Hindawi
Security and Communication Networks
Volume 2017, Article ID 6451260, 14 pages
https://doi.org/10.1155/2017/6451260

Research Article

Mlifdect: Android Malware Detection Based on Parallel Machine Learning and Information Fusion

Xin Wang,[1] Dafang Zhang,[1] Xin Su,[2,3] and Wenjia Li[4]

[1]*College of Computer Science and Electronics Engineering, Hunan University, Changsha, China*
[2]*Hunan Provincial Key Laboratory of Network Investigational Technology, Hunan Police Academy, Changsha, China*
[3]*Key Laboratory of Network Crime Investigation of Hunan Provincial Colleges, Hunan Police Academy, Changsha, China*
[4]*Department of Computer Sciences, New York Institute of Technology, New York, NY, USA*

Correspondence should be addressed to Dafang Zhang; dfzhang@hnu.edu.cn

Received 23 January 2017; Revised 4 June 2017; Accepted 6 July 2017; Published 28 August 2017

Academic Editor: Jesús Díaz-Verdejo

In recent years, Android malware has continued to grow at an alarming rate. More recent malicious apps' employing highly sophisticated detection avoidance techniques makes the traditional machine learning based malware detection methods far less effective. More specifically, they cannot cope with various types of Android malware and have limitation in detection by utilizing a single classification algorithm. To address this limitation, we propose a novel approach in this paper that leverages parallel machine learning and information fusion techniques for better Android malware detection, which is named *Mlifdect*. To implement this approach, we first extract eight types of features from static analysis on Android apps and build two kinds of feature sets after feature selection. Then, a parallel machine learning detection model is developed for speeding up the process of classification. Finally, we investigate the probability analysis based and Dempster-Shafer theory based information fusion approaches which can effectively obtain the detection results. To validate our method, other state-of-the-art detection works are selected for comparison with real-world Android apps. The experimental results demonstrate that *Mlifdect* is capable of achieving higher detection accuracy as well as a remarkable run-time efficiency compared to the existing malware detection solutions.

1. Introduction

We have witnessed more than 1.4 billion smartphones all over the world in 2015, and five out of six new phones were running Android operating system [1]. Android has become the most popular mobile OS worldwide. Meanwhile, the number of Android applications (apps) grows exponentially, which has significantly benefited the daily lives for mobile users. However, the popularity and openness of Android system have also attracted many malware authors to aim at it. According to Symantec Internet Security Threat Report [2], there were more than three times as many Android apps classified as malware in 2015 than in 2014. The private data of the users, such as IMEI, contacts list, and other user specific data, are the primary target of the Android malware, which has imposed a serious threat for the security and privacy of mobile users. Consequently, there is an urgent need for effective defense mechanisms to protect Android-based mobile devices.

Detection of known Android malware is commonly performed by antivirus tools, such as AVG (https://www.avg.com/ww-en/homepage/) and F-secure (https://www.f-secure.com/en/web/home_global/home), which detect Android malware based on their known features (e.g., signature library). However, they cannot efficiently detect unknown Android malware. Therefore, a large number of research works have been focused on Android malware detection by using various machine learning algorithms. For example, *Drebin* [3] extracted several kinds of features, such as API calls and permissions, and used the support vector machine (SVM) algorithm to detect unknown malware. However, it spent too much time to build the classifier because of the high-dimensional feature vectors. *Fest* [4] focused on selecting useful features from static analysis; it also used

SVM algorithm to classify Android apps. In addition, Du et al. [5]. utilized the SVM algorithm and the Dempster-Shafer theory to fuse results. However, these prior works only used one single classification algorithm, which makes it hard to prove which algorithm is most suitable for Android malware detection. Alternatively, Yerima et al. [6]. collected 179 features including API calls and permissions, and they chose random forest, a kind of ensemble learning algorithms, to achieve high detection accuracy. But the features cannot describe an application accurately and the random forest requires multiple iterations, which consume substantial amount of computational resource. All those phenomena make detection inefficiency.

To address the above limitations of these prior research efforts, we propose a novel detection method based on parallel machine learning and information fusion, named *Mlifdect*. We first apply static analysis to extract multilevel features, such as permissions, API calls, and deployment of components, for characterizing the behavioral pattern of Android applications, and extract more than 65,000 features. Then, we leverage feature selection algorithms to select typical features from the extracted features for better detection performance. Following that, we utilize parallel machine learning to build a classification model which consists of diverse classifiers and execute them in parallel to speed up the process. The outputs of classification are probability values instead of classification results and the output probabilities can complement further research. After combining each output probability value with local credibility, which is localized as a confidence measure of a classifier, Dempster-Shafer theory and probability analysis are used to integrate them.

To implement *Mlifdect*, there are three main challenges which need to be solved. First, the classification model of *Mlifdect* consists of six different classifiers, which may increase the overall time consumption. Second, it is difficult to combine the classification outputs of diverse classifiers as well. Third, different categories of features may have different sizes, and we may be at the risk of losing some categories if all features in some certain categories are not selected at all. To address these challenges, we first take advantages of the parallel machine learning to reduce the execution time and achieve the tradeoff between detection accuracy and time consumption. Then, we combine outputs of each classifier based on Dempster-Shaper theory and probability analysis, and obtain the final classification results. Finally, according to the size of each category, we divided them into two sets, and different feature selection algorithms will use them alternatively.

In summary, we make the following contributions to detect Android malware in this paper:

(i) We propose an Android malware detection approach based on parallel machine learning and information fusion. This approach integrates diverse algorithms and detect Android malware using various categories of features from Android malware, which can achieve a good detection results.

(ii) We take advantages of parallel machine learning to construct classification model, which could save computational resource and achieve better detection performance.

(iii) We investigate two information fusion techniques based on probability analysis and Dempster-Shaper theory to fuse probability outputs of each classifier and generate the final classification results.

(iv) We conduct extensive experiments on real-world dataset which contains 8,385 apps and compare the performance of our approach with several well-known Android malware detection approaches. The results show that our approach outperforms the existing approaches which can achieve 99.7% accuracy and 99.8% recall with 0.1% FP rate. What is more, *Mlifdect* can analyze an app within 5 seconds on average.

The rest of this paper is organized as follows. We discuss related works in Section 2. Section 3 introduces the static analysis on Android apps. Section 4 describes the design details of *Mlifdect*. We evaluate *Mlifdect* in Section 5 and conclude in Section 6.

2. Related Work

The analysis and detection of Android malware have been a vivid area of research in the past several years. Several categories of researches have been proposed to cope with the growing amount of more and more sophisticated Android malware. We divide these researches into four categories which are described as follows.

2.1. Detection Using Static Analysis. Static analysis has been widely used in the research community, and decompilation is the most common technique in this research category. Yang et al. implemented *AppContext* [7], which classified apps based on the contexts that trigger security-sensitive behaviors. Seo et al. [8]. investigated malware samples and determined suspicious APIs which were frequently used by malware. They listed suspicious APIs and compared the differences of feature occurrence frequency between malware and benign apps. Kang et al. [9] extracted permissions, suspicious APIs, and creator information such as serial number of certificate as features to classify malicious Android apps. *Flowdroid* [10] detected malware by building a precise model of Androids lifecycle, which helped to reduce missed leaks or false positives.

Our method *Mlifdect* is somewhat related to these approaches, but we employ more comprehensive features for charactering apps, such as hardware, network addresses, and intents. Moreover, our approach can detect unknown malware by developing diverse machine learning algorithms to learn features from known malwares.

