Research Article

MobiSentry: Towards Easy and Effective Detection of Android Malware on Smartphones

Bingfei Ren,1 Chuanchang Liu,1 Bo Cheng,1 Jie Guo,2 and Junliang Chen1

1State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing, China
2Beijing Liyun Technology Development Company, Beijing, China

Correspondence should be addressed to Chuanchang Liu; lcc3265@bupt.edu.cn

Received 9 July 2018; Accepted 8 October 2018; Published 21 November 2018

Academic Editor: Wenchi Cheng

Copyright © 2018 Bingfei Ren et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Android platform is increasingly targeted by attackers due to its popularity and openness. Traditional defenses to malware are largely reliant on expert analysis to design the discriminative features manually, which are easy to bypass with the use of sophisticated detection avoidance techniques. Therefore, more effective and easy-to-use approaches for detection of Android malware are in demand. In this paper, we present MobiSentry, a novel lightweight defense system for malware classification and categorization on smartphones. Besides conventional static features such as permissions and API calls, MobiSentry also employs the N-gram features of operation codes (n-opcode). We present two comprehensive performance comparisons among several state-of-the-art classification algorithms with multiple evaluation metrics: (1) malware detection on 184,486 benign applications and 21,306 malware samples, and (2) malware categorization on DREBIN, the largest labeled Android malware datasets. We utilize the ensemble of these supervised classifiers to design MobiSentry, which outperforms several related approaches and gives a satisfying performance in the evaluation. Furthermore, we integrate MobiSentry with Android OS that enables smartphones with Android to extract features and to predict whether the application is benign or malicious. Experimental results on real smartphones show that users can easily and effectively protect their devices against malware through this system with a small runtime overhead.

1. Introduction

Smartphones are becoming increasingly popular. Over 334 million smartphones were shipped worldwide in Q1 2017 according to the IDC report [1], and smartphones running Android dominate the mobile market with an 85.0% market share. As there are popularity and openness of Android, numerous applications (apps) are developed on the Android platform to provide new or enhanced services. The success of Android also has a dark side: more and more attackers turn their eyes on Android owing to its openness. As many security companies have recently reported [2–4], the overall volume of Android malware has been increasing significantly in 2016, and 39,000 new malware variants are developed using sophisticated detection avoidance techniques to elude detection.

As smartphones are increasingly used to process sensitive personal information and applications in Android often have the ability to access sensitive resources on mobile devices, it becomes easier for malware to breach privacy data and steal money of users without their consent [5]. For instance, sending SMS to premium-rate numbers secretly and encrypting data and locking devices using ransomware [6, 7]. Android provides permission mechanism to restrict the installation and resources usage of Android malware [5], which forces applications to explicitly request permissions when accessing protected resources (data, system services, peripheral, etc.). However, it can be easily bypassed through collusion attacks, and many users tend to blindly grant permissions due to lack of security knowledge. Therefore, it is a crucial and challenging issue to directly detect malware and novel malware variants on smartphones.
Traditional literature on the detection of Android malware mainly focused on static and dynamic analysis. For example, Apposcopy [8], DroidMat [9], and RiskRanker [10] are methods that use a combination of static features (permissions, deployment of components, and API calls) to characterize applications. However, these methods mainly rely on manually crafted detection patterns and can be rendered largely ineffective due to the production of novel malware variants. TaintDroid [11] and DroidRanger [11] use dynamic features through monitoring the behaviors of the running apps to identify malicious events of them. Unlike static analysis methods, approaches using dynamic features are efficient to detect new malware but usually suffer a significant overhead on smartphones. Machine learning is a buzzword in this area of malware detection. Various methods have been proposed based on the static features (API calls and permissions) analyzing approach that utilizes machine learning methods [12–14]. However, these approaches require specific static features beforehand through expert analysis, and most of them can be bypassed with the use of obfuscation techniques like repackaging [15]. Other works have been proposed for the categorization of malware families [16, 17]. However, most of these methods either use outdated datasets or focus on large families for which training data are available.

To address issues identified above, we present a novel lightweight defense system (MobiSentry) for malware classification and categorization on smartphones based on our preliminary version of this study [18]. Besides conventional static features such as permissions and API calls, MobiSentry also employs the N-gram features of operation codes (n-opcode), which can be used to deal with known obfuscation techniques like rename obfuscation (https://www.preemptive.com/obfuscation). We first investigate n-opcode analysis for both application classification and categorization on real datasets utilizing several state-of-the-art classification algorithms. Then, we conduct the performance comparisons of malware detection and categorization among these classification algorithms with multiple evaluation metrics. We then utilize the ensemble of these supervised classifiers to design MobiSentry and evaluate its performance. In addition, we integrate MobiSentry with Android OS to enable smartphones to extract features from Android Package (APK) files and to predict whether the application is benign or malicious with a small run-time overhead. The main contributions are the following:

(i) We propose a lightweight defense system for malware detection on smartphones. Unlike all related works [8, 11–13, 19–21], we systematically analyze the combination of conventional static features and n-opcode to make full use of them on malware detection and categorization.

