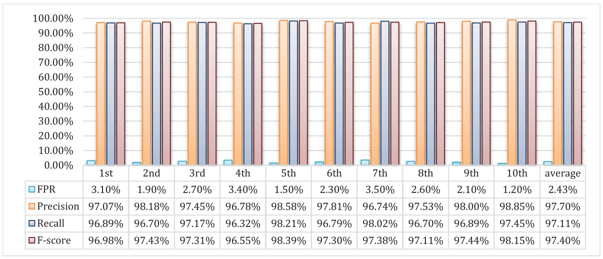
We first split the total data into 10 parts, each of which consists of 1000 sets of benign API usage data and 1060 sets of malicious API usage data without repetition (constructed from benign and malicious applications respectively). In our 10 iterations of cross validation one part of the dataset serves as the test data and the other are the training data.

We do the same process to construct the test sets. The performance with different split ratios is shown in Figure 5.

In total, the model performs well as the average precision of API usage model is 96.81% with a worst F-score of 95.22% and a worst FPR of 5.30%. From the result we can obtain our detection model to detect malware with great feasibility.

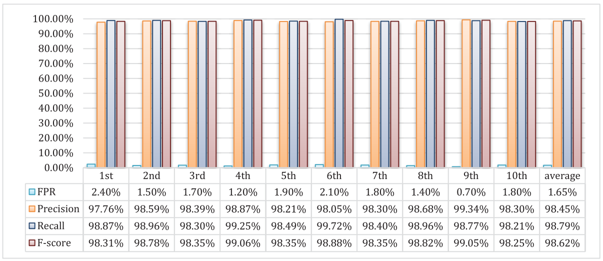
B. API FREQUENCY DETECTION MODEL

To test the model, we first vary the number layers with 4000 sets of training data and 2000 test data. The ratio of malicious and benign applications is 1:1 in both of the dataset. Table 1 presents that the FPR comes to a low level when there are 8 hidden layers. With the number of layers increasing to 16 and 32, there is no obvious improvement in precision and recall. It is appropriate to set the number of layers to 8 in this model.



	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	average
FPR	3.10%	1.90%	2.70%	3.40%	1.50%	2.30%	3.50%	2.60%	2.10%	1.20%	2.43%
Precision	97.07%	98.18%	97.45%	96.78%	98.58%	97.81%	96.74%	97.53%	98.00%	98.85%	97.70%
Recall	96.89%	96.70%	97.17%	96.32%	98.21%	96.79%	98.02%	96.70%	96.89%	97.45%	97.11%
F-score	96.98%	97.43%	97.31%	96.55%	98.39%	97.30%	97.38%	97.11%	97.44%	98.15%	97.40%

FIGURE 7. Accurancy of the API frequency detection model.



	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th	average
FPR	2.40%	1.50%	1.70%	1.20%	1.90%	2.10%	1.80%	1.40%	0.70%	1.80%	1.65%
Precision	97.76%	98.59%	98.39%	98.87%	98.21%	98.05%	98.30%	98.68%	99.34%	98.30%	98.45%
Recall	98.87%	98.96%	98.30%	99.25%	98.49%	99.72%	98.40%	98.96%	98.77%	98.21%	98.79%
F-score	98.31%	98.78%	98.35%	99.06%	98.35%	98.88%	98.35%	98.82%	99.05%	98.25%	98.62%

FIGURE 8. Accuracy of different split ratios.

We carry out the same experiments on our API frequency dataset constructed from the same Android applications as the API usage detection model does.

As is shown in Figure 7, the average precision reaches 97.70% with a worst F-score of 96.55% and a worst FPR of 3.50%. In average, the API frequency detection model successfully reduces the FPR compared with API usage detection model, which means fewer benign applications are predicted as malware.

C. API SEQUENCE DETECTION MODEL

In Table 1, we also vary the value of layer numbers of API sequence model. Similarly, 8 layers are suitable.

Figure 8 provides the result of the similar experiment we have done in other two models. The average precision of API usage model is 98.45% with a worst F-score of 98.25% and a worst FPR of 2.40%. In total, the performance is fairly stable with respect to different values.

