IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 8, 2019, accepted May 24, 2019, date of publication May 29, 2019, date of current version June 7, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2919796

An Efficient Android Malware Detection System Based on Method-Level Behavioral Semantic Analysis

HANQING ZHANG, SENLIN LUO, YIFEI ZHANG, AND LIMIN PAN
Information System and Security and Countermeasures Experiment Center, Beijing Institute of Technology, Beijing 100081, China

Corresponding author: Limin Pan (panlimin2016@gmail.com)

This work was supported in part by the Technology Innovation Program Major Projects of Beijing Institute of Technology under Grant 2011CX01015 and in part by the National 242 Program under Grant 2017A149.

ABSTRACT According to the recent report, 12 000 new Android malware samples will be generated every day. Efficient identification of evolving malware is an urgent challenge. Traditional methods based on structured features such as permissions and sensitive application programming interface (API) calls lack high-level behavioral semantics to detect evolving malware. The methods based on call graphs (CG) are good at behavioral semantic analysis but face the problem of huge time and space consumption, which leads to low detection efficiency. In this paper, we propose a novel Android malware detection method based on the method-level correlation relationship of application's abstracted API calls. First, we split each Android application's source code into separate function methods and just keep the abstracted API calls of them to form a set of abstracted API calls transactions. And then, we calculate the confidence of association rules between the abstracted API calls, which forms behavioral semantics to describe an application. Finally, we combine machine learning to identify the different behavioral patterns of malicious and benign apps to build the detection system. The results of our empirical evaluation show our system is competitive in terms of classification accuracy and detection efficiency. At dataset Drebin (benign 5.9K and malware 5.6K) and AMD (benign 20.5K and malware 20.8K), our system has achieved 96% and 98% detection results both in accuracy and F-measure. Compared with the state-of-the-art system in detecting evolving malware called MaMaDroid on the dataset of 6.0K benign and 20.5K malicious samples spanning from 2010 to 2017, our system achieves higher accuracy while improving detection efficiency by 15 times (2.9 s versus 45.7 s per sample).

INDEX TERMS Android malware detection, abstracted API call, association analysis, behavioral semantics, machine learning.

I. INTRODUCTION

In recent years, the popularity of smart phones is increasingly high and Android becomes an indispensable part of people's daily work and life. Due to the openness of free source, Android covers around 85% of worldwide smartphone market until 2018 [1]. At the same time, Android has also become a prime target for cyber-criminals. According to the recent report, in the whole year of 2018, 360 Internet Security Center intercepted about 4.342 million new samples of malware on mobile terminals, or about 12,000 new samples per day [2]. These malicious apps are created to perform different types of attacks such as stealing private information, sending message without user's permission, luring users to malicious websites, etc., which pose serious threats to smart phone users. In order to evade detection, malware continue evolving and are increasingly complex and diverse, and some notorious malicious apps have more than 50 variants, which brings a great challenge to detect them all. Therefore, the effective and efficient detection techniques are urgently needed to cope with the increasing sophistication of Android malware.

To address these challenges in Android malware detection, the research community has developed many works in this field. The researches based on permissions and intents of Android apps are prone to false positive, since benign apps also need to require sensitive permissions, which makes them

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

be misclassified as malware easily. As is proved experimentally in the literature [3], the ways based on frequency of API calls such as DroidAPIMiner [4] are unable to establish connections between the API calls to get the high-level behavioral semantics of applications, which leads to poor performance in detecting novel malicious. MaMaDroid [3] is scalable enough to the malware's evolution by exploiting the sequence of API calls to characterize the behavioral patterns of the apps. However, it has a low system efficiency, since the premise of this method is to build the CG of the entire software. The experiment in [3] shows that MaMaDroid takes around 30s for per app's analysis on a desktop equipped with a 40-core 2.30GHz CPU and 128GB of RAM, which makes the system difficult to be applied for lightweight industry scenario.

In this paper, we propose a novel Android malware detection method based on method-level correlation relationship of application's abstracted API calls. As is well-known to us, the behaviors of an app is determined by app's source code through user-defined methods, and each of methods implements specific operations by invoking API calls. The method defined in the app's source code is the basic unit of the app's behavioral semantics. Our intuition is that malware tend to use some specific API calls' combinations in a method to accomplish some maliciously inclined operations different from the benign. For example, if malware wants to steal the privacy, it may collect all the sensitive information in a method with some API calls (e.g., Landroid/telephony/TelephonyManager;->getDeviceId(); Landroid/content/pm/PackageManager;->getPackageInfo(); etc.). The benign apps tend to get the privacy information separately only when they need it, so the API calls associated with obtaining private information generally do not appear at the same time in one method in benign samples.

To analyze the differences between the combinations of API calls in the method of malicious and benign apps is the key to establish the detection system. Therefore, we introduce association rule analysis technology to characterize the API calls' relationships in the same method and capture app's behavioral semantic information. In addition, considering the excessive number of Android API calls and frequent API changes in Android framework, we adopt the method of abstracting API calls to represent app's behaviors instead of directly using specific API calls. For example, we can abstract the API calls related to network(e.g, Landroid/net/ConnectivityManager;getNetworkInfo) to the API call's attribution (e.g., android/net) without abstracting away the behaviors of an app. Since, if an app invokes an API call with attribution ''android/net'', it means that the app will do network-related operations whatever concrete API calls the app invokes. In our systems, API calls are abstracted to their attributions, abstraction granularity can be determined depending on the need of the situation.

The results of our empirical evaluation show that our system has competitive performance in classification accuracy and detection efficiency at task of Android malware detection. At dataset Drebin [5](benign 5.945K and malware 5.56K samples) and dataset AMD [6](benign 20.519K and malware 20.843K samples), our system has achieved 96% and 98% detection accuracies. In order to prove our model is robust to changes in Android malware samples, we collect a mix of older and newer apps, from 2010 to 2017(5.9k benign and 20.520k samples) to make comparative experiments with state-of-the-art(SOTA) work called MaMaDroid. As the experimental results show, our system achieves higher accuracy while improving detection efficiency by 15 times(2.9s vs 45.7s per sample) in the same environment.

In short, our work has the following major contributions:

1. We propose a novel feature representation that incorporates behavioral semantics for Android applications. To the best of our knowledge, it is the first time to use association rule of abstracted API calls to characterize the Android behavioral features. Compared to using frequency of API calls as features, it can capture the semantic information of app's behaviors so that it is more resilient against the Android malware's evolution. Compared to the CG-based ways, our feature construction method is more efficient, since without constructing call graphs.

2. We develop a practical and efficient Android malware detection system. Based on the feature representation of Android applications, we combine machine learning technology to develop a practical system. In addition, we present a comprehensive experimental study based on Drebin and AMD database and comparative experiments with state-of-art MaMaDroid, which fully demonstrates the effectiveness and efficiency of our developed system.