(ii) We propose a novel ensemble learning algorithm based on five state-of-the-art supervised classifiers to overcome the drawbacks of them and improve the performance on malware detection.

(iii) We firstly investigate and evaluate several approaches on extracting features from APK files to reduce the run-time overhead on smartphones.

(iv) We thoroughly evaluate our proposed system under a large scale of dataset to present a more real-world performance.

The remaining parts of this paper are organized as follows: we briefly refer to some related studies in Section 2. The dataset is presented in Section 3. Section 4 describes conventional static features and n-opcode features of the opcode of applications. Section 5 introduces the methods of designing MobiSentry utilizing the ensemble of five supervised classifiers. Detailed comprehensive performance comparisons of malware detection and categorization are presented in Section 6 and Section 7. In Section 8, we evaluate the effectiveness and performance of MobiSentry integrated with Android OS on real smartphones. Finally, we conclude the paper and discuss the future work in Section 9.

2. Related Work

The area of malware detection on smartphones is an emerging issue. The related work that is mainly being focused and discussed falls into three areas: malware analysis and classification, malware family identification (categorization), and malware detection on smartphones.

2.1. Malware Analysis and Classification. Previous approaches [22–28] have been proposed for malware detection, which mainly use static and dynamic features or the combination of them. AndroidLeaks [29] is a static analysis framework for automatically finding potential leaks of sensitive information in Android applications. Chen [30] focused the work on detecting malicious behavior that can be characterized by the temporal order in which an application uses APIs and permissions. He constructed a permission event graph (PEG) with these static features. 152 malicious and 117 benign applications were used to evaluate the work. AppIntent [31] provides a sequence of GUI manipulations corresponding to the sequence of events that lead to the data transmission. With the help of AppIntent, researchers can determine whether the data transmission is malicious. Zhang et al. [32] introduced graph similarity metrics to detect zero-day malware. Over 2000 malware samples and 13,000 benign applications were used in their experiments. The experimental result shows that their signature-based method can achieve 93% accuracy on malware detection with a low false-negative rate. Wang et al. [16] explored the permission-induced risk in Android apps in a systematic manner. They evaluated the malware detection result with permission ranking on a very large data, and the results show a satisfying performance (the detection accuracy was 94.62% with a false-positive rate being 0.6%). Apposcopy [8] employs a combination of static taint analysis and intercomponent call graph (ICCG) to detect Android applications. Apposcopy is evaluated on a corpus of
real-world applications and achieves superior results. There are some related works that used dynamic features to analyze and detect malware. Yuan et al. [33] proposed VetDroid to reconstruct sensitive behaviors in Android apps through dynamic analysis from permission usage perspective. TaintDroid [11] is a dynamic taint tracking and analysis system with the use of the Android’s virtualized execution environment. DroidDolphin [34] proposed a dynamic malware analysis framework that extracts useful static and dynamic features dynamically. These methods largely rely on expert analysis to manually extract features and detection patterns through expert analysis. Unlike that, we present a systematic analysis of five state-of-the-art classification methods with the combination of conventional static features and n-opcode, which can automatically learn features from raw data directly.

2.2. Malware Family Identification. Some researchers [35–37] focus their attention on the identification of malware variants of specific malware families. A fact is that 5% to 13% of apps hosted on the studied marketplaces are repackaged according to the experiments performed by DroidMOSS [38], an application similarity measurement system that applies a fuzzy hashing technique to effectively detect the changes from app-repackaging behavior. ViewDroid [15] performs a user interface-based mobile application-repackaging detection from a higher level abstraction. Experimental results show that ViewDroid can effectively and efficiently detect repackaged applications with low false-positive and false-negative rates. Battista et al. [39] presented a model checking-based approach in detecting Android malware families. With the analysis and verification of Java bytecode, a preliminary investigation shows that it can classify OpFake and DroidKungFu families, while in our work, we consider 179 families with 5560 malware family samples. Monet [17] combined runtime behaviors with static structures to detect malware variants based on a fact that the runtime behaviors of malware’s core functionalities within a malware family are usually similar in a high degree. Arp [12] is the first one to use comprehensive static features (permissions, API calls, intent filters, etc.) for malware detection. He achieved a TPR of 94% at a low FPR of 1%. Suarez-Tangil [40] proposed a text mining-based approach to measure the similarity between malware samples, which is then used to automatically classify them into families. However, he used the MalGenome dataset [41] to evaluate the method, which is outdated and only has small subset of DREBIN that is used in our work. EC2 [42] performs Android malware families prediction through static and dynamic features with the ensemble of supervised and unsupervised classifiers.

The closest research studies to our work are presented by Kang et al. [19, 20], Shen et al. [21], and Arp et al. [12]. Several efforts [19–21] present n-opcode analysis-based approaches that utilize machine learning to categorize Android malware. Taking the work of Kang et al. as an example, experiments over 2000 samples show that a high accuracy (F1-score of 98%) is achievable. Different from Kang et al. [19], Arp et al. [12] only used conventional static features based on expert analysis to manually extract features, and the evaluation result with the SVM classifier presents a malware detection accuracy of 94%. Compared with these approaches, we use the combination of conventional static features and n-opcode, which can specifically describe the characterization of Android applications. Our work also provides systematic analysis of the combination features under a large scale dataset (184,486 benign applications and 21,306 malware samples) with five state-of-the-art classifiers and the ensemble of them and outperforms these approaches (the detection rate was 98.4% with a false-positive rate being 0.5%) and categorization (overall accuracy of 98.17% for all malware families: 99.37% for large families and 48.86% for small families).