D. COMBINATION AND COMPARISON OF THREE MODELS

In this section, we utilize a voting classifier to combine these three models. Table 2 provides the FPR, precision, recall and F-score in average of the ensemble model along with three sub-models.

We compare the performance of these ensemble model and three sub-models from two aspects: accuracy and resource consumption. In the resource consumption test, we use a windows resource monitoring tool Perfmon to monitor the CPU and memory consumption when the models run.

1) ACCURACY COMPARISON

As illustrated in Table 2, the API sequence model has the best result Among these sub-models while the API usage

TABLE 1. The accuracy of DNN and LSTM with variation of layer numbers.

layers	API Frequency Detection Model				API Sequence Detection Model			
	FPR	Precision	Recall	F-score	FPR	Precision	Recall	F-score
4	6.20%	94.72%	98.23%	96.44%	7.60%	95.64%	99.67%	97.60%
8	2.50%	97.84%	98.56%	98.20%	0.70%	99.61%	99.82%	99.70%
16	2.50%	98.02%	99.28%	98.65%	2.40%	98.67%	100.00%	99.30%
32	1.70%	97.80%	99.50%	98.64%	0.50%	99.78%	100.00%	99.80%

TABLE 2. Comparison of model accuracy.

model	FPR	precision	recall	F-score
ensemble model	1.58%	99.15%	98.82%	98.98%
API usage model	3.36%	96.81%	96.17%	96.49%
API frequency model	2.43%	97.70%	97.11%	97.40%
API sequence model	1.65%	98.45%	98.79%	98.62%

detection model has the worst. However, all models perform well with each F-score value over 96% and FPR under 4%. Based on these three sub-models, the ensemble model successfully makes a slight improvement that the precision reaches 99.15% with the FPR of 1.58%.

2) RESOURCE CONSUMPTION COMPARISON

Perfmon obtains the resource that all of the applications consume, including our models. In order to get our model's resource consumption, we used Perfmon to monitor the application's resource consumption when there are no model runs. Then, the accurate model's resource consumption is obtained by the model's resource consumption minus the application resource consumption.

4 screenshots in Figure 11 show the result. The sub-figure (a) shows the application's resource consumption when there are no models running. Obviously, the CPU consumption is around 0 and the memory consumption is around 17%. The sub-figure (b) shows the API usage detection model resource consumption. From the result we can obtain that the CPU and memory consumption is stable with around 27% CPU consumption and 20% memory consumption (actually 3%). The sub-figure (c) and (d) show the API frequency model and API sequence model resource consumption. In these two sub-figures, the CPU consumption is high in the beginning, then it goes down and becomes stable. We know that in the model constructing stage there are more computing steps, that is why the consumption is higher in the beginning. After the model is constructed, the testing stage does less computing and becomes stable.

In total, after the model is constructed, our detection model consumes little and has a stable resource, which allows it to be applied to mobile devices as well.

E. COMPARISON WITH OTHER METHODS

In order to prove our detection methods are effective, we compare our methods with other well-known detection methods by experiment. Table 1 shows the experiment result.

The 10600 malicious samples and 10000 benign samples we selected respectively come from AMD and AndroZoo. We have tested each sample to the common 4 anti-virus scanners :McAfee, 360 Security Guard, Kingsoft Antivirus and Norton to ensure the reliability. From TABLE 3, we can obtain that our ensemble detection model has the highest accurate rate.

TABLE 3. Comparison with other methods.

tool name	feature	algorithm	F-score	FPR
our combination method	API	C4.5 DNN LSTM	98.98%	1.58%
Mclaughlin [12]	opcode	CNN	89.50%	6.72%
DroidDetective [24]	permission	K-map	87.67%	7.67%
Yerima [25]	API and permission	Random forest	97.42%	4.33%