2.2. Detection Using Dynamic Analysis. Dynamic analysis focuses on running apps in sandbox environment or in real devices in order to gather dynamic behaviors of apps in terms of accessing private data or using restricted API calls. For example, *AppsPlayground* performed functions like information leakage detection, sensitive API monitoring, and

kernel level monitoring [11]. Enck et al. presented *TaintDroid* [12], one of the popular system-wide dynamic taint tracking tools, and it monitored multiple sources of sensitive data. *RiskRanker* could detect particular malicious behaviors [13]. It could check whether the native code of an application contains the known exploit codes or not and capture certain behaviors like encryption or dynamic code loading.

Dynamic analysis can collect dynamic behavior of apps which is a complement to static analysis. However, the efficiency of this kind of approaches depends on code coverage during automatic execution. Moreover, some works (e.g., *TaintDroid*) need to modify Android OS to implement on smartphone, which is technically not feasible.

2.3. Detection Using Hybrid Approaches. There are also hybrid approaches that adopt both static analysis and dynamic analysis. Spreitzenbarth et al. [14] presented an automated analyzing system, Mobile-Sandbox, parsed an app's permissions and intents information, and analyzed their suspiciousness. Then, it performed dynamic analysis to log actions especially those based on native API calls. *DroidDetector* [15] extracted 192 features from both static and dynamic analyses including required permissions, sensitive API, and dynamic behaviors emulated using DroidBox and then detected malware using a DNN-based deep learning model. *Marvin* [16] collected requested permissions, components, SDK version, and a complete list of available Java methods during static analysis. Then, taint tracking, method tracing, and system call invocation recording were also performed in the dynamic analysis stage.

These methods typically consist of analyzing the application before installation and recording the execution behavior. The hybrid analysis method is gaining more popularity for its capability to dissect and investigate Android applications more accurately. However, they generally introduce a high time complexity and also have to solve the problems that dynamic analysis faced.

2.4. Detection Using Machine Learning. In recent years, several methods have been proposed based on various machine learning algorithms. These methods can analyze apps automatically and they work well in detecting unknown apps. Li et al. built their malware detection model using the support vector machine (SVM) algorithm on various features, including risky permissions and vulnerable API calls [17]. Yerima et al. developed a machine learning based approach based on improved eigenface algorithm [18]. Moreover, *DroidDeepLearner* [19] and *DroidDeep* [20] both identified malware using cutting-edge deep learning algorithm, which is more autonomous during the learning process so as to address the malware detection problem with less human intervention.

All of the above methods used traditional machine learning algorithms, which detect Android malware based on one or two classification algorithms. However, our proposed *Mlifdect* approach designs a novel classification model based on multiple algorithms and it also uses more comprehensive features to boost the detection performance.

3. Android Apps Profiling

In this section, we profile Android apps by using static analysis techniques. In terms of static analysis, we mainly focus on Android manifest files and disassembled dex code of Android apps. *AndroidManifest.xml* provides data supporting the installation and later execution of the app where we collect deployment of components, intents, requested permissions, and hardware.

(i) *App components:* there are four different types of components defined in an Android app: *activity*, *service*, *content provider*, and *broadcast receiver*. Each app can declare several components in the manifest file as required. Some malware families may share the same name of components. For example, several variants of the DroidKungFu malware use the same name for particular services [21] (e.g., *com.google.ssearch*).

(ii) *Intents*: this is responsible for communicating between apps and different components of an app. We found that some particular intents are more frequently used in Android malware than benign apps, such as *SIG_STR, BOOT_COMPLETED*. For example, *BOOT_COMPLETED* is used to trigger malicious activity directly after rebooting the smartphone.

(iii) *Requested permissions:* permission mechanism is one of the most important security mechanisms in Android platform. Permissions are actively granted by the user when an app is installed, and they are mainly used to limit the use of some functions and the access to some components among apps. In previous research works [22, 23], Android malware would request some particular permissions more frequently than benign samples. For example, majority of malware would request the *SEND_SMS* permission to send out premium SMS messages. We thus gather all permissions listed in the manifest to profile Android applications.

(iv) *Hardware:* if an app needs to request the camera, bluetooth, or the GPS module of the smartphone, these features have to be declared in the manifest file. Besides, if an app requests GPS and network at the same time, then it means that the app can read location information and reveal it out through network.

The second part is disassembled code of the apps which contains API calls, protected strings, commands, and network. The *dex* code can run in the Dalvik virtual machine directly, which is similar to exe files in Windows platform.

(i) *API calls:* API (application programming interface), documented in Android SDK, is a set of functions provided to control principal actions of Android OS. Certain API calls are frequently found in malware and may allow access to sensitive data or information of Android-enabled devices. As these API calls can specially lead to malicious behavior, they are extracted and gathered in a separated feature set.

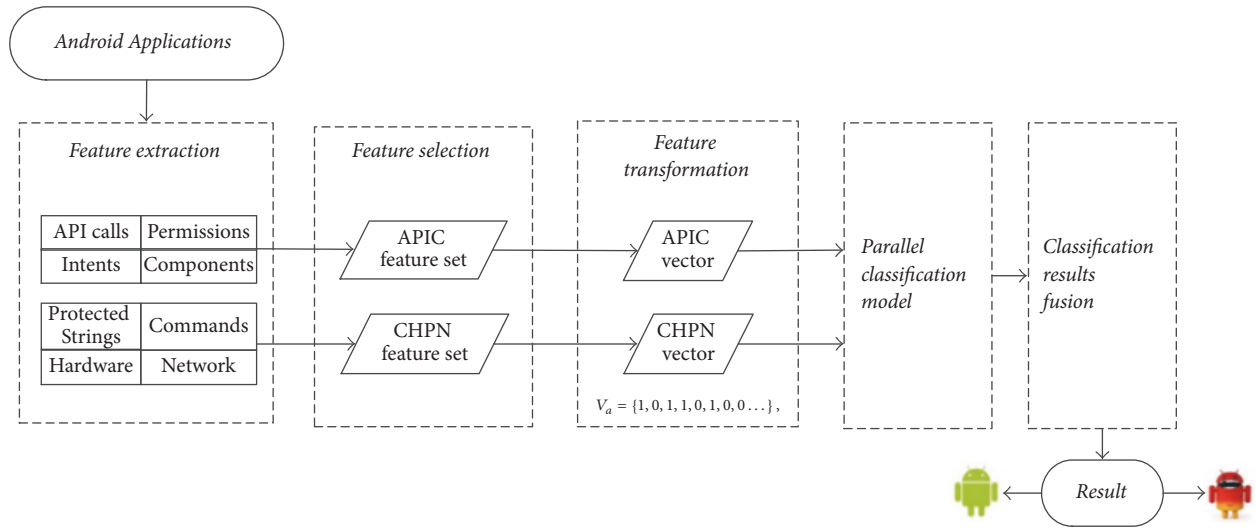

Figure 1: Architecture of Mlifdect.

(ii) *Protected strings:* Seo et al. [8] analyzed malware samples and defined some suspicious strings which were more often used by malware than benign apps. For example, malware may use *getSubscriberId* to disclose a smartphone's International Mobile Subscriber Identity (IMSI). *Cipher* is frequently used for obfuscation. And the string *content:sms* may mean leaking SMS contents.

(iii) *Commands:* Android OS is developed based on Linux and uses some commands as Linux. Seo et al. [8] also concluded some commands called by malware frequently. The mobile bot-net malware, Android-Spyware MinServ contained commands such as *note, push*. For example, Android apps can send a message to premium-rate numbers by using *note* and transmit user information to a remote control and command (C&C) server by using *push*.