2.3. Malware Detection on Smartphones. There are few approaches that aim to detect malware on mobile devices [43–47]. Taking [43] as an example, it presents a method for analysis and detection of malware utilizing static features like permissions and hardware and software features on 1377 malware and 1377 benign samples, which runs natively in the device to predict applications with 94.48% of accuracy and 36-millisecond time overhead.

Our work introduces n-opcode to learn features automatically from raw data to identify novel malware or malware variants. Meanwhile, in order to validate our approach systematically, we conduct several comparison experiments using five supervised classifiers under a large scale of samples. Furthermore, we integrate our work with Android OS to enable smartphones the ability to detect malware. In addition, we investigate and evaluate several methods on extracting features from APK files to further reduce the run-time overhead on smartphones.

3. Dataset

In order to present a realistic performance result that can reflect the real world, we leverage a large scale of app samples to evaluate our approach, consisting of benign applications and malware samples. As the only official app store, the Google Play store contains millions of applications uploaded by developers. However, it does not provide API for analyzers to download a large scale of free apps. Thus, collecting a big dataset of applications is a strict challenge. We collected benign and malicious applications from multiple sources: (i) PlayDrone [48], the first scalable Google Play store crawler that can scan and download applications from the app store. It also shared the whole Android APKs crawled from the Google Play store using PlayDrone online (https://archive.org/details/playdrone-apks). (ii) Anzhi [49], the biggest third-party app market in China. With the help of Wang et al. [16], we got free apps provided by Anzhi. (iii) AndroTotal [50], a publicly available tool (https://bitbucket.org/andrototal/tools) for detecting malware that is used to submit APKs for scanning and retrieve detecting results. It exposes a free-of-charge API to let researchers gain access to all the uploaded samples. (iv) DREBIN, the largest third-party malware dataset with labeled families.
The collected dataset is impure, and further work needs to be done to prepare reliable ground truth data. We utilized VirusTotal [51], a free online service that analyzes suspicious files and URLs to detect types of malware including viruses, worms, and Trojans. It uses different (currently 52) antivirus engines to scan for potential malware: if at least 10 antivirus engines recognize the app as malicious, we then treat it as a benign app to get a high-quality ground truth. Otherwise, the app is considered as malware. In total, we get 205,792 samples consisting of 184,486 benign applications and 21,306 malware samples. As 80/20 is a commonly used ratio for the size of the training set and test set, we split the whole dataset for the subsequent evaluations according to this ratio.

In this work, we utilize the whole dataset or the subset of it for all of our subsequent experiments including malware classification. As DREBIN is the largest labeled dataset of malware families that contains 179 malware families with 5560 samples, we select and analyze it for malware family categorization in our work. In addition, we consider malware families that have more than 15 samples as large families (considering families fewer than 15 samples as small). The number of samples of each malware family in DREBIN is unbalanced: Figure 1 presents the cumulative distribution of the size of malware families. There are 33 large families in total that account for almost 89% of samples in the dataset, which will largely affect the detection result. This suggests us to analyze and evaluate malware family categorization on large families, small families, and the whole dataset separately.

4. Feature Extraction

In this section, we introduce the conventional static features and n-opcode that are used in our work. Furthermore, we explain how to extract and select these features from Android application package files to optimize malware classification and categorization in our work. We first present some fundamentals about the Android environment.

4.1. Android Fundamentals. Android is an open-source mobile operating system, and users install applications for their smartphones with Android through the Google Play store, third-party marketplaces, and even unknown sources. Smartphones use APK files to install Android apps. Android applications are archive files (APK files) with an .apk suffix that contains all the contents like compiled codes (.dex suffix files) and all the data and resources used in the application. Meanwhile, each Android application must have a manifest file named "AndroidManifest.xml" in its APK file to provide essential information like the following:

(a) **Package name** serves as a unique identifier for the application

(b) **Components** compose the application including **Activities** that are used to interact with users, **Services** that are utilized to execute background tasks, **Broadcast Receivers** mainly for interprocess communications, and **Content Providers** that manage a shared set of apps

(c) **Permissions** required by the application to access correspondingly resources provided by devices

(d) **Hardware or software features** used by the application, like camera and Bluetooth services

Each application runs in its own Linux process in isolation from other apps (usually we call it the sandbox environment), and the Android system assigns a unique user ID to each application to protect data created by the app. Apps with the same user ID can run in the same process and share their data easily. In addition, in order to access device data such as the user’s contacts, camera, and Bluetooth, apps need to explicitly request permissions in their AndroidManifest.xml file and have to get explicitly granted these permissions by users.