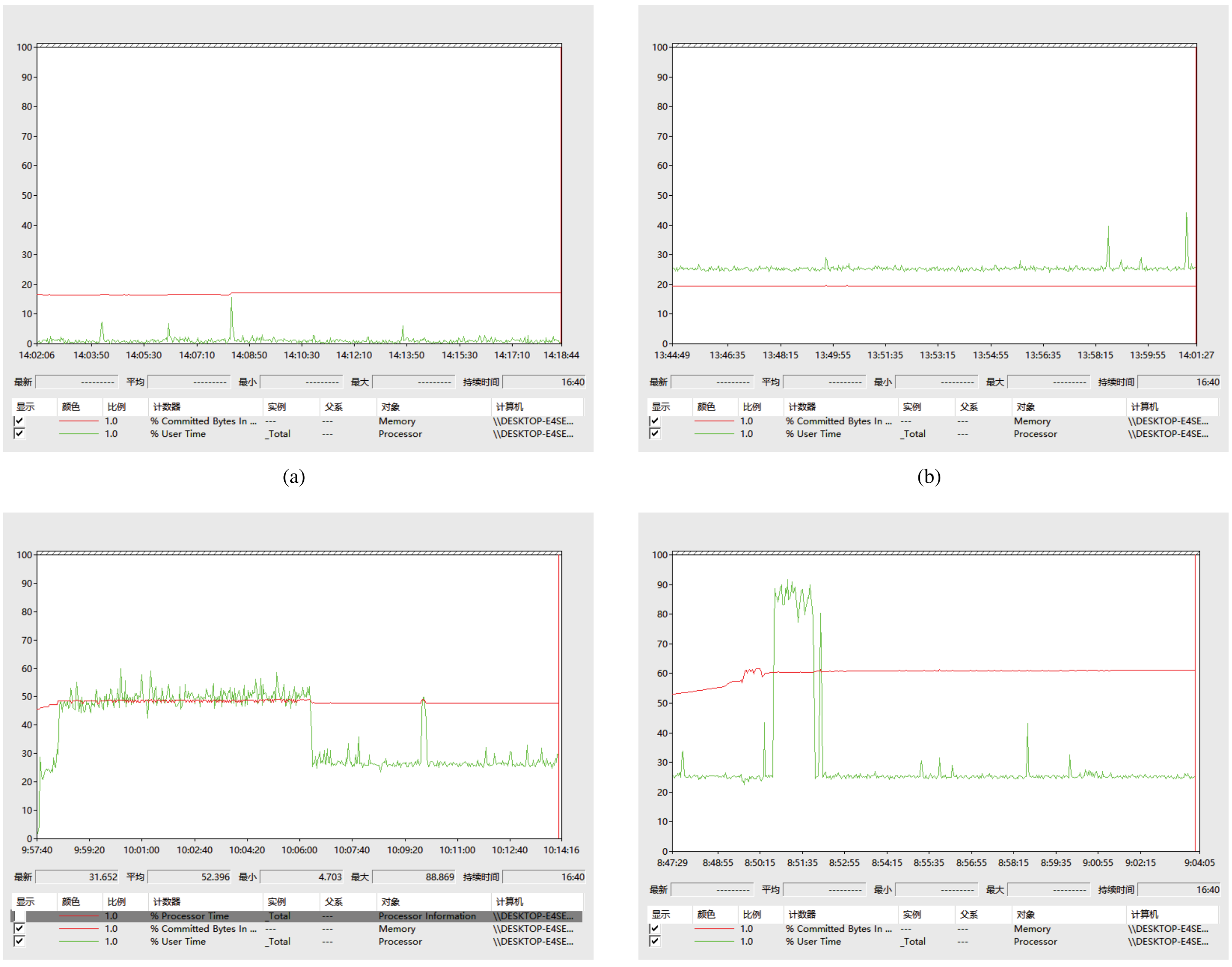
V. RELATED WORKS

Numerous methods have been proposed to detect android malwares. And which can be basically divided into two categories

A. STATIC DETECTION METHODS

Static methods is the first approach proposed to detect android malwares. Among which, many use permission based methods: Mahindru and Singh [26] detect malwares based on the dynamic permissions methods; Adrienne *et al.* [27] build a tool to detect over privilege in compiled Android applications; AndroidLeaks [28] automatically finds potential leaks of sensitive information based on maps of permissions and APIs. Although simple and efficient, the permission based methods are not practical as modern Android applications must request the permission before installing. What's more, the permissions should be granted in application running time.