(iv) *Network:* many Android apps require network access during running time. Ferreria et al. pointed out that the majority of apps currently use insecure network connections in [24]. Malware apps regularly leak privacy data out by establishing network connections. Some of network addresses might be involved in botnets. For example, malware from Basebridge family would send privacy data secretly to http://b3.8866.org:8080 [25]. Therefore, the IP addresses and URLs found in the disassembled code can be profiled for Android malware detection.

After Android apps profiling, Table 1 gives an example of features from these eight types of features. We will explain the details of extracting these aforementioned features in the next section.

4. System Design

Traditional machine learning algorithms try to learn one hypothesis from training data, such as decision tree and support vector machine. However, Android apps generally contain an overwhelming number of characteristics, and it is hard to find a classification algorithm that is suitable for all these various features. Therefore, one or two classification algorithms may not work well when dealing with more complicated real-world Android malware. Moreover, traditional feature selection is processed from all features, which may be at the risk of losing some small but useful categories.

In this section, we propose the *Mlifdect*, which is a novel detection approach based on parallel machine learning and information fusion. This approach utilizes a set of various classifiers to detect Android malware for the purpose of avoiding the phenomenon that one single classifier cannot effectively deal with all types of collected characteristics. Then, *Mlifdect* performs all these classification tasks in parallel to reduce time consumption brought by multiple classifiers.

The architecture of our approach is shown in Figure 1, which contains five main components. *Feature extraction* is responsible for extracting features that mentioned in Section 3 (which is presented in Section 4.1). *Feature selection* aims at traversing the entire feature sets and selecting the typical ones (which is discussed in Section 4.2). *Feature transformation* is responsible for transforming the extracted features into multidimensional vectors (which is presented in Section 4.3). *Parallel classification model* is used to build classification model trained with space vectors based on machine learning algorithms (which is described in Section 4.4). *Classification results fusion* is the last component, which focuses on combining the outputs of classifiers and generating the results (which is described in Section 4.5). Next, we will introduce each component in detail.

4.1. Feature Extraction. As the number of Android apps grows rapidly, it is a huge waste of time and human labor to extract features manually. Therefore, we design an automatic and extensible tool for decompressing apk files and extracting features. Our tool first uses APKTool [26], an open

Table 1: Examples of eight types of features from static analysis.

APP components	Intents	Requested permissions	Hardware	API calls	Protected Strings	Commands	Network
com.google.ssearch	BATTERY_CHANGED_ACTION	INTERNET	android.hardware.telephony	getDeviceId( )	HttpURLConnection	chmod	https://maps.google.com/
com.eguan.state	PHONE_STATE	SEND_SMS	android.hardware.wifi	findViewById()	getSimSerialNumber	root	http://www.umeng.com/
com.google.update	BOOT_COMPLETED	ACCESS_FINE_LOCATION	android.hardware.location	getApplicationContext()	sms to	getprop	http://media.admob.com/
...	...	...	...	...	...	...	...

```
<rule>
    <id>1</id>
    <category>Permission</category>
    <description>Permissions in manifest</
        description>
    <regex>android\.permission\.\w*\b|com\.android
        \.\w*\.permission\.\w*\b</regex>
    <targetfile>smali</targetfile>
</rule>
```

Listing 1: An example of feature extraction rule.xml.

source recompilation tool, to decompress the .APK files to *AndroidManifest.xml* and *dex* files. Then, several customized extraction rules are implemented by regex. Next, we collect and save the features by applying the rules to unpacked files. The rule used to extract permissions is shown as Listing 1.

In Listing 1, the category denotes the type of features we want. The value of regex is defined to match permissions such as *android.permission.SEND_SMS*. And the target file means that permissions will be matched in AndroidManifest.xml while API calls can be matched in dex code. In addition, the other categories are extracted in this way.

Based on aforementioned extraction approach, we totally extract more than 65,000 features which cover the eight different kinds of features for Android apps.

4.2. Feature Selection. After feature extraction, there are totally 65,804 features collected. However, the dimensions of features are too large which may introduce a very high computational overhead, which will impact the detection results [27]. In addition, some features are common in both the normal and malware samples, which may downgrade the overall quality of the classification model, such as *WRITE_EXTERNAL_STORAGE*, a permission used to obtain privilege to write data on SD card. Some features only appear in very few apps in our dataset, such as *DOWNLOAD_WITHOUT_NOTIFICATION*. Therefore, we need to select typical features before building a classification model based upon them.

In this work, we use feature selection algorithms to select typical features which are valuable to distinguish application classes. As explained before, we extracted eight kinds of features based on static analysis. Let us assume that all those features are selected together by a feature selection algorithm; some categories may have very few features left after the selection process, such as the command category. To avoid this situation, we divide the eight kinds of features into two sets. The first set contains four categories: API calls, requested permissions, intents, and components, which we name as the APIC set. The rest contains command, hardware, protected strings, and network URL, which is called CHPN.

Given the large number of features of APIC set, we use *FrequenSel* to select features and it has been proved that *FrequenSel* works well in selecting features from the APIC set. *FrequenSel*, which is proposed by *Fest*, selects features by discovering the differences of the feature occurrence frequency between malware and benign apps and selects features that have higher occurrence frequency than a predefined threshold value in either malware or benign (the details can be found in [4]). After selection, we totally obtain 287 features, which is called the *APIC feature set*.

On the other hand, the size of features in CHPN set is relatively less than that in the APIC set; the feature occurrence frequency in CHPN set also is less than that in APIC set. If we still use *FrequenSel*, we may miss some useful features. Therefore, we utilize the *information gain* algorithm [28] to select features from CHPN set. Finally, we obtain 99 features, which is named as the *CHPN feature set*. In the experiment section, we will explain the reason why we choose these two kinds of selection algorithms in detail.

4.3. Feature Transformation. Malicious activities are usually reflected in specific patterns and combinations of the extracted features. For example, an instance sending premium SMS messages needs the permission *SEND_SMS* in the set of requested permission. Ideally, we would like to formulate Boolean expressions that capture these dependencies between features and return true if a malware is detected. However, inferring Boolean expressions from real-world applications is a hard problem and it is difficult to solve efficiently.

To address this problem, we aim at capturing the dependencies between features using concepts from machine learning. As most learning approaches operate on numerical vectors, we first need to map the extracted feature sets into a vector. For this purpose, we define a joint set F that includes all observable strings contained in the eight feature sets, which can be obtained by applying the below equation, in which f_i represents the corresponding feature.

$$F = \{f_1, f_2, \ldots, f_n\}. \tag{1}$$

We ensure that elements of different sets do not collide by adding a unique prefix to all strings in each feature set. Using the set F, we define an $|F|$-dimensional vector space, where each dimension is either 0 or 1. A sample a is mapped to this space by constructing a vector V:

$$V = \{v_1, v_2, \ldots, v_n\},$$
$$v_m = \begin{cases} 1 & \text{if } f_m \in F_a \\ 0 & \text{if } f_m \notin F_a. \end{cases} \tag{2}$$