The remainder of this paper is organized as follows: SectionII discusses the related work in detecting Android malware. In the SectionIII, we first present the framework of our system and then introduce our method in detail. SectionsIV discusses the result of experiments on two public data sources to prove the detection performance, by also presenting the result and discussion about the comparison experiments with MaMaDroid. In the SectionV, we discuss some potential limitations of our approach and the future work of our research. Finally, SectionVI concludes our work.

II. RELATED WORK

In recent years, there have been ample works in this Android malware detection field. We can roughly divide these works into two categories: dynamic analysis and static analysis. There are many representative works in dynamic malware analysis. Crowdroid [7] monitors application system calls and gets a log file as feature set, and then uses clustering algorithms for malware analysis and detection. DroidDolphin [8] extracts the features of the API calls' sequences collected from applications during the runtime by n-gram to complete the classification. MADAM [9], a host-based malware detection system, builds detection system by monitoring features belonging to different Android levels including user, application and kernel. Dynamic analysis needs to execute a program and observe the results [10], which could fight

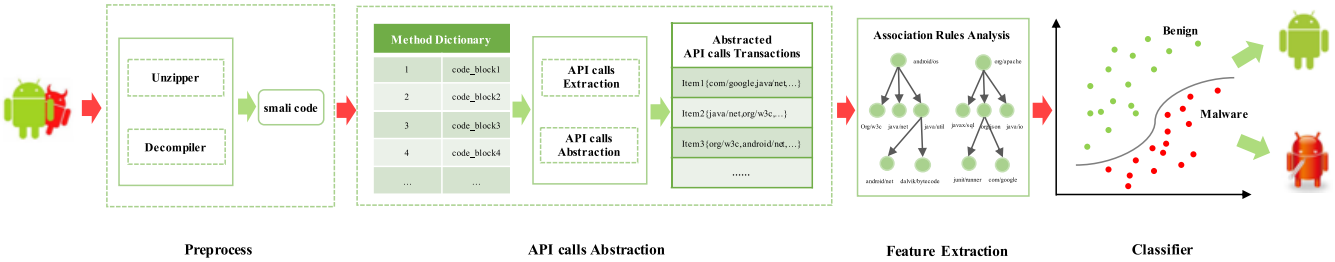


FIGURE 1. The framework of the detection system.

against confusion and dynamically loaded code, but provides limited code coverage and spends more time and computation resources.

Compared to dynamic analysis, static analysis provides more efficient and accurate analysis results, but is hard to cope with confusion and dynamically loaded code. DroidAPIMiner [4] relies on the top-169 API calls that are more frequent in the malware than in the benign sets to finish the classification task. DroidMat [11] uses clustering algorithm to process static features such as API calls, permissions and intents, and then classifies the clustering results through KNN. Drebin [5] extracts numerous features include API calls, network addresses, sensitive system permissions to accomplish classification by using SVM. SiGPID [12], an efficient detection system based on permission usage analysis to cope with the rapid increase in the number of Android malware. Kumar *et al.* [13] proposed a novel method to distinguish between malware and benign applications based on association rule for permission mining. HinDroid [14] propose a novel feature extraction method that uses meta-path to characterize the semantic relatedness of apps and API calls, and then combine the SVM algorithm to detect malware. What's more, Opcode is also a commonly used analysis material in static analysis. The methods proposed in paper [15] and [16] both use n-grams of Opcode as feature extraction to represent the characteristics of applications. Zhang *et al.* [17], propose a weighted probability graph of Dalvik Opcode and extract topology features as the representation of applications, and then search the similarities with these features of programs to detect Android malware.

Recently, the CG-based methods have been proposed, since the methods based on permissions, sensitive API calls, Opcode, etc. lack behavioral relationship information. DroidSIFT [16] extracts a weighted contextual API dependency graph as program semantics to construct feature sets for malware classification. We have already introduced, and compared against, MaMaDroid [3] ,this detection tools extract the app calls and use Markov chains to model the behaviors of Android apps through the sequences of API calls and get good performance in detecting unknown samples at sizable amount of memory and unbearable time cost. Gao *et al.* [18], extract topological signature for Android apps by capturing the invocator-invocatee relationship at local neighborhoods in the function call graphs, and then use SVM to classify malware. Ma *et al.* [19], build multi-level features to classify samples based on the API information by constructing control flow graph of Android applications.

Overall, the detection tools such as Drebin and DroidAPIMINER based on traditional static structural features cannot deal with the evolution of malware because of lack of behavioral semantic analysis. CG-based methods are good at behavioral semantic analysis but have limitations in time efficiency. Our system uses association rule analysis techniques to extract the co-occurrence relationship between abstracted API calls in each Android application method and form behavioral semantics to describe Android application's behaviors, which can effectively detect evolving malware with good time efficiency.

III. PROPOSED METHOD

A. THE FRAMEWORK OF OUR METHOD

In this Section, we introduce the framework of our system. Figure1 shows the architecture of our developed malware detection system. To detect the malware, we first unzip and decompile the Android Application Package (APK) file to get smali code, and then transform the smali code into separate method blocks. By parsing method blocks of smali code, all the API calls invoked in method are extracted. Next, through abstracting the API calls, we transform app's source code into a set of abstracted API calls transactions. Finally, the vector representation of the Android application is obtained by using association rule analysis techniques, and the malware classification is completed by the Random Forest [20] algorithm.

Our approach follows a classical data mining rationale. The framework of our system could be the following four major components.

1) PREPROCESS

In this section, our system uses the tools called Andguard [21] to unzip and decompile Android APK files to get smali code.

2) API CALLS ABSTRACTION

After getting the smali code of APK, we extract all the smali code of each methods and generate a method dictionary for every APK file, and then replace the method code with the API calls that appear in the method. Rather than analyze API

calls itself, the system here focuses on the abstracted API calls by transforming the API calls to their attributions to represent the app's behaviors. After the abstraction operation, abstracted API calls transactions of each sample are generated for further analysis.

3) FEATURE EXTRACTION

The vector representation of sample will be obtained by associated behavioral analysis in this section. We present a methodology known as association rule analysis to discover interesting relationships hidden in data sets including malware and benign. The uncovered relationships between app's behaviors are represented in the form of the confidence value between two abstraction API calls such as operation: ''java/io'' to operation: ''android/net''. All confidence values between two abstracted API calls constitute the behavioral vector representation of the app to be analyzed.

4) CLASSIFIER

Given the vector representation of all the malware and benign in datasets by association analysis, we use machine learning algorithm to construct the classifier. Each newly collected unknown Android app could be labeled either benign or malicious by the trained classifier.