4.2. Conventional Static Features. Several static features have been proposed in previous approaches [12, 16, 30–32]. We leverage some of them that are proposed in [12] to characterize the applications.

4.2.1. Requested Permissions. One of the most significant security mechanisms in Android is the permission system. With permissions granted at installation time, applications can access specific resources provided by devices. The previous work [16] has shown that malware tends to request certain permissions or the combination of permissions than benign applications. For instance, malicious malware often requests SEND_SMS permission and sends SMS to premium-rate numbers. Thus, we obtain all permissions listed in the manifest file.

4.2.2. Hardware Components. In order to access specific hardware resources (GPS, microphone, etc.), applications have to explicitly declare these features in the manifest file, which may have security or privacy issues. Applications that
have access to GPS are able to collect and leak location information of users. In addition, the combination of hardware components and requested permissions even can lead to a heavy hazard. Take the Internet permission and microphone as an example, assuming that an application has access to both of them when in a private business conversation, it can easily record the conversation and expose it through the Internet without the user’s consent.

4.2.3. API Calls. Certain API calls found in malicious applications represent that they are allowed to access sensitive resources of devices. Like getDeviceId(), a special API provided by Android can be used to obtain the International Mobile Equipment Identity (IMEI) of devices.

We ignore other static features like app components and filtered intents used in [12]; this is because the name of them can be easily obfuscated and they are not discriminative in malware classification as they introduce more noisy information [42].

4.3. n-Opcode Features. Android applications are generally developed in Java and then compiled and converted to optimized bytecode for the Dalvik virtual machine (Dalvik bytecode) (https://source.android.com/devices/tech/dalvik/dalvik-bytecode). Dalvik bytecode is the common executable format for Android applications and can be disassembled to provide information about process flows and API calls [12].

In order to elude detection, malicious Android applications often utilize sophisticated detection avoidance techniques like repacking and rename obfuscation, which mainly change the name of components, classes, and methods to make them unrecognized. As new malware variants tend to have the same core malicious codes as previous versions, the process flows and logic of the application may not be modified as they raise little suspiciousness in malware identification. Therefore, Dalvik bytecode of compiled applications can be used to distinguish malware from benign applications. We consider the concept of n-gram originated from Natural Language Processing [52] to represent the opcode sequences of applications. The extraction flow of n-gram of opcodes (n-opcode) is shown below in Figure 2.

In order to extract the conventional static features and n-opcode features from the manifest and the source code (actually smali files that are decompiled from the apk file), we need the Android application reverse engineering techniques. There are several optional tools that can be used to extract metadata from apk files and the manifest files. As we integrate our work with Android and run it on smartphones, detailed description is provided by Android and run it on smartphones; detailed description is presented in Section 8. Finally, we choose and leverage Baksmani (https://github.com/JesusFreke/smali) (an Android application assembler/disassembler) and zipfile (https://docs.python.org/2/library/zipfile.html) (a Python library for decompressing apk files) to extract features for further malware classification and categorization.

4.4. Feature Selection. In order to achieve the most optimized malware detection and categorization accuracy, we need to select the best combination of conventional static features and our n-opcode. As the conventional static features are decided, the only variable factor in our features is the value for n in n-opcode. Thus, we perform experiment and analysis with n-opcode of up to 9-opcode for both application classification and categorization on a subset of our real datasets utilizing five state-of-the-art classification algorithms: SVM, KNN, AdaBoost (Ada), random forest (RF), and gradient boosting machine (GBM). We utilize these five supervised classifiers in the subsequent experiments in this paper. There are several open-source libraries that have already implemented these classification algorithms, and we leverage SVM, RF, and AdaBoost classifier provided by scikit-learn (http://scikit-learn.org/stable/) and GBM classifier provided by the LightGBM framework (https://github.com/Microsoft/LightGBM). Then, we adopt the 5-fold cross-validation approach on each combination of conventional static features and n-opcode (n ranges from 1 to 9).

4.4.1. Malware Classification. We first evaluate the combination features for malware classification on a subset of our datasets consisting of 1963 malware and 1963 benign applications. Figure 3 shows the results of the evaluation.

As shown in Figure 3, the performances of these algorithms are similar except for KNN. The accuracy increases to the best as n equals 2, and SVM achieves the best accuracy of 97.06%, but the accuracy decreases when n is greater than 3. This may have a reason that more noisy information is introduced and affect the classification performance when the length of the opcode sequence (n-opcode) is increased. KNN shows the worst performance, but the increasing trend is also similar to others.

4.4.2. Malware Categorization. We then evaluate the combination features for malware family identification on the DREBIN dataset that contains 179 families with 5560 malware family samples. As the purpose in this section is to select features, we evaluate the whole dataset (large families and small families) without considering large and small families separately. The performance result is shown in Figure 4, which shows a similar trend with the result of malware classification in Figure 3. Each algorithm achieves its best performance when n equals 2 or 4.