Thus a feature vector can be translated into $V = \{0, 1, 0, 0, \ldots\}$; 1 indicates that the feature is contained in this sample, whereas 0 indicates that the feature is not contained in this sample. However, V is often a sparse vector; in order to reduce the storage overhead, we transform V to a compressed format V^*. Assuming that the features are arranged in a fixed order, then we can index a feature by its position, and V^* is defined as follows:

$$V^* = \{2, 5, 9, \ldots\}. \tag{3}$$

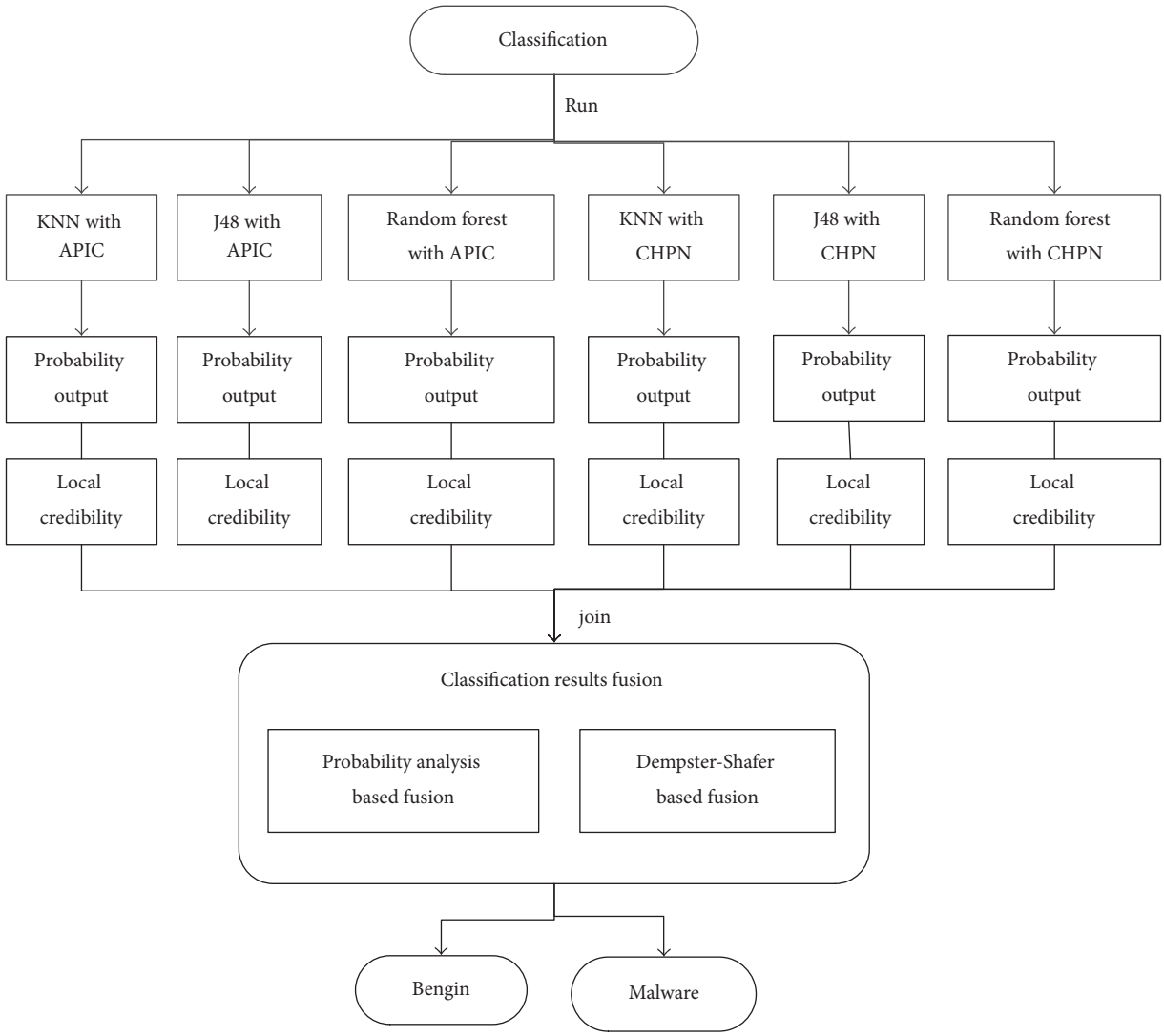

Figure 2: Classification model.

The positions of nonzero elements in V are stored in V^*, which saves a great amount of memory space. In this way, we can obtain two kinds of vector spaces which will be used to train or test classifiers based on the APIC feature set and CHPN feature set.

4.4. Parallel Classification Model. Prior machine learning works try to learn one hypothesis from training data, and it is hard to find an ideal algorithm that works best for all different types of Android malware. To solve this problem, our approach applies various machine learning classifiers in parallel for identifying Android malware, which has many potential benefits more than just accuracy improvement [29]. For instance, it is possible to speed up the process of prediction and harness the various strengths of the constituent classifiers, such as complementing white box analysis through close observation of intermediate output from base classifiers. Figure 2 shows an overview of classification model based on parallel machine learning. As we can see, the classification model contains six different classifiers, and each classifier outputs the probabilities that apps are benign. Then, local credibility, which is defined to describe the confidence measure of a classifier, is calculated for the information fusion. The details of *Mlifdect* classification algorithm are shown in Algorithm 1.

First, we input three machine learning algorithms (KNN, random forest, and J48) and two kinds of training sets (APIC and CHPN) as variables. Then, we create six threads and dispatch a kind of machine learning algorithm and training set to each thread (lines (1)–(9)). For example, J48 with APIC means that J48 algorithm and APIC training set are used to build a classifier, while the random forest with CHPN classifier is constructed by random forest algorithm and CHPN training set. After threads creation and data distribution, the classification model begins to activate the threads separately and concurrently to construct classifiers. For each classifier, the possibility that an app belongs to benign will be calculated and displayed, and the local credibility will be calculated (lines (10)–(14)).

```
Input: CPHN, APIC
Output: probabilities and local credibility
(1) alg[ ] = {KNN, random forest, J48}, data[ ] = {APIC, CHPN};
(2) for i from 0 to 2 do
(3)     for j from 0 to 1 do
(4)         create a thread
(5)         dispatch alg[i] and data[j] to the thread
(6)         j ⇐ j + 1
(7)     end for
(8)     i ⇐ i + 1
(9) end for
(10) run all threads, start classification
(11) for each thread do
(12)    output the probabilities of apps
(13)    calculate the local credibility
(14) end for
```

Algorithm 1: Classification.

The strengths of using different classifiers for malware detection lie in multiple aspects. For example, the interpretable intermediate possibility can be useful for conducting further analysis and the local credibility is crucial for the Basic Probability Assignments (BPAs) which will be used in Dempster-Shafer theory.

After calculating the possibilities p of each class (benign or malware) from each individual classifier, confusion matrix is used to obtain the local credibility. Confusion matrix is commonly used in supervised learning, which can reflect the relationship between the classification results and actual values. Confusion matrix can be expressed as follows:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}. \tag{4}$$

In confusion matrix, the data of each row represents the number of real samples, while the data of each column represents the sample number of classification results. Assume that an arbitrary data of confusion matrix is represented as C_{ij} ($i \in [0,1]$, $j \in [0,1]$); p represents the probability of real sample when the output category of a classifier is n. The following equation states that

$$\mathrm{lc}_i = \frac{C_{ii}}{\sum_{j=0}^{1} C_{ij}}, \tag{5}$$

where lc_i will be localized as a confidence measure of the classifier. Finally, we define $M(0) = \mathrm{lc}_0 * (1 - p)$ and $M(1) = \mathrm{lc}_1 * p$, where M is the BPA of Dempster-Shafer fusion.

4.5. Classification Results Fusion. In order to obtain the baseline results for investigating the parallel classifiers approach to Android malware detection, the information fusion approach which involved a combination of decisions from each individual classifier was developed. Two different fusion schemes were considered.

