Accepted Manuscript

Characterizing Android apps' behavior for effective detection of malapps at large scale

Xing Wang, Wei Wang, Yongzhong He, Jiqiang Liu, Zhen Han, Xiangliang Zhang

PII: S0167-739X(17)30772-0

DOI: <http://dx.doi.org/10.1016/j.future.2017.04.041>

Reference: FUTURE 3443

To appear in: *Future Generation Computer Systems*

Received date: 21 March 2016

Revised date: 4 April 2017

Accepted date: 25 April 2017

Please cite this article as: X. Wang, W. Wang, Y. He, J. Liu, Z. Han, X. Zhang, Characterizing Android apps' behavior for effective detection of malapps at large scale, *Future Generation Computer Systems* (2017), <http://dx.doi.org/10.1016/j.future.2017.04.041>

This is a PDF file of an unedited manuscript that has been accepted for publication. As a service to our customers we are providing this early version of the manuscript. The manuscript will undergo copyediting, typesetting, and review of the resulting proof before it is published in its final form. Please note that during the production process errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

Characterizing Android Apps' Behavior for Effective Detection of Malapps at Large Scale

Xing Wang^a, Wei Wang^{a,*}, Yongzhong He^a, Jiqiang Liu^a, Zhen Han^a,
Xiangliang Zhang^b

^a*Beijing Key Laboratory of Security and Privacy in Intelligent Transportation,
Beijing Jiaotong University, 3 Shangyuncun, Beijing 100044, China*

^b*Division of Computer, Electrical and Mathematical Sciences & Engineering,
King Abdullah University of Science and Technology (KAUST), Saudi Arabia*

Abstract

Android malicious applications (malapps) have surged and been sophisticated, posing a great threat to users. How to characterize, understand and detect Android malapps at a large scale is thus a big challenge. In this work, we are motivated to discover the discriminatory and persistent features extracted from Android APK files for automated malapp detection at a large scale. To achieve this goal, firstly we extract a very large number of features from each app and categorize the features into two groups, namely, app-specific features as well as platform-defined features. These feature sets will then be fed into four classifiers (i.e., Logistic Regression, linear SVM, Decision Tree and Random Forest) for the detection of malapps. Secondly, we evaluate the persistence of app-specific and platform-defined features on classification performance with two data sets collected in different time periods. Thirdly, we comprehensively analyze the relevant features selected by Logistic Regression classifier to identify the contributions of each feature set. We conduct extensive experiments on large real-world app sets consisting of 213,256 benign apps collected from six app markets, 4,363 benign apps from Google Play market, and 18,363 malapps. The

*Corresponding author

Email addresses: spark@bjtu.edu.cn (Xing Wang), wangwei1@bjtu.edu.cn, <http://infosec.bjtu.edu.cn/wangwei> (Wei Wang), yzhhe@bjtu.edu.cn (Yongzhong He), jqliu@bjtu.edu.cn (Jiqiang Liu), zhan@bjtu.edu.cn (Zhen Han), xiangliang.zhang@kaust.edu.sa (Xiangliang Zhang)

experimental results and our analysis give insights regarding what discriminatory features are most effective to characterize malapps for building an effective and efficient malapp detection system. With the selected discriminatory features, the Logistic Regression classifier yields the best true positive rate as 96% with a false positive rate as 0.06%.

Keywords: Android, malicious apps detection, feature comparison.

1. Introduction

Android platform dominates the smartphone operating system market, and has become the main attack target for malicious applications (malapps). According to the report from IDC [1], Android takes the first place again with 87.6 percent market share in the second quarter of 2016. Android provides a kind of online mechanism (i.e., app market) for delivering applications for Android-based devices. The official [2] and third-party app markets allow users to easily find, download and use a vast amount of the third-party applications (or apps). The number of available apps on Google Play has increased to over 2.6 million in December 2016 [3]. Meanwhile, many alternative markets [4, 5] have a considerable number of apps and their own users. However, at the same time, Android platform has become an increasingly attractive target for attackers. Malicious and vulnerable applications have been found on both the official and third-party markets. Moreover, new threats continuously pop up at an increasing rate. Kaspersky Labs reported that they found 175,442 new unique malicious programs that were run on Android in the first half of 2014, which is 18.3 percent more than in the entire year of 2013 [6].

This ever-growing malware threat has stimulated research into Android application security. Existing work mainly focused on (i) permission security model analysis [7, 8, 9], (ii) app vulnerability mitigation [10, 11], and (iii) malapp behavior analysis and detection [12, 13, 14, 15, 16, 17, 18, 19, 20] based on static or dynamic analysis.