B. PREPROCESS

Since APK files cannot be analyzed directly, some pre-processing operations are required before feature extraction. Unlike the application for desktop based Portable Executable(PE) files, Android app also called as APK file is essentially an zip file, and we can open it with unzipping tools such as WinRar [22]. After decompressing the APK file, we can get the following files: AndroidManifest.xml, META-INF, res, lib, assets, clssses.dex, resources.arsc. ''classes.dex'' file generated after compiling the code written for Android and could be interpreted by the DalvikVM [23] is the main concern of our system. In order to get the Android app's behavioral data, we need to convert the dex file to analyzable format. Smalli code can be decompiled directly from APK files, and contains all the information we need, thus, it becomes the target format as shown in Figure2. In our system, we use Andguard [21] to unzip and decompile the APK files, and get smali code of each Android APK sample.

C. API CALLS ABSTRACTION

In this section, we focus on the motivations for API calls abstraction and the detailed abstraction steps. API calls abstraction is based on two considerations: reduce the number of API calls for raising efficiency and enhance the robustness to cope with the changes in Android framework. Android apps get access to operating system functionality and system resources by using API calls. Therefore, we can use them to represent the app's behaviors to detect malware. If we directly select API calls as the analysis object, it is inevitable to face the problem of combination explosion, since the total number of API calls is too large. API calls abstraction could reduce

```
.method private fillPostData()V
    .locals 5
    .prologue
    const-string v2, "phone"
    invoke-virtual {p0, v2}, Lcom/satismangrooup/stlstart/Bragushterra;->getSystemService(Ljava/lang/String;)Ljava/lang/Object;
    move-result-object v1
    check-cast v1, Landroid/telephony/TelephonyManager;
    .local v1, "telephonyManager":Landroid/telephony/TelephonyManager;
    invoke-virtual {v1}, Landroid/telephony/TelephonyManager;->getDeviceId()Ljava/lang/String;
    move-result-object v2
    iput-object v2, p0, Lcom/satismangrooup/stlstart/Bragushterra;->imei:Ljava/lang/String;
    invoke-virtual {v1}, Landroid/telephony/TelephonyManager;->getNetworkOperator()Ljava/lang/String;
    move-result-object v2
    iput-object v2, p0, Lcom/satismangrooup/stlstart/Bragushterra;->operatorName:Ljava/lang/String;
    :try_start_0
    invoke-virtual {p0}, Lcom/satismangrooup/stlstart/Bragushterra;->getPackageManager()Landroid/content/pm/PackageManager;
    move-result-object v2
    invoke-virtual {p0}, Lcom/satismangrooup/stlstart/Bragushterra;->getPackageName()Ljava/lang/String;
    move-result-object v3
    const/4 v4, 0x0
    invoke-virtual {v2, v3, v4}, Landroid/content/pm/PackageManager;-
>getPackageInfo(Ljava/lang/String;I)Landroid/content/pm/PackageInfo;
    ......
.end method
```

FIGURE 2. An example of smali code from a malicious app(com.satismangrooup.stlstart.Bragushterra).

the number of API calls and enable comprehensive behavioral analysis without losing major information, it becomes a necessary pre-operational step for associated behavioral analysis. In addition, the use of abstract API calls instead of specific API calls can solve the problem of quick update of Android system API calls to a certain extent, which helps to better detect evolving malware.

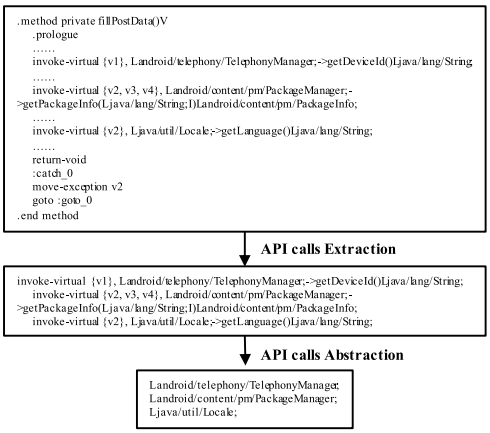


FIGURE 3. An example of API calls abstraction for a single method.

Before abstraction operation, we extract all the user-defined methods by scanning the smali code of each APK file to construct method dictionary. And then, each method is processed separately. An example with detailed process is shown in Figure3. At first, we get the whole smali code of method. And then we pick out the instructions (invoke-virtual, invoke-direct, invoke-static, invoke-super) related to calling API calls to extract API calls in the method and delete other smali instructions that we do not care about. Finally, we have API calls abstraction operations by preserving some attributes(referring to the rhetorical words separated by separators in the API calls' package name such as ''android'',''net'', etc.) of API calls. In the example shown in Figure3, the entire package name is preserved as an abstract result, but in the case, the abstraction granularity can be flexible. Next, detailed abstract methods as following will be discussed.

By analyzing the real world app file, the system API calls can be divided into three different types: Android system API calls, user-defined API calls in the application, and others that have been confused. The latter two are abstracted into two types: self-defined and confused. For the system API calls, we will further elaborate the abstraction steps. According to the latest Android platform [24], there are 443 packages in the Android API-Level 28, and if we directly analyze the relationship between the two packages, the feature dimension of an APK file is up to $A^2_{445} = 197580$, which is unacceptable to the original intention of lightweight detection system. Thus, further coarsening of the abstract granularity is required.

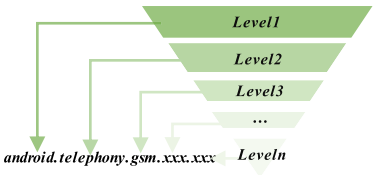


FIGURE 4. Hierarchical level schematic diagram of API call's attribution. The attribution here refers to the rhetorical words separated by separators in the API calls' package name such as "android", "telephone", etc.

The package name of the Android system API calls is split by dot notation and listed in the hierarchical level as shown Figure4. The meaning of each level is refined layer by layer. For example, "android.telephony.gsm" can be split into the following three levels: level-1:android, level-2: telephony, level-3: gsm. Level-1 means that the API call comes from Android family, and level-2 means that the API call has operation about telephone, and level-3 means providing functional service for utilizing GSM-specific telephony features. Therefore, we could make a trade-off between the precious behavioral meaning and the number of feature dimensions by removing high level attribution just like Figure5. Our system retains only the first two levels of attributions, abstracts the 443 packages in API Level-19 into 73 plus self-defined and confused, adding up to a total of 75 abstracted operations.

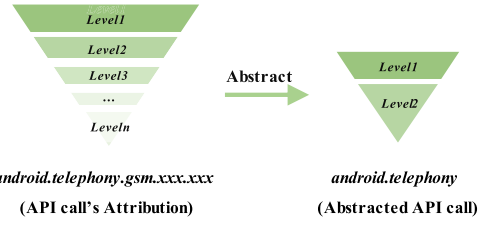


FIGURE 5. An example of API call abstraction operation.