Probability Analysis Based Fusion. This method is inspired by traditional voting strategies. The majority voting, which is generally used, has some limitations, such that a subset of classifiers (majority in number) may agree on the misclassification of an instance by chance. It is suitable for detecting benign applications and lacks the accuracy in the detection of malware. On the contrary, the veto voting may affect the classification performance as outcome may depend only on one single algorithm and it is more suitable for detecting malware [30], while our new method can overcome the deficiencies and evaluate both malware and benign instances more objectively.

```
Input: probabilities outputs of apps
Output: classification results
(1) for i from 1 to 6 do
(2)     collect p_i
(3) end for
(4) p = p_1 + p_2 + p_3 + p_4 + p_5 + p_6
(5) if p <= threshold then
(6)         return 0
(7) else
(8)         return 1
(9) end if
```

Algorithm 2: Probability analysis based fusion.

As shown in Algorithm 2, we sum up the probabilities (p) from each individual classifier, where p_i is used to quantify the possibility that an instance does not belong to malware, and calculate as equation (line (5)). Finally, the algorithm compares p with predefined threshold. If the value of sum is smaller than the threshold, the algorithm will return 0 and identify this app as malicious; otherwise the algorithm will return 1 and regard it as benign (lines (5)–(9)).

Dempster-Shafer Theory Based Fusion. It was first introduced in 1976 by Shafer [31] as an extension of Dempster's probabilities on multivalued mapping [32]. Dempster-Shafer theory has a strong ability to deal with uncertain information and

needs weaker conditions than Bayesian theory. Whether an Android app is malware is an obvious uncertain problem and the outputs of diverse classifiers are independent of each other, which satisfy the conditions of Dempster-Shafer theory.

In our approach, all the .apk files should be represented as $\Theta = 0, 1$. 0 represents malware, and 1 represents normal instance. The following conditions should be met:

$$M^{\Theta}(\emptyset) = 0, \quad \sum_{A \in 2^{\Theta}} A = 1, \tag{6}$$

where function M is defined as BPA; $M(0)$ and $M(1)$ are calculated in Section 4.4. If multiple sources exist, Dempster-Shafer evidence combination formula can be defined as follows:

$$(m_1 \oplus m_2 \oplus \cdots \oplus m_n)(A) = \frac{1}{K} \sum_{A_1 \cap A_2 \cap \cdots \cap A_n = A} m_1(A_1) * m_2(A_2) * \cdots * m_n(A_n), \tag{7}$$

where the constant K can be calculated as

$$K = \sum_{A_1 \cap A_2 \cap \cdots \cap A_n \neq \emptyset} m_1(A_1) * m_2(A_2) * \cdots * m_n(A_n) = 1 - \sum_{A_1 \cap A_2 \cap \cdots \cap A_n = \emptyset} m_1(A_1) * m_2(A_2) * \cdots * m_n(A_n) \tag{8}$$

K is a coefficient that reflects the degree of conflict of evidences, which is in the range of $[0, 1]$. If K is closer to 1, it means greater conflict between evidences. On the contrary, when K is closer to 0, it indicates that the conflict is smaller. $1/(1 - K)$ is a normalization factor which can prevent a nonzero value be assigned to empty sets.

5. Evaluation

To validate the proposed *Mlifdect*, we perform an empirical evaluation regarding its effectiveness and efficiency. Our dataset contains 8,385 applications, 3,982 of which are malware and the rest are benign samples. Malware samples are collected from Drebin [3] and Android Malware Genome Project [33], and benign samples are downloaded from Google Play and scanned with VirusTotal [34], to ensure that none of them is malicious. In particular, we deploy our detected system in a laptop which is equipped with 8 G memory, i5-4219U CPU, and windows 7 OS and then conduct the following three experiments:

(1) *Detection performance*: first, we evaluate the detection performance of *Mlifdect* on an app dataset of 3,982 malware and 4,403 normal apps using 10-fold cross-validation.

(2) *Performance comparison*: next, we compare the detection performance of *Mlifdect* with some state-of-the-art approaches, including *Drebin* [3] and emphFest [4], as well as two detection approaches presented by Yerima et al. One is based on improved eigenface algorithm [18] and another is based on ensemble learning [6], which we call them *Eigenspace* and *HAEL*.

(3) *Run-time performance*: finally, we evaluate the run-time performance of *Mlifdect*. For this experiment, we use different measurements including usage rate of memory and CPU with the same machine.

5.1. Detection Performance. In our first experiment, we evaluate the detection performance of *Mlifdect* using 10-fold cross-validation. Moreover, we use three types of metrics to evaluate the detection performance which are accuracy, recall, and F-measure. And they are defined as follows, where TP, FP, TN, and FN are the number of True Positive, False Positive, True Negative, and False Negative samples.

$$\text{Accuracy} = \frac{(\text{TP} + \text{TN})}{(\text{TP} + \text{FP} + \text{FN} + \text{TN})}, \quad \text{Recall} = \frac{\text{TP}}{(\text{TP} + \text{FN})}, \quad F\text{-measure} = 2 * \frac{\text{TP}}{(2 * \text{TP} + \text{FP} + \text{FN})}. \tag{9}$$

5.1.1. Detection with Different Feature Selection Algorithms. First of all, we evaluate performance of *Mlifdect* to find the most suitable feature selection algorithms. For this purpose, we use two categories of feature sets (mentioned in Section 4.4) and choose two feature selection algorithms: *information gain* and *FrequenSel* to select typical features from the two feature sets. Therefore, we can obtain four kinds of combinations, such as using *FrequenSel* to select both feature sets or using *information gain* to select features from APIC feature set while *FrequenSel* to choose from CHPN feature set. Then, we utilize KNN, random forest, and J48 algorithms to build the classification model based on the four different kinds of selected feature sets. Moreover, the Dempster-Shafer theory based fusion is evaluated in the experiment.

Figures 3, 4, and 5 show the accuracy, recall, and F-measure results of the four different selected feature sets, respectively. We observe that when using *FrequenSel* to select features from APIC feature set and using *information gain* to select CHPN feature set, we can achieve the highest accuracy, recall, and F-measure. This result also means that this type of combination can build the best classification model than the others.

It is known that *information gain* would lead to distribution bias and long tail effect, which cannot properly select typical features for classification. APIC feature set consists of a large number features, and about 88% of them selected by *information gain* make contributions under 0.01. In other words, most features distributed on the tail are unimportant, and they contribute very little to machine

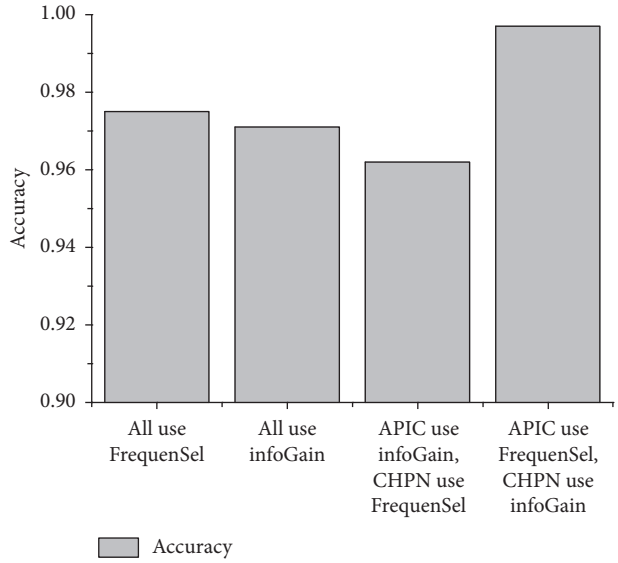

Figure 3: Accuracy of Mlifdect with different feature selection algorithms.