Thus, API based approaches have been proposed. DREBIN [29] build a SVM detection model based on APIs and many other information of an application. R-PackDroid [30] detects ransom wares based on system API package. Fan *et al.* [31] build a detection system based on the API log information. Although they are accurate and efficient, the analyzed APIs are extracted from either source code or from de-compiled files. Compared to APIs extracted from CFG (as our methods do), they contains many redundant



(a)

最新	---------	平均	---------	最小	---------	最大	---------	持续时间	16:40

显示	颜色	比例	计数器	实例	父系	对象	计算机
✓	🟥	1.0	% Committed Bytes In ...	---	---	Memory	\\DESKTOP-E4SE...
✓	🟩	1.0	% User Time	_Total	---	Processor	\\DESKTOP-E4SE...

(b)

最新	---------	平均	---------	最小	---------	最大	---------	持续时间	16:40

显示	颜色	比例	计数器	实例	父系	对象	计算机
✓	🟥	1.0	% Committed Bytes In ...	---	---	Memory	\\DESKTOP-E4SE...
✓	🟩	1.0	% User Time	_Total	---	Processor	\\DESKTOP-E4SE...

(c)

最新	31.652	平均	52.396	最小	4.703	最大	88.869	持续时间	16:40

显示	颜色	比例	计数器	实例	父系	对象	计算机
	🟥	1.0	% Processor Time	_Total	---	Processor Information	\\DESKTOP-E4SE...
✓	🟥	1.0	% Committed Bytes In ...	---	---	Memory	\\DESKTOP-E4SE...
✓	🟩	1.0	% User Time	_Total	---	Processor	\\DESKTOP-E4SE...

(d)

最新	---------	平均	---------	最小	---------	最大	---------	持续时间	16:40

显示	颜色	比例	计数器	实例	父系	对象	计算机
✓	🟥	1.0	% Committed Bytes In ...	---	---	Memory	\\DESKTOP-E4SE...
✓	🟩	1.0	% User Time	_Total	---	Processor	\\DESKTOP-E4SE...

FIGURE 9. Comparison of the resource consumption.

APIs. In our methods, we also analyze the APIs, but we extracted them from CFG of the application.

Many approaches focus on the structure features of CFG. Gascon *et al.* [32] build a structural feature map based on function call graphs and use a linear SVM to analyze them. Cesare and Xiang [33]construct a malware signature by the set of control flow graphs and detect with feature matching techniques. Kruegel *et al.* [34] detect worm applications based on the structural of the CFG of a worm application and use graph coloring technique to classify the structures. Although these methods performs well, the structure of a CFG is complex. Besides the features database should be huge enough to cover all the features and be updated regularly and what's more they can only detect known type malwares. So, in our methods, we only extracted APIs from CFG and let machine learning methods do the analyzing job.

Machine learning methods have advantages in detecting unknown malwares. Hou *et al.* [35]develop an intelligent malware detection system using cluster-oriented ensemble classifiers. Atici *et al.* [36] construct a CFG in grammar forms and use various classification methods to analyze them. DroidScribe [37] builds a classification models based on run-time behavior. McLaughlin *et al.* [12] utilize CNN to build an android malware detection system based on opcode sequence from a disassembled program. Though, these methods performs well, they cannot detect the chronological features of applications. The chronological features may help to understand what an application does.

B. DYNAMIC DETECTION METHODS

TaintDroid [38] tracks the sensitive information of an application in real time to protect the privacy of the users. DroidScope [39] monitors application behaviors at three different platform layers. Chen *et al.* [40] generate API CFG based on the real-time behaviors of software and use data mining methods to analyze them. Although able to detect malwares at runtime, they cannot be deployed in the smart phone to detect malwares directly. Besides, in order to get as much information as possible, one should trigger all the application functions manually.

VI. CONCLUSIONS

In this paper, we present methods to detect Android malwares. We de-compile the android applications and construct the CFG of each application. We construct 3 kinds of system API data sets: API usage data sets (indicates which API the CFG contains), API frequency data sets (indicates how many times the CFG uses corresponding API) and API sequence data sets (indicates what the API sequence appearing in CFG is) sets based on the CFG. As far as I know, we are the first to construct the API sequence data sets of an android application. For each data sets, we build a 2-class classification model and use the model to detect whether the incoming application is malicious. We evaluate the accuracy of each model using standard classification metrics - Precision, Recall and F-score - and compare the performance of these three model. With the combination method, an ensemble model is constructed and achieves 98.98% detection precision.