After the above series of processing steps, we could get abstracted API calls items of each method and then generate abstracted API calls transactions for each APK file for further behavioral association analysis.

D. FEATURE EXTRACTION

In this section, we introduce feature extraction based on associated analysis in detail. The method defined in the app's source code is the basic unit of the app's behavioral semantics. Malware and benign usually show different behavioral patterns in the construction of function method, which manifest in different API combinations owned by different function methods. To discover these potential patterns of behaviors, we use the method-level associated analysis to construct the characteristics. Before elaborating on the detailed feature extraction steps, some prerequisite knowledge from [25], [26] will be introduced firstly.

1) ITEMSET

Each type of abstract API call can be called an item, such as "java.io", "android.net". The collection of all abstract API types is called an itemset. Let $I = [I_1, I_2, I_3 \cdots I_d]$ (In our system, $d = 75$) be the set of all items in abstracted API calls. A sample contains an indefinite number of methods, and we call the collection of abstract API calls contained in each method a transaction. At the API calls abstraction stage, we have got a set of abstracted API calls transactions of each APK file. If an APK file contains k methods, we can use $T = [t_1, t_2, t_3 \cdots t_k]$ to represent the APK file. Transaction t_k contains a subset of items chosen from itemset I. In association analysis, a collection of zero or more items is termed an itemset. If an itemset contains k items, it is called a $k-$itemset. In our system, if a method contains i abstracted API calls, it is called $i-$itemset.

2) SUPPORT COUNT

Transaction width is defined as the number of items presented in a transaction. A transaction such as t_k is said to contain an itemset X if X is a subset of t_k. In associated analysis, an important property of an itemset is its support count, which refers to the number of transactions that contain a particular itemset in the whole transactions. Mathematically, the support count: $\sigma(X)$, for an itemset X can be stated as follows:

$$\sigma(X) = |\{t_i | X \in t_i, t_i \in T\}| \tag{1}$$

where the symbol "$|.|$" denotes the number of elements in a set.

3) ASSOCIATION RULE

An association rule is an implication expression of the form X to Y, where X and Y are disjoint itemsets, and X or Y is the subset of I. In our system, because the confidence of the rules between the two API calls has provided enough information to express their behavioral semantics, and in order to maximize the detection efficiency at the same time, we only care about rules between two items such as {java.io} to {java.net}, {android.net} to {org.xml}.

4) CONFIDENCE

The strength of an association rule can be measured in terms of its confidence. The formal definition of the support can be

Algorithm 1: Feature Extraction Based on Associated Analysis Process

Input: S : sample's smali code
$I = [I_1, I_2, I_3 \cdots I_d]$: set of all items in abstracted API calls
Output: $confidence =< Rule, confidence >$
From S, get method dictionary $D =< N_i, C_i >$,
the number of D is K;
$confidence = \{\}$;
$T = \{\}$;
for $i = 1; i \leq K; i = i + 1$ **do**
  C_i API call Extract $\Rightarrow C_i'$;
  C_i' API call Abstraction $\Rightarrow t_i = [p_1, p_2, \cdots p_d]$;
  $T[N_i] = t_i$;
for $r_1 \in I[I_1, I_2, I_3 \cdots I_d]$ **do**
  $\sigma(r_1) = |\{t_i | r_1 \in t_i, t_i \in T\}|$;
  for $r_2 \in I[I_1, I_2, I_3 \cdots I_d]$ **do**
    if $r_1 ! = r_2$ **then**
      $\sigma(r_1 \cup r_2) = |\{t_i | r_1 \cup r_2 \in t_i, t_i \in T\}|$;
      $c(r_1 -> r_2) = \frac{\sigma(r_1 \cup r_2)}{\sigma(r_1)}$;
      $confidence[r1 -> r2] = c(r_1 -> r_2)$;
return $confidence =< Rule, confidence >$;

seen in Eq.(1). It determines how frequently items in Y appear in transactions that contain X and measures the reliability of the inference made by a rule(X->Y), which is the key to represent the characteristics of Android app's behaviors in our system.

$$Confidence, c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)} \tag{2}$$

5) FEATURE EXTRACTION BASED ON ASSOCIATED ANALYSIS

The co-occurring relationship between API calls in same method is able to be well described by the confidence of association rules between the abstracted API calls. Therefore, in our system, we combine all 75 abstracted API calls and generate $A_{75}^2 = 5550$ association rules, and then calculate the confidence scores of them to get the confidence matrix as the feature representation of Android applications just like in Figure6. Complete feature extraction process can be seen in Algorithm1. Compared to the systems using pure frequency of API call as feature, we argue that our system is able to build high-level semantics and discover behavioral patterns through association analysis.

E. CLASSIFIER

Classification is the last step of our system. In this stage, our system uses the classifiers related to machine learning to label apps as either malicious or benign. During our experiments, we extract the features described as SectionIII-D from the datasets and train the model with different classification algorithms including the Nearest Neighbor [27], Support Vector Machines [28] and Random Forest [20]. And the results will be discussed in detail in the SectionIV-A.

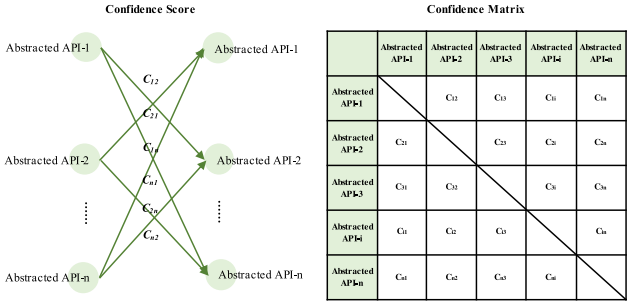


FIGURE 6. Feature extraction based on association rule between the abstracted API calls(we use C_{ij} to represent confidence of rule(Abstracted API_i -> Abstracted API_j).

IV. EXPERIMENTS AND ANALYSIS

In this section, we conduct two experiments to prove the effectiveness and efficiency of our proposed method. In the first set of experiments, we evaluate the detection performance of our proposed Method on the two public malware datasets(AMD and Drebin). And in the second set of experiment, we compare our system with SOTA system called MaMaDroid in terms of time efficiency and evolving malware detection performance under the same conditions using a mix of older and newer apps ranging from 2010 to 2017 downloads from Androzoo [29]. The following experiments are performed on a Window7 operation system, powered by Intel(R) Core(TM) i7-6700 CPU@3.40GHz and 8 GB of RAM.

TABLE 1. Performance indices of android malware detection.