As the number of apps and malapps as well as their variants explosively

increases in the market, it is crucial to precisely characterize the behavior of an app, so as to develop automated methods for Android malapp detection at a large scale. The main challenge of characterizing apps is threefold as follows. First, apps running on Android have distinct characteristics compared to traditional desktop software. One of Android apps' characteristics is that they consist of four types of interacted components. For instance, malapps on Android can get unauthorized access to protected resources by hijacking an existing exported component in a legitimate app without implementing their own, which is known as component hijacking attack [11]. Second, Android malapps is becoming increasingly sophisticated by leveraging legitimate apps and system vulnerabilities to evade detection systems. For example, Zhou and Jiang [21] indicated that 86% of malicious samples repackage other legitimate apps as a major propagation tactic. The common features shared by the original legitimate apps and repackaged malicious apps are largely overlapped, which may lead to false positives. Third, some unprotected data on smartphone such as sensor data can be exploited to steal confidential information. For instance, Xu *et al.* [22] demonstrated that it is possible to infer the PIN code user pressed on a touchscreen from the accelerometer data and the orientation sensor data. Michalevsky *et al.* [23] used the gyroscope to measure acoustic signals, which is able to eavesdrop speech without accessing to the microphone on the phone.

In order to better characterize the behavior of Android apps for malapp classification, in this work, we aim at discovering discriminatory and persistent features of apps. After extracting the 11 single-type feature sets from APK files, we categorize them into two groups, namely, app-specific features and platform-defined features, according to the generality and specificity of feature sets. Platform-defined features are inherently related to Android system, and thus can be extracted from all the Android apps. In contrast, app-specific features are defined by individual apps, and thus can only be extracted from a few specific apps that own this type of feature. Classification based the full feature set can achieve higher accuracy. However, its accuracy will degrade while detecting new data set. Platform-defined features are more inherent, and more

55 persistent than app-specific features. It is potential to detect novel malapps or even malapps based on zero day attack by using platform-defined features. By categorizing the features into app specific and platform-defined, we understand better how the features perform differently, how to select features in detection tasks and when to retrain the classification models.

60 Extensive experiments are conducted with different feature sets and feature groups. First, in order to study the discriminative power of each feature set, we feed them into four classifiers to compare the classification performance. Second, we evaluate the persistence of app-specific and platform-defined features with past and new data sets collected in different time periods. For each feature group, we train the classifier on the past data set and evaluate it on both the
65 past and the new data set. By comparing the detection accuracy, we find that the platform-defined features are more persistent than app-specific features in terms of keeping classification performance. Third, the composition of relevant features selected by Logistic Regression classifier is thoroughly analyzed to reveal the most useful features of each feature set for automated malapp analysis
70 and detection.

In summary, the contributions of our work are fourfold:

- We explore three types of new features and combine them with the features that have also been employed in existing work [24, 17, 9, 18] to characterize behaviors of Android apps. We then categorize these feature sets into
75 app-specific and platform-defined features. We propose to employ four classifiers, namely, Logistic Regression (LR), linear Support Vector Machine (SVM), Decision Tree (DT) and Random Forest (RF), and compare the discriminative power of different feature sets and the performance of different classifiers.
80
- We conduct experiments on data sets collected in different time periods. The empirical results reveal that platform-defined features are more persistent than app-specific features in terms of keeping classification performance.

- 85 • We analyze the composition of relevant features and discover the usage patterns of features in the malapps. These patterns help to understand the behaviors of malapps with the most suitable features for automated malapp detection.
- 90 • We conduct extensive experiments with a very large benign app sets (as many as 217,619) collected from six app markets and Google Play market and with 18,363 malapps collected in the wild¹. The experimental results demonstrate the effectiveness of our methods and models (a true positive rate as 96% with a low false positive rate as 0.06%).

The rest of this paper is organized as follows: Section 2 describes the feature sets and classifiers used in this study. The data set is introduced in Section 3. Section 4 provides a detailed evaluation and discussion. Section 6 concludes the paper.

2. Approach

The purpose of our study is to find discriminatory and persistent features to effectively classify malapps based on a variety of features which are directly extracted from APK files with static analysis techniques. Thus, we treat the malapp detection as a binary classification problem and propose a learning-based approach.

The framework of our methods is shown in Figure 1 consisting of four steps. First, we collect a large amount of apps from six app markets and malapps in the wild. Second, we extract as many features from the apps as possible (including three types of new features underlined in Figure 1), in order to characterize each app with a vector. Finally, we conduct comprehensive experiments including: (A) comparing the performance of different feature sets, (B) feature persistence analysis, (C) classifier comparison and (D) relevant feature composition analysis.