In the future we would but are not limited to do the following works:

- Addressing the malicious APIs position in source code. In DeepLog [23], Du *et al.* build a workflow model for anomaly log detection and with which the uses can diagnose the anomaly. Inspired by this approach, we can build an APIs workflow to diagnose malwares and find the root cause of malicious behavior.
- Detecting the malicious families that malwares belong to. In this paper, the models we build are 2-class classification models, which means we can only determine whether the application is malicious or not. In the future, we will build a multi-class classification model to determine which malicious family the application belong to if detected malicious.

ACKNOWLEDGMENT

The authors would like to thank the editor and the anonymous referees for their constructive comments.

REFERENCES

[1] Gartner. (2017). *Worldwide Sales of Smartphones Grew 9 Percent in First Quarter of 2017*. [Online]. Available: https://www.gartner.com/newsroom/id/3725117
[2] Symantec. (2018). *2018 Internet Security Threat Report*. [Online]. Available: https://www.symantec.com/security-center/threat-report
[3] Symantec, "Internet security threat report," Cupertino, CA, USA, Tech. Rep., 2015, vol. 20.
[4] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android security," *IEEE Security Privacy*, vol. 7, no. 1, pp. 50–57, Jan./Feb. 2009.
[5] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2012, pp. 95–109.
[6] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2012, vol. 25, no. 4, pp. 50–52.
[7] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time Android application auditing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2015, pp. 899–914.
[8] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices*, Oct. 2011, pp. 3–14.
[9] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. USENIX Security Symp.*, vol. 30, 2011, p. 88.
[10] Z. Ma, Y. Liu, Z. Wang, H. Ge, and M. Zhao, "A machine learning-based scheme for the security analysis of authentication and key agreement protocols," in *Proc. Neural Comput. Appl.*, Dec. 2018, pp. 1–13.
[11] Z. Ma, X. Wang, R. Ma, Z. Wang, and J. Ma, "Integrating gaze tracking and head-motion prediction for mobile device authentication: A proof of concept," *Sensors*, vol. 18, no. 9, p. 2894, 2018.
[12] N. McLaughlin *et al.*, "Deep Android malware detection," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2017, pp. 301–308.
[13] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories (MSR)*, May 2016, pp. 468–471.
[14] Y. Li, J. Jang, X. Hu, and X. Ou, "Android malware clustering through malicious payload mining," in *Research in Attacks, Intrusions, and Defenses*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds. Cham, Switzerland: Springer, 2017, pp. 192–214.
[15] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Cham, Switzerland: Springer, 2017, pp. 252–276.
[16] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
[17] J.-W. Jo and B.-M. Chang, "Constructing control flow graph for Java by decoupling exception flow from normal flow," in *Computational Science and Its Applications—ICCSA*, A. Laganá, M. L. Gavrilova, V. Kumar, Y. Mun, C. J. K. Tan, and O. Gervasi, Eds. Berlin, Germany: Springer, 2004, pp. 106–113.
[18] X. Wu *et al.*, "Top 10 algorithms in data mining," *Knowl. Inf. Syst.*, vol. 14, no. 1, pp. 1–37, 2008.
[19] B. A. Farbey, "Structural models: An introduction to the theory of directed graphs," *J. Oper. Res. Soc.*, vol. 17, no. 2, pp. 202–203, Jun. 1966.
[20] A. Bundy and L. Wallen, "Breadth-first search," in *Catalogue of Artificial Intelligence Tools*. Berlin, Germany: Springer, 1984, p. 13.
[21] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
[22] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.
[23] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2017, pp. 1285–1298.
[24] S. Liang and X. Du, "Permission-combination-based scheme for Android mobile malware detection," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2014, pp. 2301–2306.
[25] S. Y. Yerima, S. Sezer, and I. Muttik, "High accuracy Android malware detection using ensemble learning," *IET Inf. Secur.*, vol. 9, no. 6, pp. 313–320, Nov. 2015.
[26] A. Mahindru and P. Singh, "Dynamic permissions based Android malware detection using machine learning techniques," in *Proc. 10th Innov. Softw. Eng. Conf.*, Feb. 2017, pp. 202–210
[27] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, Oct. 2011, pp. 627–638.
[28] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *Proc. Int. Conf. Trust Trustworthy Comput.*, 2012, pp. 291–307.
[29] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 23–26.
[30] D. Maiorca, F. Mercaldo, G. Giacinto, C. A. Visaggio, and F. Martinelli, "R-PackDroid: API package-based characterization and detection of mobile ransomware," in *Proc. Symp. Appl. Comput.*, Apr. 2017, pp. 1718–1723.
[31] C.-I. Fan, H.-W. Hsiao, C.-H. Chou, and Y.-F. Tseng, "Malware detection systems based on API log data mining," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, Jul. 2015, pp. 255–260.
[32] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of Android malware using embedded call graphs," in *Proc. ACM Workshop Artif. Intell. Secur.*, Nov. 2013, pp. 45–54.
[33] S. Cesare and Y. Xiang, "Malware variant detection using similarity search over sets of control flow graphs," in *Proc. IEEE 10th Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Nov. 2011, pp. 181–189.