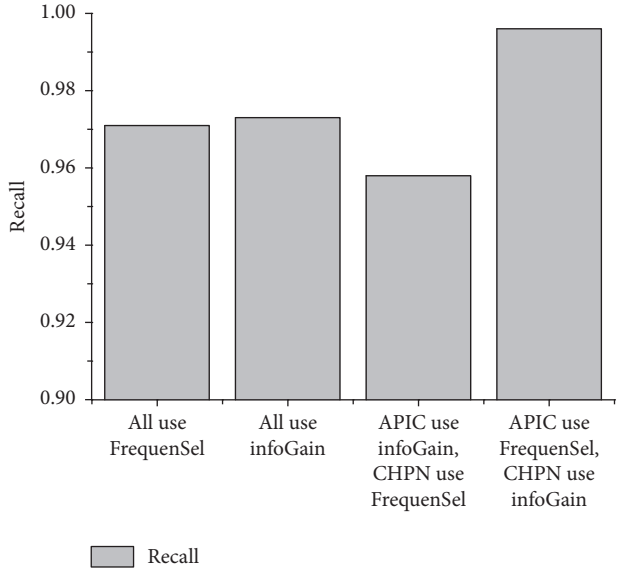

Figure 4: Recall of Mlifdect with different feature selection algorithms.

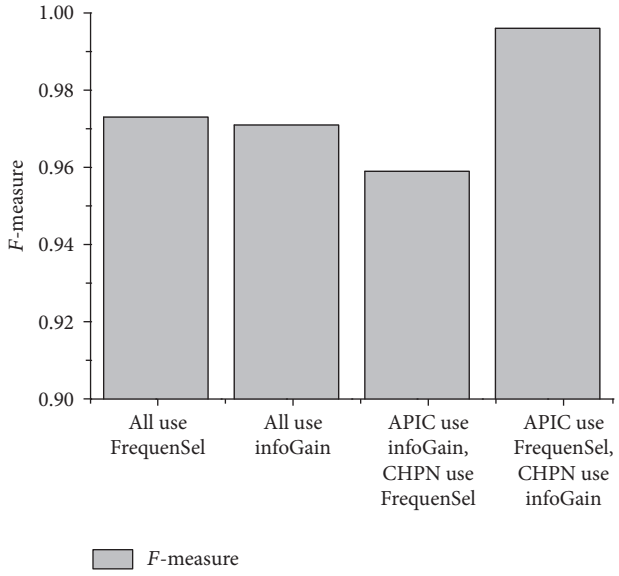

Figure 5: F-measure of Mlifdect with different feature selection algorithms.

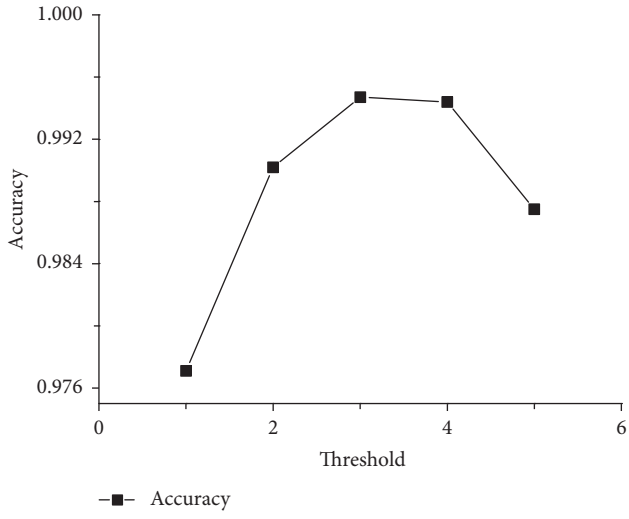

Figure 6: Accuracy of Mlifdect with different thresholds.

learning algorithms. Moreover, we have to manually cut off the tail in different positions, which will change the number of features and influence the accuracy of classification.

CHPN feature set contains far less features than APIC, and 30 features only remain after being filtered by *FrequenSel*, which cannot make a good use of the machine learning algorithms. So we use *information gain* in turn and discard the features which have no contribution to the classification. Results have shown that it works better. And in the following experiments, we utilize *FrequenSel* to select APIC feature set and *information gain* to select CHPN feature set.

5.1.2. Detection with Different Thresholds. Following that, we will explore the value of threshold that can get best performance of *Mlifdect* with the probability analysis based fusion. After feature selection, we build the classification model. Then, we combine the outputs of individual classifier when threshold $= 1, 2, 3, 4, 5$, and the results are shown in Figures 6, 7, and 8.

From the three figures, we have three observations: (1) the result of accuracy can achieve 99.7% when the theshold = 3. (2) When threshold = 4, the result of recall can achieve the highest result compared to other values of threshold. (3) The result of F-measure is similar to accuracy.

Based on observations towards output probability of each individual classifier, we find that there are 95% of apps whose probabilities range from 0 to 0.1 or 0.9 to 1. Consequently, when the threshold is set as 1 or 5, *Mlifdect* has already achieved a good detection performance. If the sum

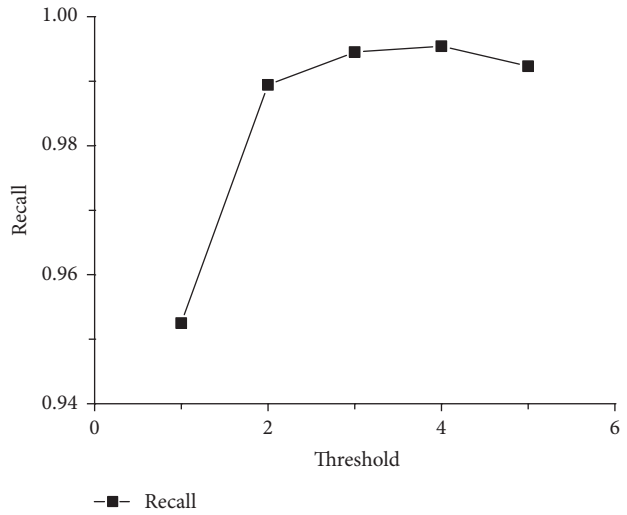

Figure 7: Recall of Mlifdect with different thresholds.

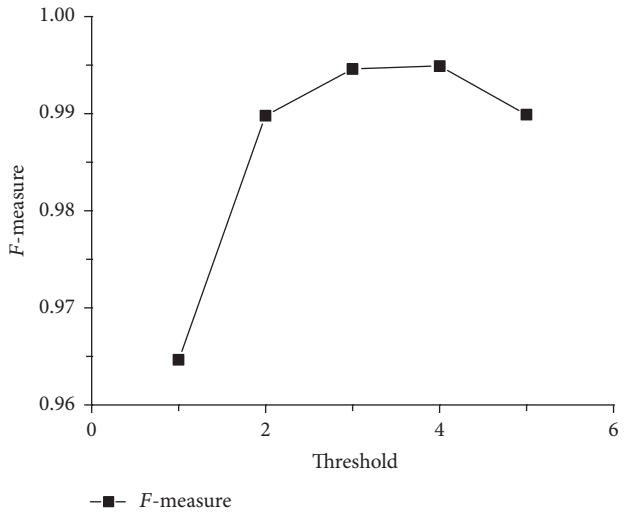

Figure 8: *F*-measure of Mlifdect with different thresholds.

of probabilities result of the six classifiers is smaller than 3, it means that more than half of the classifiers consider the instance as malware. Therefore, with the threshold growing from 1 to 3, the accuracy and recall will increase accordingly. Moreover, there are only a few apps whose sum results are between 3 and 4, when threshold equals 3 or 4 they have similar accuracy and recall. Finally, when the sum of probabilities result is larger than 4, it is obvious that there are more than half of classifiers which identify it as a benign one. In this situation, assuming the threshold is increased to 5, it would lead to a decrease of classification performance. Therefore, 3 is the best threshold value, and we set theshold = 3 in the following experiments.