Indices	Description
Precious	$\frac{TP}{TP+FP}$
Recall	$\frac{TP}{TP+FN}$
F-measure	$\frac{2 * Precious * Recall}{Precious + Recall}$
Acc	$\frac{TP+TN}{TP+TN+FP+FN}$

To test our proposed method, the metrics shown in Table1 are used in the following experiments. Among the equations, FP is the number of apps that are mistakenly classified as malicious; FN is the number of apps that are mistakenly classified as benign; TP is the number of apps that are correctly classified as malicious; TN is the number of apps that are correctly classified as benign.

A. DETECTION PERFORMANCE EVALUATION

1) EXPERIMENT PURPOSE AND PROCEDURE

In order to evaluate the accuracy of our system in the detection of unknown Android applications with the same age distribution, we conduct a full test experiment on the two well-known

TABLE 2. Overview of datasets used for our system evaluation experiments.

Dataset	Malware		Benign	
	Number	Description	Number	Description
#Drebin	5,560	From Drebin[5] (2010 to 2012)	5,945	From Androzoo[28] (2010 to 2012)
#AMD	20,843	From AMD[6] (2010 to 2016)	20,519	From Androzoo[28] (2010 to 2016)

public datasets: Drebin dataset [5] and AMD dataset [6]. In addition, in order to choose the appropriate classification algorithm, we perform corresponding experiments on the three representative algorithms including SVM, KNN and Random Forest.

In our experiment, we perform 5-fold cross validations on the combined dataset composed of malicious and benign and choose the Precious, Recall, F-measure and Acc as the evaluation index.

In order to show the different behavioral patterns between malware and benign, we extract the top-10 most important association rules of abstracted API calls from the trained model, and list the differences by calculating the average confidence value of the rules in malicious and benign applications. Meanwhile, a brief analysis of the key abstracted API calls involved is presented in the end.

2) DATA SOURCE

To make the experiment more convincing, we collect 2 well-known datasets, the detail information can be seen in Table2. The first dataset is Drebin [5] which contains 5,560 applications from 179 different malware families. The samples have been collected in the period of August 2010 to October 2012. To maintain the same chronological distribution as the malware, we collect 5,945 benign samples from Androzoo [29] ranging from 2010 to 2012 and compose a test dataset. AMD [6] is another dataset we collect, and it contains 24,553 samples, categorized in 135 varieties among 71 malware families ranging from 2010 to 2016. However, because of the network mistake, we only successfully downloaded 20,843 at the end. Similarly, in order to stay consistent with malicious apps and form a complete dataset, we collect 20519 benign from Androzoo [29] ranging from 2010 to 2016.

In the actual process, not all sample's API calls can be extracted due to the errors during unpacking. In our experiment, the successful ratio of #Drebin is 11,332(malware: 5,448 benign: 5,883) out of 11,505(malware: 5,560 benign: 5,945), and #AMD is 40533(malware: 20,583, benign: 19,950) out of 41,362(malware: 20,843 benign: 20,519).

3) RESULTS AND ANALYSIS

We test the performance of different algorithms with the behavioral relationship features. Table4 shows different detection results from our system with #AMD dataset, and Table3 is the detection result with #Drebin dataset. From the result, we can know that the performances of the three algorithms are consistent on the two datasets. KNN's performance is the worst overall, and Random Forest outperforms KNN and SVM in the both of datasets. By using a single feature, the highest detection accuracy of our experiment (Acc) is 98%, which fully demonstrates the effective detection performance of our method.

TABLE 3. Detection performance results with different algorithms on #Drebin dataset.

Algorithm	Precious	Recall	F-measure	Acc
KNN	0.90	**0.96**	0.93	0.92
RF	**0.97**	0.95	**0.96**	**0.96**
SVM	0.94	0.94	0.94	0.94

TABLE 4. Detection performance results with different algorithms on #AMD dataset.

Algorithm	Precious	Recall	F-measure	Acc
KNN	0.95	0.97	0.96	0.96
RF	**0.99**	**0.98**	**0.98**	**0.98**
SVM	0.97	0.97	0.97	0.97

In order to better reveal the classification's results, we list the top-10 most important association rules of abstracted API calls for malware classification. The scores of the association rules are obtained from the Gini coefficient in Random Forest [30]. The top-10 important features and their average confidence scores between malware and benign can be seen in Figure7. Among the rules, we find some interesting phenomena. At first, malware tend to have higher confidence scores than benign among the top-10 association rules. The combinations of abstracted API calls like the ten rules may be more dangerous. And then, seven of the ten rules are related to ''android/telephone''. We can speculate that the API calls associated with telephone are critical to malware analysis. Meanwhile, we find no rules related to ''org*'' in the top 10 rules, which portends that the API calls related to ''org'' such as ''org/w3c.dom'', ''org/json'', etc. may contribute less to malware classification.

B. OUR SYSTEM VS MAMADROID

1) EXPERIMENT PURPOSE AND PROCEDURE

There are reasons to believe that our systems are capable of detecting evolving malware, since our system could capture

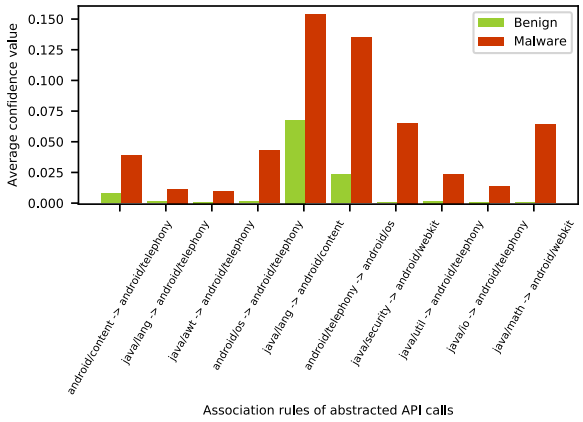


FIGURE 7. Comparison of the top-10 most important association rules of abstracted API calls in malware and benign applications. *x*−axis lists the top-10 most important rules of malware classification. *y*−axis lists the average confidence values of association rules of malware and benign. Malware tend to have higher confidence scores than benign among the top-10 association rules.

semantic information of app behavioral. In addition, because of not extracting the function call graphs, theoretically our method should have better detection efficiency than the CG-based methods. Therefore, in order to prove the ability in dealing with the evolution of malware and efficiency of our method, we conduct a comparative experiment with SOTA system that takes evolution into account called MaMaDroid on the dataset span across multiple years (2010-2017). During our experiments, we use the same experimental method as MaMaDroid and follow their modified source code.