[34] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Recent Advances in Intrusion Detection*, A. Valdes and D. Zamboni, Eds. Berlin, Germany: Springer, 2006, pp. 207–226.

[35] S. Hou, L. Chen, E. Tas, I. Demihovskiy, and Y. Ye, "Cluster-oriented ensemble classifiers for intelligent malware detection," in *Proc. IEEE 9th Int. Conf. Semantic Comput.*, Feb. 2015, pp. 189–196.

[36] M. A. Atici, S. Sagiroglu, and I. A. Dogru, "Android malware analysis approach based on control flow graphs and machine learning algorithms," in *Proc. 4th Int. Symp. Digit. Forensic Secur. (ISDFS)*, Apr. 2016, pp. 26–31.

[37] S. K. Dash *et al.*, "Droidscribe: Classifying Android malware based on runtime behavior," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2016, pp. 252–261.

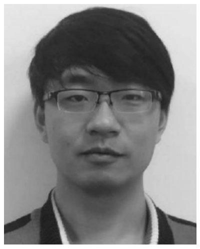
[38] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, p. 5, 2014.

[39] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. 21st USENIX Conf. Secur. Symp.*, 2012, pp. 569–584.

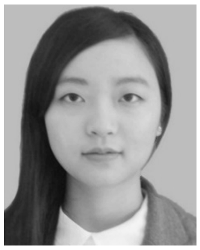
[40] Z.-G. Chen, H.-S. Kang, S.-N. Yin, and S.-R. Kim, "Automatic ransomware detection and analysis based on dynamic API calls flow graph," in *Proc. Int. Conf. Res. Adapt. Convergent Syst.*, 2017, pp. 196–201.

ZHUO MA received the B.S. degree in computer science and technology, and the M.S. and Ph.D. degrees in computer architecture from Xidian University, in 2003, 2006, and 2010, respectively, where he is currently an Associate Professor with the School of Cyber Engineering. He has published over 20 journal and conference papers. His research interests include information security and cryptography.

HAORAN GE received the B.S. degree in computer science and technology from Xidian University, in 2017, where he is currently pursuing the master's degree with the School of Cyber Engineering. His research interests include system security and machine learning.



YANG LIU received the B.S. degree in computer science and technology from Xidian University, in 2017, where he is currently pursuing the master's degree with the School of Cyber Engineering. His research interests include formal analysis of protocols and deep learning neural networks.



MENG ZHAO received the B.S. degree in information security from Jinan University, in 2016. She is currently pursuing the master's degree with the School of Cyber Engineering, Xidian University. Her research interests include blockchain and access control.

JIANFENG MA received the B.S. degree in computer science from Shaanxi Normal University, in 1982, and the M.S. and Ph.D. degrees in computer science from Xidian University, in 1992 and 1995, respectively, where he is currently a Professor with the School of Computer Science and Technology. He has published over 150 journal and conference papers. His research interests include information security, cryptography, and network security.