Tables 1 and 2 show the top ten important n-opcode features for malware classification and malware family categorization, in which we omit the prefix “0x” of the opcode (e.g., using d8 represents 0xd8). And there is a correspondence (http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html) between opcode and the name of opcode in Dalvik. In Table 1 which shows top ten n-opcode features for malware classification, we found one 3-opcode (“390f6e”) and two 4-opcode (“6e0c086e” and “08546e0c”) which only appeared in malware samples from the study in [19] in our correspondingly selected 3-opcode and 4-opcode as 19999th, 4833th, and 13527th ranked features. Meanwhile, we also found that one 5-opcode
“0c1a6e0a39” from the study in [53] is in our selected 5-opcode as the 19992th ranked feature. Another observation is that one 4-opcode “0d076e12” is extension of the second-ranked 3-opcode “0d076e.” In Table 2, which presents top ten n-opcode features for malware categorization that was only previously researched in [19], only one 5-opcode “0c22702271” has been found in our selected 5-opcode as the 15563th ranked feature. Respectively, the similarity and difference in results among the studies in [19, 53] and our work are mainly according to the fact that different benign samples were used between these works, and the malicious datasets used in [19, 53] are just the subset of datasets used in our work.

Besides accuracy, time and memory overhead are other significant factors when choosing features for malware detection. The number of selected n-opcode features increased as n becomes larger [19], which means more time and memory space are needed when extracting n-opcode features. Thus, we select the combination of conventional static features and 2-opcode for our subsequent experiments, although KNN has the highest malware categorization accuracy when n equals 4. Through feature extraction, more than 2,000,000 features are extracted from each app. As the number of features is too large to be efficiently processed, we employ a linear SVM.
5. Methods

In this section, we introduce our methodology for malware classification and categorization. First, we consider five state-of-the-art supervised classifiers to evaluate the performance comparisons based on the combination of conventional static features and our $n$-opcode with multiple evaluation metrics. Second, we propose MobiSentry which utilizes the ensemble of these classifiers for classification and categorization separately.

### 5.1. Supervised Classifiers

We consider five off-the-shelf classifiers mentioned before in Section 4.4 for both malware classification and malware family identification: KNN, Ada, RF, GBM, and SVM. Beforehand, we adopt the $k$-fold cross-validation approach based on the grid search to find the optimal parameter settings. In total, we consider the combination of the five algorithms described above with two ensemble rules: majority voting and malware override.

#### Definition 1 (majority voting).
We consider a sample as the detection result which has the maximum frequency. For instance, if an application is detected as malware by 3 of the 5 classifiers, which means 2 of the 5 classifiers treat it as malware, then we regard malicious applications as 1 and let 0 represents benign applications in our subsequent experiments, which indicates that the optional parameter value is 0 or 1 in Step 7 and Step 8. If one detection result of a test sample achieves a maximum frequency, it is assigned to the malware class.

#### Definition 2 (malware override).
We treat a sample as a benign application only if the detection results of all the classifiers are benign. Otherwise, we consider this sample as malware.

As described in Algorithm 1, we train these five supervised classifiers on training dataset $S_1$ in the initial step (Step 1), while the other steps are largely dependent on the type of malware detection (Step 2), which can be divided into two categories: malware classification and categorization.

### 5.2. Ensemble of Classifiers

To further improve the accuracy of malware detection, we employ the ensemble of multiple classifiers to take the advantage of each algorithm. All the prediction results of each classifier will then vote to generate a final decision. In total, we consider the combination of the five algorithms described above with two ensemble rules: majority voting and malware override.

#### Table 2: Top ten $n$-opcode features for malware family categorization.

<table>
<thead>
<tr>
<th>2-Opcode</th>
<th>3-Opcode</th>
<th>4-Opcode</th>
<th>5-Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>12b1</td>
<td>6e0c11</td>
<td>0d72280d</td>
<td>1f6e1a6d8</td>
</tr>
<tr>
<td>0774</td>
<td>0d076e</td>
<td>d8545b5c</td>
<td>6e280d281a</td>
</tr>
<tr>
<td>b78e</td>
<td>1f3907</td>
<td>0d076e12</td>
<td>1c70156e6e</td>
</tr>
<tr>
<td>1768</td>
<td>221a54</td>
<td>0d6e0728</td>
<td>1a126e9c54</td>
</tr>
<tr>
<td>d281</td>
<td>69126e</td>
<td>62707222</td>
<td>0c710ce607</td>
</tr>
<tr>
<td>d81a</td>
<td>5b1a5b</td>
<td>136e0c1a</td>
<td>6e06e50e69</td>
</tr>
<tr>
<td>461f</td>
<td>0e1211</td>
<td>221a7054</td>
<td>62dd495012</td>
</tr>
<tr>
<td>0707</td>
<td>69541a</td>
<td>541a126e</td>
<td>0c28070707</td>
</tr>
<tr>
<td>6a1a</td>
<td>720b71</td>
<td>1c6e1c6e</td>
<td>0e52d85952</td>
</tr>
<tr>
<td>6174</td>
<td>710a81</td>
<td>0e70386e</td>
<td>12326e0c6e</td>
</tr>
</tbody>
</table>