At the first part of the experiments, a comparative experiment on evolving malware detection performance is carried out. To be consistent with MaMaDroid, we've reduced the API abstraction granularity. In the experiment, we use the same abstraction way as MaMaDroid in family mode. The abstracted system API calls include nine possible families, i.e., com.google, android, java, javax, org.xml, org.apache, junit, org.json, and org.w3c.dom. Similarly, we also use the same experimental method as MaMaDroid. The most significant trait between this experimental method and traditional methods is using previous samples for training, and then using new samples for testing and vice versa. The setting is more appropriate to evaluate our system's robustness to the evolution of malware. The malware samples in the experiment are from 2010 to 2017 , each of them is marked with a year number, and the benign datasets consist of two sets of samples which is denoted as oldbenign (from year of 2014) and newbenign (from year of 2017). Because the classification is missing from the code provided by the authors, during reproducing the MaMaDroid system, Random Forests that impress the final effect by using 51 trees with maximum depth 8 is used as classification algorithm as MaMaDroid shows in paper. We use 3/4 of the dataset for training and 1/4 for testing. The accuracy evaluation metrics in the process of experiment all adopt F-measure.

The second part of the experiments are to test the detection efficiency. As is known to us, the time cost of detection system mainly includes two parts: the time of feature extraction and the time of classification. The characteristic dimensions of the two systems are almost the same, and classification time based on Random Forest is less than 0.01s, which accounts for less than 1% of the whole detection process time in the actual detection task. Therefore, only feature extraction time is considered in the analysis of time efficiency. In the experiment, we firstly select 6 groups of samples distribution from 2010 to 2017, and each group has 100 samples. Among them, samples from 2010-2012 are combined into a single group, and samples for each year from 2013-2017 are the other 5 groups respectively. And then, the feature extraction time of each APK file is measured in both systems separately for further statistical analysis. The maximum, minimum and average analysis time of each group of samples are recorded in detail. Through these data, the performance of the two systems in term of time efficiency is further analyzed.

TABLE 5. Overview of the datasets used for comparative experiments with MaMaDroid.

Category	Name	Date range	#sample	#our work	# [3]
Benign	oldbenign	2014	2,992	2,923	2,942
	newbenign	2017	3,000	2,915	2,907
		Total Benign	**5,992**	**5,838**	**5,948**
Malware	Drebin	2010-2012	5,560	5,439	5,518
	2013	2013	3,000	2,997	2,873
	2014	2014	3,998	2,987	2,510
	2015	2015	2,962	2,886	2,809
	2016	2016	2,960	2,923	2,740
	2017	2017	3,040	2,916	2,830
		Total Malware:	**20,520**	**20,148**	**19,280**

2) DATA SOURCE

An exhaustive list of dataset is shown in Table5. Our datasets are composed of malware and benign samples. As for benign datasets, they consist of oldbenign and newbenign. All of them come from the Androzoo [29] ,the difference is that oldbenign sample's dex_date's value is 2012, and the dex_date of newbenign samples is 2017. The datasets of Android malware contain the chronological distribution from 2010 to 2017. It should be noted that we take Drebin data as a separate year database like the work [3] do. The Malware samples from 2013 to 2017 are also all from Androzoo [29], We use dex_date's value to distinguish the years and select those samples with more than 10 detection engines marked as malicious. The exact number of all samples can be obtained from the "#sample" column in the Table5. During samples' preprocesses, there are samples that cannot successfully be extracted features in both systems. The concrete information about the number of successfully extracted samples can be obtained from the column called "#our work" and the column named "#[3]" in the Table5.

TABLE 6. Detection performance on F-measure of our work vs MaMaDroid [3]. As shown in the table, among the 36 test results, 25 of our work are better than or equal to the MaMaDroid.

	Testing Sets											
	Our work	[3]	Our work	[3]	Our work	[3]	Our work	[3]	Our work	[3]	Our work	[3]
Training Set	Drebin&oldbenign		2013&oldbenign		2014&oldbenign		2015&oldbenign		2016&oldbenign		2017&oldbenign	
Drebin&oldbenign	**0.97**	0.93	0.73	**0.84**	0.60	**0.80**	0.59	**0.82**	0.55	**0.78**	0.44	**0.51**
2013&oldbenign	**0.78**	0.67	**0.93**	0.90	0.79	**0.87**	0.67	**0.73**	0.68	**0.70**	**0.29**	0.20
2014&oldbenign	**0.90**	0.45	**0.90**	0.81	0.91	**0.92**	**0.89**	0.76	**0.86**	0.72	**0.61**	0.22
	Drebin&newbenign		2013&newbenign		2014&newbenign		2015&newbenign		2016&newbenign		2017&newbenign	
2015&newbenign	**0.97**	0.94	**0.97**	**0.97**	**0.96**	0.93	**0.94**	0.93	0.90	**0.91**	**0.73**	0.57
2016&newbenign	**0.93**	0.83	**0.96**	0.92	**0.96**	0.89	**0.95**	0.91	**0.94**	**0.94**	**0.89**	0.84
2017&newbenign	**0.97**	0.61	**0.93**	0.80	**0.88**	0.85	**0.85**	**0.85**	0.90	**0.92**	**0.93**	0.92

3) RESULTS AND ANALYSIS

a: DETECTION ACCURACY

We divide the datasets into different testing sets and training sets as experiment settings in work [3] and get 36 experiment results. As shown in Table6, we can see that both detection systems have good performance in the detection of unknown malware when evolution of Android is taken into account. In our system, we obtain 0.94 F-measure on average when we test same year samples with training, and obtain 0.84 and 0.72 F-measure one and two years after training. MaMaDroid's performance is comparable to ours on the average F-Measure, which is 0.92 at same year, 0.84 after one year training, 0.70 after two year training. When using newer samples for training and older for testing, 14 out of the 15 test results show that our system performs better. From a global perspective, among the 36 results, 25 of our work are better than or equal to the MaMaDroid, and 13 of the MaMaDroid are better than or equal to our work. Therefore, our work can achieve even better results than MaMaDroid in detecting evolving malware.

b: RUNTIME PERFORMANCE

In our experiment, the exact time that the selected samples spend on getting the features is measured in both systems. The statistics in Table7 is the raw data we measured. During processing the raw data, we find that some APK files spend several hours in the feature extraction process (the maximum time taken by a single APK file is around 3.5 Hours). In order to make the results of the analysis more reasonable, we remove all files with the analysis time of more than 20 minutes and get the final statistics shown in rows that marked with # in Table7. At the same time, in order to better show the difference in detection efficiency between the two systems, the maximum and minimum values and the average values of the sample analysis time of each year in our table are displayed by the method of line graph, and the specific information can be seen in Figure8 and Figure9.

As shown in the Figure8 and Figure9, two conclusions can be drawn as following.

1. Our system performs far better than MaMaDroid in terms of time efficiency. Parsing the complex calling relationships between each method in the application is an inevitable operation of MaMaDroid, and our system focuses on the relationship between API calls in the individual method. Hence, our method has a lower complexity in principle. As seen in Figure8, The average value, maximum value and minimum value of time consumption of each sample group in our system are significantly less than MaMaDroid.