5.2. Results Comparison

5.2.1. Comparison with Related Approaches. In this experiment, we compare the detection results of *Mlifdect* with several static analysis based approaches for detection of Android malware. In particularly, we consider *Eigenspace* [18], *HAEL* [6], *Fest* [4], and *Drebin* [3]. Moreover, we also compare *Mlifdect* with several single classification algorithms, such as KNN and random forest.

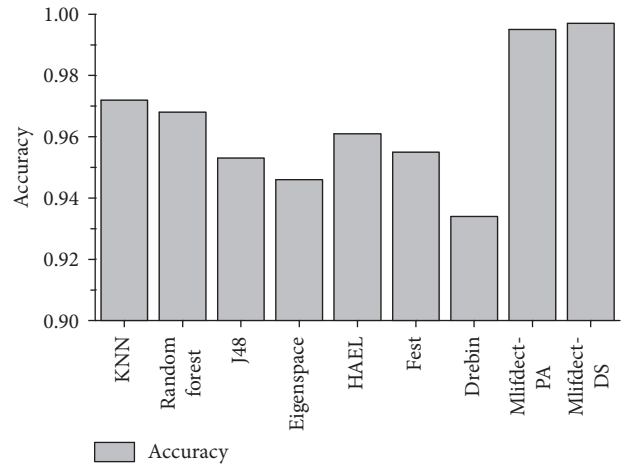

Figure 9: The comparison of accuracy.

Here, we still use the same scheme to select features from two categories of feature sets and use 10-fold cross-validation approach to build classification model. What is more, the two information fusion methods are measured, and the Dempster-Shafer theory based fusion is called *Mlifdect-DS* whereas *Mlifdect-PA* represents the probability analysis based fusion.

As shown in Figure 9, the fusion method using Dempster-Shafer theory is a little better than using probability analysis, which can achieve 99.7% accuracy. And the best accuracy comes from *Mlifdect* which is attributed to the integration of six base classifiers, including KNN, random forest, and J48, and makes prediction with information fusion. Note that our approach is superior to *HAEL* even if it is also an ensemble learning algorithm. Moreover, compared to *Eigenspace*, *Fest*, and *Drebin*, *Mlifdect* extracts eight types of features, which can characterize apps in a more comprehensive manner.

5.2.2. Classification of Unknown Apps. In order to verify whether our approach is efficient or not, the performance of classifying unknown applications is also evaluated. We utilize *Mlifdect* and some related works to detect 2,350 unknown apps which are downloaded from some third-party markets (e.g., Gfan). Table 2 presents the detection results.

In Table 2, *Mlifdect* achieves more outstanding accuracy, recall, and *F*-measure, all of which are close to 98.3% in identifying unknown apps. The results demonstrate that our malware detection model can achieve high accuracy and recall in classifying real-world apps even with limited prior knowledge on them.

5.3. Run-Time Performance. To analyze the run-time performance of *Mlifdect-DS* which has the highest accuracy rate, we use memory and CPU utilization rate as well as running time as our evaluation metric.

Table 2: Result of unknown apps classification.

Alg	Correctly classified	Accuracy	Recall	F-measure
Eigenspace	2,148	91.4%	91.2%	91.8%
HAEL	2,254	95.5%	96.1%	95.9%
Fest	2,165	92.2%	92.9%	92.6%
Drebin	2,109	89.7%	89.9%	90.1%
Mlifdect-PA	2,309	98.2%	98.3%	98.2%
Mlifdect-DS	2,314	98.5%	98.3%	98.4%

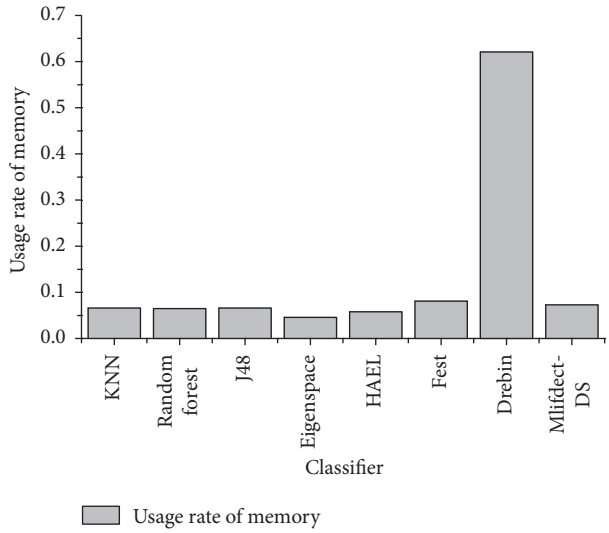

Figure 10: The max usage rate of memory.

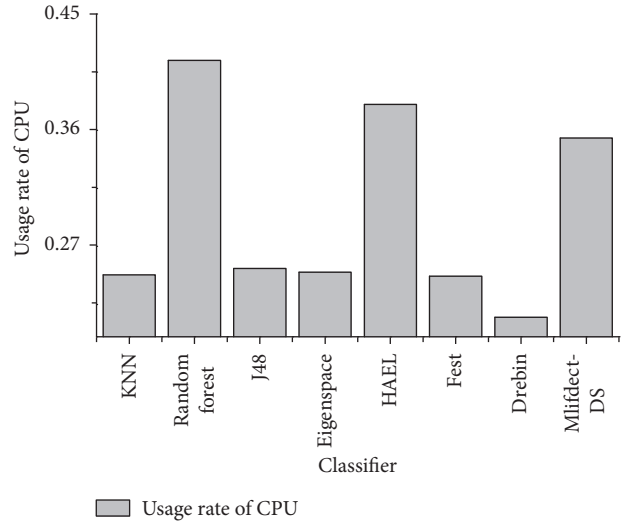

Figure 11: The average usage rate of CPU.

5.3.1. Usage Rate of Memory and CPU. For comparison, we consider the some other methods including *Enginspace*, *HAEL*, *Fest*, *Drebin* and three other single classifiers which use KNN, random forest, and J48, respectively. Figure 10 shows the max usage rate of memory. The information stored in memory includes loading files (training set for classification), code, and global variables in program. For this reason, *Mlifdect* achieves a similar performance when compared to the three single classifiers, and it takes a little bit less memory than *Fest*. *Eigenspace* and *HAEL* save a little amount of memory because they use less features, while *Drebin* consumes a large amount of memory because it works without feature selection and has to deal with a huge training file.

The usage rate of CPU is defined as the ratio of CPU busy time and the whole cycle within the task manager refresh cycle. As we can see from Figure 11, the classifiers built with random forest including *HAEL* run fast so that the usage rate of CPU are higher than others. Our approach combines six classifiers which are built with KNN, random forest, and J48, so it takes more time to classification than any single classifier. However, due to the design of parallel processing, *Mlifdect* compromises the average usage rate of CPU among the three single classifiers, whereas it enhances the rate when compared to *Eigenspace* and *Fest*.

5.3.2. Running Time of Mlifdect. Table 3 demonstrates that *Mlifdect* outperforms *Eigenspace*, *Fest*, and *Drebin* in terms of the time spent in building model and classifying completely. *Drebin* costs too much time in building model and classifying because of the high dimensions of features. *HAEL* performs very fast because it only uses random forest algorithm. By using parallel machine learning, our approach has the win-win situation of detection accuracy and detection time.