#### Table 3: Number of feature sets.

<table>
<thead>
<tr>
<th>Feature type</th>
<th>Number of features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Requested permissions</td>
<td>98</td>
</tr>
<tr>
<td>2 Hardware components</td>
<td>30</td>
</tr>
<tr>
<td>3 API calls</td>
<td>17,212</td>
</tr>
<tr>
<td>4 $N$-gram</td>
<td>35,000</td>
</tr>
</tbody>
</table>

#### Table 4: Optimized parameters for each classifier.

<table>
<thead>
<tr>
<th>Classifier</th>
<th>Parameters</th>
<th>Range</th>
<th>Parameters that generate best results</th>
</tr>
</thead>
<tbody>
<tr>
<td>KNN</td>
<td>n_neighbors</td>
<td>(2, 20)</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>n_jobs</td>
<td>(1, 20)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>leaf_size</td>
<td>(10, 50)</td>
<td>30</td>
</tr>
<tr>
<td>Ada</td>
<td>n_neighbors</td>
<td>(1, 20)</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>splitter</td>
<td>(&quot;gini,&quot; &quot;entropy&quot;)</td>
<td>&quot;gini,&quot; &quot;random&quot;</td>
</tr>
<tr>
<td></td>
<td>estimator</td>
<td>(&quot;gini,&quot; &quot;entropy&quot;)</td>
<td>&quot;random&quot;</td>
</tr>
<tr>
<td>RF</td>
<td>max_depth</td>
<td>(10, 18)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>min_samples_leaf</td>
<td>(2, 6)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>criterion</td>
<td>(&quot;gini,&quot; &quot;entropy&quot;)</td>
<td>&quot;entropy&quot;</td>
</tr>
<tr>
<td>GBM</td>
<td>Num of leaves</td>
<td>(10, 35)</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>max_depth</td>
<td>(5, 13)</td>
<td>35</td>
</tr>
<tr>
<td>SVM</td>
<td>Kernel</td>
<td>(&quot;linear,&quot; &quot;rbf&quot;)</td>
<td>&quot;rbf&quot;</td>
</tr>
<tr>
<td></td>
<td>C</td>
<td>(2, 2)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>gamma</td>
<td>(2, 2)</td>
<td>2</td>
</tr>
</tbody>
</table>

5.3. Malware Classification. For malware classification, we then count the frequency of each unique malware detection result for each test sample (Step 4), and we regard malicious applications as 1 and let 0 represents benign applications in our subsequent experiments, which indicates that the optional parameter value is 0 or 1 in Step 7 and Step 8. If one detection result of a test sample achieves a maximum frequency, it is assigned to the malware class.

<table>
<thead>
<tr>
<th>Feature type</th>
<th>Number of features</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Requested permissions</td>
<td>98</td>
</tr>
<tr>
<td>2 Hardware components</td>
<td>30</td>
</tr>
<tr>
<td>3 API calls</td>
<td>17,212</td>
</tr>
<tr>
<td>4 $N$-gram</td>
<td>35,000</td>
</tr>
</tbody>
</table>
the corresponding result (malicious or benign) when using the majority voting rule (Step 13). Meanwhile, if all the detection results of a test sample are 0, it is considered to be benign when using the malware override rule (Step 16), which means all the classifiers identify this sample as a benign application. Otherwise, we identify this sample as malware (Step 18).

5.4. Malware Categorization. For malware categorization, we measure the frequency of each identified family for each test sample (Step 4), and if the test sample meets a maximum frequency (Step 6), then it can be assigned to the corresponding family (Step 21).

According to the definition of the malware override rule, it is clear that this rule mainly focuses on binary classification. Therefore, we use the two rules in the ensemble of classifiers for malware classification and only utilize the majority voting rule for malware family categorization.

6. Performance Comparison of Malware Classification

In this section, we present detailed comprehensive performance comparisons of malware classification. First, we evaluate the detection performance of the five selected supervised classifiers with multiple evaluation metrics. Then, we provide the performance of the ensemble of classifiers with the following algorithm:

Malware override and majority voting rules are described in Section 5. Meanwhile, we perform an experiment to compare the performance of our MobiSentry with that of some baseline methods.

6.1. Evaluation Metrics. As the evaluation metrics are not standardized and different approaches rely on different metrics [13], we use multiple metrics to evaluate our work compared to previous works that only use few metrics [12, 19, 53]. We consider true-positive rate (TPR), false-positive rate (FPR), and accuracy as the metrics (as in the following formulas) used in our experiments:

\[
\begin{align*}
\text{TPR} & = \frac{TP}{TP + FN}, \\
\text{FPR} & = \frac{FP}{FP + TN}, \\
\text{precision} & = \frac{TP}{TP + FP}, \\
\text{recall} & = \frac{TP}{TP + FN}.
\end{align*}
\]

where TP, FP, TN, and FN indicate true positive, false positive, true negative, and false negative separately. Meanwhile, we also utilize the receiver-operating characteristic (ROC) curve plot and the precision-recall-curve
In the experiment, MobiSentry employed manually crafted security rules for malware detection, and the accuracy of malware detection of MobiSentry is that Kirin negative rate of MobiSentry. The main reason for the higher performance (97.4% under the malware override) is labeled as malware by 8 engines.