TABLE 7. Runtime performance of our work vs MaMaDroid [3].

Group	Min(second)	Max(second)	Mean(second)	Mean ratio
	[3] our work	[3] our work	[3] our work	[3]:our work
2010-2012	1.56 0.01	165.69 6.42	15.7 0.72	21.8
*2010-2012	1.56 0.01	165.69 6.42	15.7 0.72	21.8
2013	1.56 0.01	462.22 5.43	33.69 1.61	21.0
*2013	1.56 0.01	462.22 5.43	33.69 1.61	21.0
2014	4.53 0.07	546.19 12.45	33.15 2.00	16.6
*2014	4.53 0.07	546.19 12.45	33.15 2.00	16.6
2015	0.47 0.05	**7162.13** 12.45	**160.32** 2.68	**59.7**
*2015	0.47 0.05	**1341.46** 12.45	**40.3** 2.68	**15.0**
2016	3.61 0.04	**12770.95** 14.77	**374.01** 3.97	**94.2**
*2016	3.61 0.04	**1459.54** 14.77	**87.21** 3.97	**22.0**
2017	0.94 0.02	**4092.77** 15.37	**103.98** 6.56	**15.7**
*2017	0.94 0.02	**515.75** 15.37	**64.21** 6.56	**9.8**
		Total Mean:	**120.14 2.90**	**41.4**
		***Total Mean:**	**45.70 2.90**	**15.6**

note: the groups that marked with * mean the experiment results after removing files with the analysis time of more than 20 minutes

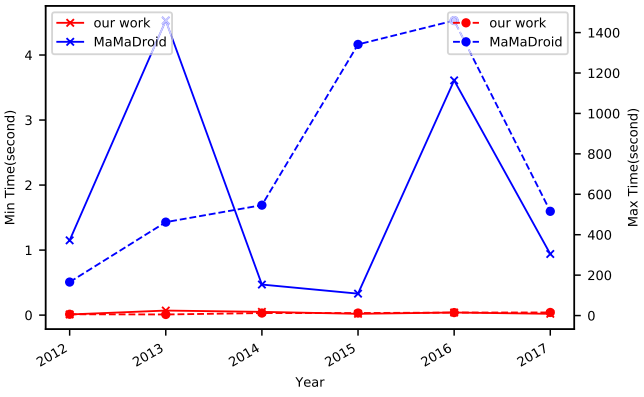


FIGURE 8. Results on the maximum and the minimum analysis time in each group of our work vs MaMaDroid.

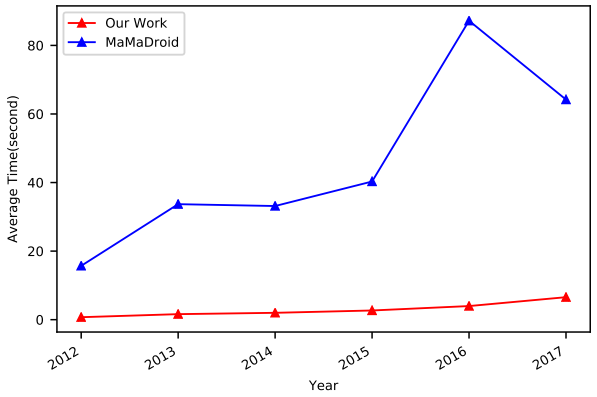


FIGURE 9. Results on the average analysis time in each group of our work vs MaMaDroid.

In the 2010-2012 sample group, the time efficiency gap between the two systems is the largest, and MaMaDroid consumes 21.8 times as much time as our system on average(our work: 0.72s per sample vs MaMaDroid: 15.7s per sample). The smallest time efficiency gap is in the 2017 samples group, but the average MaMaDroid still consumes 9.8 times as much time as our system(our work: 6.56s per sample vs MaMaDroid: 64.21s per sample). Among all the samples of 6 groups, the average elapsed time for MaMaDroid sample analysis is 45.7 seconds per sample, which is 16.5 times that of our system (2.9s per sample).

2. The stability of our system is much better than MaMaDroid. The time consumption of our method is linearly related to the number of methods in the application's source code. Hence, the average analysis time(as shown in Figure9) of each group increases steadily with samples age moving forward(app's complexity increases over time [3]), and among all samples, the longest analysis time of an app does not exceed 15.37s. However, when MaMadroid based Soot [31] performs CG extraction, the time and memory consumption of the system grows exponentially as the logical complexity of the application's code increases, which brings a lot of instability to the system. As the Figure8 shows, the Max analysis time of each group shows great fluctuations in MaMaDroid. Some extreme samples even need more than 3 hours of analysis time, and these instability performances are intolerable in the actual industrial scenarios.

V. DISCUSSIONS

In this section, we first discuss some potential limitations of our approach. Furthermore, we discuss the future work of our research. Due to the inherent characteristics of static analysis, our system is powerless against the technology such as Bytecode Encryption, Dynamic Loading and Native Code, because these techniques make the system unable to get system API calls to analyze malicious behaviors of the application. Also, considering the principles of our approach, the attacker may can disrupt malicious behavioral relationship analysis by adding a large amount of extraneous code containing normal system API calls combinations to application in the case of knowing the structure of the detection system.

As future work, we plan to analyze the samples that are not correctly classified and find out the reasons to provide guidances for subsequent search of finding more appropriate API abstraction granularity, and more effective association rules. In addition, for the defects of static detection, we will consider the integration of dynamic analysis to build the next generation of hybrid detection system.

VI. CONCLUSIONS

To deal with the challenges of efficient identification of evolving Android malware, we propose a novel malware detection method based on method-level correlation relationship of app's abstracted API calls. In our approach, we use API calls abstraction techniques to reduce the number of API calls for raising efficiency and enhance the robustness to cope with the changes in Android framework. Instead of using call graphs, we transform each app's source code into a set of abstracted API calls and then calculate the strength of association rules between every two abstracted API calls transactions to get the confidence matrix that incorporates behavioral semantics as the feature representation of Android applications, which makes feature extraction much more efficient than CG-base ways and sophistic malware more difficult to evade the detection.

When detecting malware with the same age distribution on dataset Drebin [5] and dataset AMD [6], our system can achieve 96% and 98% detection accuracies. What's more, compared with SOTA research work in detecting evolving malware called MaMaDroid [3], our method does not require the time-consuming operation of building function call graphs, hence our approach has obvious advantage in terms of time efficiency and system stability. Sufficient experimental results show that the average analysis time per app file of MaMaDroid is 15.6 times longer than that of our system (Our Work: 2.9s per sample vs MaMaDroid: 45.7s per sample) in the case of ensuring detection accuracy, and our system is more stable than MaMaDroid without the extreme situation that analyzing a single file takes more than 3 hours.