¹All the features of our app set as well as the source code of our methods are available in our website (http://infosec.bjtu.edu.cn/wangwei/?page_id=85).

Figure 1: Analysis steps of our method.

In this section, we firstly discuss what kind of features we extract, and then describe the machine learning classifiers we use in this work.

2.1. Features

2.1.1. Feature Set Description

The features we extracted can be categorized into 11 feature sets (abbreviated as FS) that can be described with the following 9 questions.

Q_1 : What are the names of the app (and of the modules in it)? **Component names** (FS_1). The majority of Android malapps are repackaged legitimate apps [21], in which the attackers insert the same malicious payload (usually in terms of components) into many different legitimate apps. We include component names as a feature set to capture the behavior of component reuse presented in both benign and malicious apps.

Q_2 : What resources does the app request to access to? **Requested permissions** (FS_2) and **hardware and software requirements** (FS_3). What resources the apps try to access to is very important for malapp detection. In Android system, **requested permissions** (FS_2) and **hardware and software**

requirements (FS_3) indicate the demands of the apps for system resources. Permission request patterns can characterize the apps' intents of resource accessing. In this paper, we use all the permissions defined by Android platform and the third-party apps.

Hardware and software requirements. Android apps signal their hardware and software requirements to devices in their manifest files with `<uses-feature>` elements. We thus extract the hardware and software feature descriptors defined in Android documents [25] as the third feature set.

Q_3 : *What does the app declare its functionality?* **Filtered intents** (FS_4). Android platform uses *intent* as a messaging object an app and the platform can send to another app's component for requesting an action or process. Malapps often declare with an intent filter to receive specific system events, e.g., `BOOT_COMPLETED`, for activating malicious activity. In this work, we extract all the intent filters in the manifest files of the samples as a feature set.

Q_4 : *What resources does the app actually access?* **Restricted API calls** (FS_5) and **used permissions** (FS_6). Requesting a permission does not mean that the app actually accesses to the corresponding resources. We scan the disassembled code of the app samples and record whether they invoke API calls protected by some permissions. Additionally, we use the API-permission mapping provided by *PScout* [26] to obtain the used permissions. Used permissions and restricted API calls reflect the resources an app actually access at different levels of granularity.

Q_5 : *Who develops the app?* **Certificate information** (FS_7). App developers must sign their APK files with a certificate, the private key of which is held by themselves. This certificate helps to distinguish a developer from others. Developer information such as the country, email address, organization, state or province, as well as the SHA-1 thumbprint, can be extracted from the certificate.

Q_6 : *With whom does the app communicate?* **URLs, IP addresses, file paths and numbers** (FS_8) in source codes. By matching with regular expression patterns, we collect all the URLs, IP addresses, file path strings, and

numbers (with more than three digits) in the disassembled code as a feature set. These strings may involve many malicious behaviors.

160 *Q₇: What kind of files does the app (i.e., APK file) carry? **Payload information** (FS_9).* *Payload* indicates the files inside the APK archive file. We include payload information as a feature set, since some malapps contain extra .apk files in the host apps that tricks users to install these malicious .apk files, and since malapp can change the file name extension from .apk or .dex into
165 .png, so as not to arouse suspicion.

*Q₈: Whether the app dynamically loads external executable files? **Code patterns** (FS_{10}).* In this feature set, we check whether an app dynamically loads .dex file or Linux native code, whether an app executes shell commands, whether an app use Java reflection techniques, and whether an app invokes
170 cryptographic functions, etc.

*Q₉: What suspicious operations does the app perform? — **Suspicious API calls** (FS_{11}).* Inspired by Drebin [17], we extract certain API calls that allow access to sensitive smartphone resources such as accessing device ID, sending and receiving SMS messages, which are frequently used by malapps.

175 Drebin [17] also collects abundant static features including permissions, components, filtered intents and many others. We extend the features suggested by Drebin with the additional three new feature sets and some refinement including certification information, payload information and code patterns (i.e., FS_7 , FS_9 and FS_{10}).

180 2.1.2. Feature Types

We categorize all the features into two types: **platform-defined features** and **app-specific features**, according to the generality and specificity of feature sets. Platform-defined features are inherent in Android platform, i.e., they can be applied to all Android apps. In contrast, app-specific features are spe-
185 cific to apps. For instance, permissions defined by Android system (e.g., the “INTERNET”) are platform-defined features because they are applicable to all apps. However, permissions defined by a third-party app are more specific. In

addition, in terms of quantity, the app-specific features grow with the increase of data set, while the amount of platform-defined features keeps stable instead.