Table 3: The comparison of detection time.

Alg	Building model	Classifying
Eigenspace	187.1 s	842.9 s
HAEL	4.34 s	52.9 s
Fest	59.9 s	483 s
Drebin	467 s	2552 s
Mlifdect	25.4 s	103.6 s

In summary, all results of these experiments clearly show that *Mlifdect* can achieve high accuracy and high efficiency in real-world apps scenario without requesting for substantial hardware support.

6. Conclusion

In this paper, we introduce *Mlifdect*, an Android malware detection approach based on parallel machine learning and information fusion. *Mlifdect* combines concepts from static analysis, machine learning, and information fusion. In this

work, we first extract a total of 65,804 features from eight types of features of Android apps. Next, we concurrently build the classification model which contains six different classifiers based on three algorithms (KNN, random forest, and J48) and two kinds of features sets (APIC and CHPN) selected by *FrequenSel* and *information gain*, respectively, and then we use the fusion method with probability analysis and Dempster-Shafer theory to identify Android malware samples. Our evaluation results depict the potential of this approach, where *Mlifdect* outperforms other related approaches and it can classify Android benign and malware apps with 99.7% accuracy. Moreover, we also evaluate the run-time performance, which shows that *Mlifdect* introduces relatively low classification overhead. Thus, we consider our approach proposed in this paper as an effective yet lightweight solution to classify real-world Android apps. Moreover, the basic machine learning based classifiers can provide interpretable intermediate output that can be useful for further analysis if needed.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by the National Science Foundation of China under Grant no. 61472130, the Science and Technology Projects of Hunan Province (no. 2016JC2074), the Research Foundation of Education Bureau of Hunan Province, China (no. 16B085), and the Open Research Fund of Key Laboratory of Network Crime Investigation of Hunan Provincial Colleges (no. 2016WLFZZC008).

References

[1] IDC, "Apple, huawei, and xiaomi finish 2015 with above average year-over-year growth, as worldwide smartphone shipments surpass 1.4 billion for the year," ifundefinedselectfont, 2016, http://www.idc.com/getdoc.jsp?containerId=prUS40980416.

[2] Symantec, "internet security threat report," ifundefinedselectfont, 2016, https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf.

[3] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: effective and explainable detection of android malware in your pocket," in *Proceedings of the NDSS Symposium 2014*, February 2014.

[4] K. Zhao, D. Zhang, X. Su, and W. Li, "Fest: a feature extraction and selection tool for Android malware detection," in *Proceedings of the 20th IEEE Symposium on Computers and Communication, (ISCC '15)*, pp. 714–720, July 2015.

[5] Y. Du, X. Wang, and J. Wang, "A static android malicious code detection method based on multi-source fusion," *Security and Communication Networks*, vol. 8, no. 17, pp. 3238–3246, 2015.

[6] S. Y. Yerima, S. Sezer, and I. Muttik, "High accuracy android malware detection using ensemble learning," *IET Information Security*, vol. 9, no. 6, pp. 313–320, 2015.

[7] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "AppContext: differentiating malicious and benign mobile app behaviors using context," in *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE '15)*, vol. 1, pp. 303–313, IEEE, May 2015.

[8] S.-H. Seo, A. Gupta, A. M. Sallam, E. Bertino, and K. Yim, "Detecting mobile malware threats to homeland security through static analysis," *Journal of Network and Computer Applications*, vol. 38, no. 1, pp. 43–53, 2014.

[9] H. Kang, J.-W. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, Article ID 479174, 2015.

[10] S. Arzt, S. Rasthofer, C. Fritz et al., "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pp. 259–269, ACM, June 2014.

[11] V. Rastogi, Y. Chen, and W. Enck, "AppsPlayground: automatic security analysis of smartphone applications," in *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY '13)*, pp. 209–220, ACM, February 2013.

[12] W. Enck, P. Gilbert, S. Han et al., "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, article 5, 2014.

[13] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*, pp. 281–294, June 2012.

[14] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: having a deeper look into Android applications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*, pp. 1808–1815, Association for Computing Machinery, March 2013.

[15] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, Article ID 7399288, pp. 114–123, 2016.

[16] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "MARVIN: efficient and comprehensive mobile app classification through static and dynamic analysis," *Computer Software and Applications Conference*, pp. 422–433, 2015.

[17] W. Li, J. Ge, and G. Dai, "Detecting malware for android platform: an SVM-based approach," in *Proceedings of the 2nd IEEE International Conference on Cyber Security and Cloud Computing*, pp. 464–469, November 2015.

[18] S. Y. Yerima, S. Sezer, and I. Muttik, "Android malware detection: an eigenspace analysis approach," in *Proceedings of the Science and Information Conference, (SAI '15)*, pp. 1236–1242, IEEE, London, UK, July 2015.

[19] Z. Wang, J. Cai, S. Cheng, and W. Li, "DroidDeepLearner: identifying android malware using deep learning," in *Proceedings of the 2016 IEEE 37th Sarnoff Symposium*, pp. 160–165, Newark, NJ, USA, September 2016.

[20] X. Su, D. Zhang, W. Li, and K. Zhao, "A deep learning approach to android malware feature learning and detection," in *Proceedings of the 2016 IEEE Trustcom/BigDataSE/ISPA*, pp. 244–251, Tianjin, China, August 2016.

[21] X. Jiang, "Security alert: new droidkungfu variant," ifundefinedselectfont, 2011, https://www.csc.ncsu.edu/faculty/jiang/DroidKungFu3/.

[22] B. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: a perspective combining risks

and benefits," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, pp. 13–22, ACM, June 2012.

[23] K. A. Talha, D. I. Alper, and C. Aydin, "APK Auditor: permission-based android malware detection system," *Digital Investigation*, vol. 13, pp. 1–14, 2015.

[24] D. Ferreira, V. Kostakos, A. R. Beresford, J. Lindqvist, and A. K. Dey, "Securacy: an empirical investigation of android applications' network usage, privacy and security," in *Proceedings of the 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks, (WiSec '15)*, June 2015.

[25] Android, "basebridge," ifundefinedselectfont, 2011, http://www.symantec.com/securityresponse/writeup.jsp?docid=2011-060915-4938-99&tabid=2.

[26] Apktool, ifundefinedselectfont, http://code.google.com/p/android-apktool/.

[27] M. Z. Mas'Ud, S. Sahib, M. F. Abdollah, S. R. Selamat, and R. Yusof, "Analysis of features selection and machine learning classifier in android malware detection," in *Proceedings of the 5th International Conference on Information Science and Applications, ICISA 2014*, pp. 1–5, IEEE, May 2014.

[28] J. T. Kent, "Information gain and a general measure of correlation," *Biometrika*, vol. 70, no. 1, pp. 163–173, 1983.

[29] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, Springer, Berlin, Germany, 2001.

[30] R. K. Shahzad and N. Lavesson, "Comparative analysis of voting schemes for ensemble-based malware detection," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, vol. 4, no. 1, pp. 98–117, 2013.

[31] G. Shafer, *A Mathematical Theory of Evidence*, Princeton University Press, Princeton, NJ, USA, 1976.

[32] A. P. Dempster, "Upper and lower probabilities induced by a multivalued mapping," *Annals of Mathematical Statistics*, vol. 38, pp. 325–339, 1967.

[33] Y. Zhou and X. Jiang, "Dissecting android malware: characterization and evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pp. 95–109, San Francisco, Calif, USA, May 2012.

[34] V. Total, ifundefinedselectfont, 2013, https://www.virustotal.com/en/.