REFERENCES

[1] (2019). *Smartphone Market Share*. [Online]. Available: http://www.michaelshell.org/tex/ieeetran

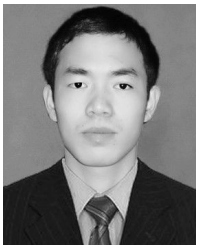
[2] (2018). *12,000 New Samples Per Day*. [Online]. Available: http://blogs.360.cn/post/review_android_malware_of_2018.html?from=timeline

[3] L. Onwuzurike, E. Mariconti, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android malware by building Markov chains of behavioral models (extended version)," *ACM Trans. Privacy Secur.*, vol. 22, no. 2, p. 14, 2019.

[4] Y. Aafer, W. Du, and H. Yin, "DroidAPIminer: Mining API-level features for robust malware detection in Android," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.* Cham, Switzerland: Springer, 2013, pp. 86–103.

[5] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. Annu. Symp. Netw. Distrib. Syst. Secur. (NDSS)*, vol. 14, 2014, pp. 23–26.

[6] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current Android malware," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*. Cham, Switzerland: Springer, 2017, pp. 252–276.
[7] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2011, pp. 15–26.
[8] W.-C. Wu and S.-H. Hung, "DroidDolphin: A dynamic android malware detection framework using big data and machine learning," in *Proc. Conf. Res. Adapt. Convergent Syst.*, 2014, pp. 247–252.
[9] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "MADAM: Effective and efficient behavior-based Android malware detection and prevention," *IEEE Trans. Depend. Sec. Comput.*, vol. 15, no. 1, pp. 83–97, Jan./Feb. 2018.
[10] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of Android malware and Android analysis techniques," *ACM Comput. Surv.*, vol. 49, no. 4, p. 76, 2017.
[11] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android malware detection through manifest and API calls tracing," in *Proc. 7th Asia Joint Conf. Inf. Secur.*, Aug. 2012, pp. 62–69.
[12] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-An, and H. Ye, "Significant permission identification for machine-learning-based Android malware detection," *IEEE Trans. Ind. Informat.*, vol. 14, no. 7, pp. 3216–3225, Jul. 2018.
[13] R. Kumar, X. Zhang, R. U. Khan, and A. Sharif, "Research on data mining of permission-induced risk for Android IoT devices," *Appl. Sci.*, vol. 9, no. 2, p. 277, Jan. 2019.
[14] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, "HinDroid: An intelligent Android malware detection system based on structured heterogeneous information network," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2017, pp. 1507–1515.
[15] A. Shabtai, Y. Fledel, and Y. Elovici, "Automated static code analysis for classifying android applications using machine learning," in *Proc. Int. Conf. Comput. Intell. Secur.*, Dec. 2010, pp. 329–333.
[16] G. Canfora, A. De Lorenzo, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Effectiveness of opcode ngrams for detection of multi family Android malware," in *Proc. 10th Int. Conf. Availability, Rel. Secur.*, Aug. 2015, pp. 333–340.
[17] J. Zhang, Z. Qin, K. Zhang, H. Yin, and J. Zou, "Dalvik opcode graph based Android malware variants detection using global topology features," *IEEE Access*, vol. 6, pp. 51964–51974, 2018.
[18] T. Gao, W. Peng, D. Sisodia, T. K. Saha, F. Li, and M. Al Hasan, "Android malware detection via graphlet sampling," *IEEE Trans. Mobile Comput.*, to be published.
[19] Z. Ma, H. Ge, Y. Liu, M. Zhao, and J. Ma, "A combination method for Android malware detection based on control flow graphs and machine learning algorithms," *IEEE Access*, vol. 7, pp. 21235–21245, 2019.
[20] A. Cutler, D. R. Cutler, and J. R. Stevens, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 157–176, 2004.
[21] *Androguard*. Accessed: 2019. [Online]. Available: https://github.com/androguard
[22] *Winrar*. Accessed: 2019. [Online]. Available: https://www.win-rar.com/
[23] *Dalvik Opcodes*. Accessed: 2019. [Online]. Available: http://pallergabor.uw.hu/androidblog
[24] *Android APIS Reference*. Accessed: 2019. [Online]. Available: http://www.android/doc.com/reference/packages.html
[25] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. ACM SIGMOD Rec.*, 1993, vol. 22, no. 2, pp. 207–216.
[26] P.-N. Tan, *Introduction to Data Mining*. New Delhi, India: Pearson Education, 2018, pp. 328–332.
[27] E. Fix and J. L. Hodges, Jr., "Discriminatory analysis-nonparametric discrimination: Small sample performance," Univ. California, Berkeley, CA, USA, Tech. Rep. ADA800391, 1952. [Online]. Available: https://apps.dtic.mil/docs/citations/ADA800391
[28] T. Joachims, "Text categorization with support vector machines: Learning with many relevant features," in *Proc. Eur. Conf. Mach. Learn.* Berlin, Germany: Springer, 1998, pp. 137–142.
[29] 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.
[30] R. Genuer, J.-M. Poggi, and C. Tuleau-Malot, "Variable selection using random forests," *Pattern Recognit. Lett.*, vol. 31, no. 14, pp. 2225–2236, Oct. 2010.
[31] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot—A Java bytecode optimization framework," in *Proc. 1st CASCON Decade High Impact Papers*. New York, NY, USA: IBM, 2010, pp. 214–224.



HANQING ZHANG received the B.E. degree from Harbin Engineering University, Harbin, China, in 2017. He is currently pursuing the M.E. degree with the Information System and Security and Countermeasures Experimental Center, Beijing Institute of Technology. His current research interests include machine learning, data mining, and mobile security.

SENLIN LUO received the B.E. and M.E. degrees from the College of Electrical and Electronic Engineering, Harbin University of Science and Technology, Harbin, China, in 1992 and 1995, respectively, and the Ph.D. degree from the School of Information and Electronics, Beijing Institute of Technology, Beijing, China, in 1998. He is currently the Deputy Director, the Laboratory Director, and a Professor with the Information System and Security and Countermeasures Experimental Center, Beijing Institute of Technology. His current research interests include machine learning, medical data mining, and information security.

YIFEI ZHANG received the master's degree from the School of Information Engineering, Communication University of China, Beijing, China, in 2015. He is currently pursuing the Ph.D. degree with the Information System and Security and Countermeasures Experimental Center, Beijing Institute of Technology. His current research interests include cyberspace security, operating system security, and virtualization security.

LIMIN PAN received the B.E. and M.E. degrees from the College of Electrical and Electronic Engineering, Harbin University of Science and Technology, Harbin, China. She is currently with the Information System and Security Countermeasures Experimental Center, Beijing Institute of Technology. Her current research interests include machine learning, medical data mining, and information security.