Furthermore, we evaluate the performance of the ensemble of these classifiers with majority voting and malware override rules based on our combinational features. In contrast, we consider the following three methods as baselines (https://github.com/renbingfei/MobiSentry): (i) the best classifier (GBM) of the five basic classifiers in our experiment with the optimized parameters introduced before (Table 4 in Section 4); (ii) the first work that uses comprehensive static features proposed in DREBIN [12]. We employed static features used in DREBIN and utilized a linear support vector machine (SVM) for malware detection. As the parameters of SVM used in DREBIN are not specified, we kept the parameters as default values; and (iii) a highly cited Android malware detection method proposed in recent years [54]. We chose the “max” method with 5 hidden classes for the mixture of naïve Bayes model (MNB) used in [54]. The comparison result is depicted in Table 5.

Table 5 shows the performance comparisons of MobiSentry. Although DREBIN [12] has a better result compared to the method in [54], the accuracy of GBM classifier based on our combinational features provides the best performance among baselines with a higher false-positive rate (3%) than them. The ensemble of basic classifiers gains significant performance (97.4% under the majority voting rule and 98.4% under the malware override rule) and outperforms other benchmarks, even roughly 10% higher than the method in [54], with a lower false-positive rate.

We also compare MobiSentry with another lightweight research tool named “Kirin” [55] for Android malware detection under the same set of malware samples. Kirin reports only 1057 apps out of 2,000 malicious apps to be malware. This corresponds to a false-negative rate of 47%, which is quite high compared to the less than 1% false-negative rate of MobiSentry. The main reason for the higher accuracy of malware detection of MobiSentry is that Kirin uses manually crafted security rules for malware detection, which can be rendered largely ineffective due to the production of novel malware variants, while MobiSentry employed n-opcode to learn features automatically from raw data to identify novel malware or malware variants.

6.3. False-Positive Analysis. In the experiment, MobiSentry misses nearly 0.6% malware of the whole test samples when employing the ensemble rules. After manually analyzing these positives, we discover that the reason for mislabeled benign applications as malware mainly comes from the following three aspects:

(i) Mislabeled by VirusTotal: we leveraged VirusTotal to prepare the reliable ground truth data before evaluations after collecting the dataset, and then we relabeled the datasets according to the reports returned by VirusTotal. The reports contain false positives, which will make the dataset impure. Meanwhile, in order to have better ground truth data, we considered apps with a higher number of malicious labels in the reports as malware, which miss 12 malware. For instance, a malware with the package name com.getfreerecharge.mpaisafrerecharge is labeled as malware by 8 engines.

(ii) Affected by adware: as the adware is in a twilight area between benign application and malware, these kinds of apps are hardly determined according to their behavior. Thus, the existence of adware will affect the detection result [13].

(iii) Ensemble rules: the majority voting rule relies on the maximum detection result, while the malware override rule considers a sample as benign application only when all the results detected by the five basic classifiers are benign; this will cause a smaller value of true negative (TN) and, correspondingly, will achieve a larger FPR.

7. Performance Comparison of Malware Categorization

This section describes the performance comparisons of malware categorization. As introduced in Section 5, the malware override rule mainly focuses on binary classification. Therefore, we only utilize the majority voting rule for malware family categorization. The malware family categorization actually is a multiclass/multilabel classification, which is different from binary-class classification. We start by explaining the metrics used to evaluate the performance of the multiclass classification.

7.1. Evaluation Metrics. The balance of malware families in the dataset largely affects the multiclass categorization performance when using default metrics to evaluate it. For example, the previous work frequently leverages a metric F1-score calculated using a method provided by the scikit-learn library to measure the accuracy in binary classification and ignores a parameter average of the method even in multiclass classification. By default, the method returns the score for each class when the value of average is none, and the returned score may be highly inaccurate when the number of samples for each class is highly skewed. When setting the value of this parameter as “weighted,” the method will calculate F1-score taking label imbalance into account. As the dataset for our malware categorization is largely imbalanced (as introduced in the following section), we consider F1-score and precision with the parameter average
assigned to “weighted” to evaluate the performance of malware categorization.

7.2. Performance Comparisons and Analysis. As DREBIN is the largest labeled dataset of malware families that contains 179 malware families with 5560 samples, we consider the dataset of DREBIN to evaluate the performance in the malware categorization study. Like introduced in Section 3, the number of samples of each malware family in DREBIN is imbalanced, and we analyze and evaluate malware family categorization on large families (more than 15 samples), small families (less than 15 samples), and the whole dataset separately.

The result for performance comparisons of our five basic classifiers is shown in Figure 6. As shown in Figures 6(a) and 6(b), the performances of malware categorization using five basic classifiers on both large families and the whole dataset are greatly satisfied, especially on large families, and the GBM classifier achieves the best accuracy (95.26% for the whole dataset and 98.67% for large families). However, the accuracy of small families is relatively low (maximum 47.79% achieved by the Ada classifier), and even the worst accuracy achieved by the KNN classifier is less than 35%. Figure 6(d) shows the detection accuracy on large families, and we observe that a malware family named "Hamob" achieves the lowest accuracy, which is less than 70% and roughly 20% lower compared with other families. After analyzing a few samples belonging to Hamob, we find that this malware family is more a type of adware than malware. The most prominent characteristics of Hamob are pushing and showing advertisements to users. This makes it hard to distinguish it from other malware families.

Furthermore, we evaluate the performance of the ensemble of these classifiers with the majority voting rule based on our combinational features extracted from the DREBIN dataset. In addition, we consider the similar methods used in Section 6 as baselines: (i) the best classifier (GBM) of the five basic classifiers in this experiment of malware categorization on the whole dataset and large families; (ii) the best classifier (Ada) in the experiment of malware categorization on small families; (iii) the first work that uses comprehensive static features proposed in [12]; and (iv) a highly cited Android malware detection method proposed in recent years [54]. The result is shown in Table 6. The accuracy of GBM and Ada classifiers based on our combinational features provides the best performance among baselines, and the reason for outperforming [12] may be based on the fact that the work presented in [12] only uses the SVM classifier under conventional static features, and we leverage five classifiers to detect malware under the combination of n-opcode and conventional static features. The ensemble of basic classifiers gains significant performance and outperforms other benchmarks.

8. Run-Time Performance

In order to enable smartphones to extract features from APK files and to detect malware at the time of installation, we
integrate MobiSentry with Android OS. The most significant time consumption occurs when extracting features; thus, we firstly investigate and evaluate several methods on extracting features from APK files to further reduce the run-time overhead on smartphones. We mainly consider four frequently used approaches for reverse engineering of Android APKs: (i) Androguard [56], a tool written in python that is mainly used to play with Android files, such as APK, DEX, and ODEX files; (ii) ApkTool [57], a tool for reverse engineering Android APK files; (iii) Baksmali, an assembler/disassembler for the dex format files; and (iv) the combination of zipfile (a python library) and Baksmali, which we mentioned before in Section 5.

We have conducted two experiments to evaluate the time consumption for feature extraction by using different sizes of apk files and dex files in our preliminary work [18], and the result showed that the size of apk files has no direct influence on time consumption for feature extraction because of the existence of resource files in apk files, but time consumption can be directly affected by the size of the dex file. Under this premise, we conduct contrast experiments on these four approaches with 500 applications, and the result is shown in Figure 7.

Experimental results show that extracting features utilizing the combination of zipfile and Baksmali takes the least amount of time.

To analyze the run-time overhead of MobiSentry, we integrate it with Android OS and envelope the learned detection model from our large scale of dataset to the smartphone to enable it the ability of malware detection. Then, we evaluate the detection performance of it on both PC (Intel® Xeon(R) CPU E5620 @ 2.40 GHz × 16 and 132 GB of main memory) and smartphone (Xiaomi 4). In contrast, we utilize the five basic classifiers and the ensemble of them (MobiSentry) as usual. The results are presented in Figure 8 and Table 7. Figure 8 shows the time consumption of malware detection after the process of extracting features from a given application, and we observe that MobiSentry
is able to detect application within less than 1 second. Meanwhile, the ensemble of basic classifiers also achieves the best and satisfying detection accuracy. The limitation of the ensemble approach is that the learned model is larger than five basic classifiers, which is mainly because of the large scale of dataset (more than 200,000 samples) and selected features (more than 180,000). As the storage space on smartphones is largely increased by now, even up to 256 GB for some new Android smartphones like Samsung S8, the model size of the ensemble approach is acceptable. In addition, although there exists false positive using MobiSentry in the real world, the FPR is relatively low (about 0.6% introduced in Section 6), which indicates one false alarm will be informed to users when installing nearly 200 applications.

9. Conclusions

In this paper, we present a lightweight defense system for malware classification and categorization on smartphones. Besides conventional static features such as permissions and API calls, we systematically analyzed the N-gram features of operation codes (n-opcode), which can be used to deal with known obfuscation techniques like rename obfuscation. Next, we proposed a novel ensemble learning algorithm based on five state-of-the-art supervised classifiers to overcome the drawbacks of them and improve the performance of malware detection. Moreover, we firstly investigated and evaluated several methods on extracting features from APK files to further reduce the run-time overhead on smartphones. Finally, we thoroughly evaluated our proposed system under a large scale of dataset to present a more real-world performance.

Future work might focus on analyzing cluster algorithms to improve the detection accuracy of novel malware variants identification. Moreover, as we observed that some 4-opcode features are just extensions of several 3-opcode features, we may also investigate the feasibility of detecting malware with the combination of different lengths n-opcode features.

Data Availability

The codes and data used to support the findings of this study have been deposited in the GitHub repository (https://github.com/renbingfei/MobiSentry).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is based on our preliminary version of this study [18] and was supported in part by the National Natural Science Foundation of China under Grant no. U1536112, Special Funds for Public Industry Research Projects of State Administration of Grain of China under Grant no. 201513002, and National High-tech R&D Program of China (863 Program) under Grant no. 2013AA102301.

References


[57] Apktool: https://ibotpeaches.github.io/Apktool/.