As shown in Table 1, in our feature sets, platform-defined features contain 96 platform-defined permissions in FS_2 , 101 platform-defined intent action strings and 31 platform-defined intent category strings in FS_4 , and all the features in FS_3 , FS_5 , FS_6 , FS_9 , FS_{10} , FS_{11} . The app-specific features contain all the features in FS_1 , FS_7 , FS_8 , 62,916 third-party intent action strings, 6,524 third-party intent category strings as the main part of FS_4 and 7,900 permissions defined by third-party apps in FS_2 .

We define $Occur(f)$ to denote the number of occurrences of feature f in 166,365 benign samples. Then the Average Frequency of feature set FS_k is defined by

$$AverageFrequency(FS_k) = \frac{\sum_{f \in FS_k} Occur(f)}{|FS_k|}, \quad (1)$$

where $|FS_k|$ denotes the number of features in FS_k .

We also show the average frequency of features in 166,365 benign apps for each feature set. We see that app-specific features are much sparser than platform-defined features.

2.2. Classification Models

All the features extracted from an app are embedded into a high dimensional feature vector to represent the app's behaviors for further analysis. In total we have 2,374,340 features. However, each of feature vector typically has only around hundreds of non-zero values, as for the app-specific features, a specific feature only appears in a very few samples.

Formally, an app is represented as a feature vector $\mathbf{x} \in \mathbb{R}^d$, where $\mathbf{x} = (x_1, \dots, x_d)$ and d is the number of features, x_i is the value of the i th feature. The label of samples is denoted as $y \in \{-1, 1\}$, with $y = -1$ for benign apps and $y = 1$ for malapps. A vector representing an app is then fed into the classifiers to detect whether it is malicious or not. In this work, we employ four classifiers, that are Logistic Regression, linear SVM, Decision Tree and Random

Table 1: Description of the platform-defined and app-specific features.

Feature Type	Feature Set	# of Features	Average Frequency in 166,365 Benign Samples
App-specific Features	FS_1 Component Names	1,295,664	2.2
	FS_2 Permissions	7,900	27.4
	FS_4 Filtered Intents	69,440	4.7
	FS_7 Certification Information	200,060	1.6
	FS_8 URLs, IP Addresses, File Paths, Numbers	766,739	13.8
Platform-defined Features	FS_2 Permissions	96	13,806.0
	FS_3 Hardware Features	41	17,197.1
	FS_4 Filtered Intents	132	4,006.8
	FS_5 Restricted API calls	34,188	160.9
	FS_6 Used Permissions	96	17,242.4
	FS_9 Payload Information	60	21,421.1
	FS_{10} Code Patterns	5	64,266.8
	FS_{11} Suspicious API calls	15	66,824.9

Forest. These classifiers are all widely used for binary classification. Both the Logistic Regression and linear SVM generate simple and easy to interpret linear decision boundaries. The results of Decision Tree also can readily be explained. As an ensemble method, Random Forest often outperforms other non-ensemble classifiers, so it can be used as a benchmark.

Logistic Regression (LR): Logistic Regression is a type of probabilistic statistical model widely used for binary classification problems. Given the training sets of N apps $\{(\mathbf{x}^{(i)}, y^{(i)}), i = 1, \dots, N\}$, the probability distribution of the class label y given a feature vector $\mathbf{x}$ is modeled by logistic regression as follows:

$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T \cdot \mathbf{x}) \quad (2)$$

where $\mathbf{w} \in \mathbb{R}^d$ are parameters of the logistic regression model; and $\sigma(\cdot)$ is the sigmoid function that ensures $0 \leq \sigma(z) \leq 1$ by defining

$$\sigma(z) \triangleq [1 + \exp(-z)]^{-1}. \quad (3)$$

In experiments, we can threshold the output probability of Eq. 2 to induce the decision rule of logistic regression as

$$\hat{y}(x) = 1 \iff p(y = 1|\mathbf{x}) > t, \quad (4)$$

for predicting the label of the feature vector $\mathbf{x}$, where t is the threshold value.

Due to the large number of features in our problem, regularization is required to avoid over-fitting and generate sparse solution. In our experiment, ℓ_1 regularized logistic regression is adopted. Specifically, we use the LIBLINEAR [27] library in which the LR model is trained by solving the following unconstrained optimization problem

$$\min_{\mathbf{w}} \quad \|\mathbf{w}\|_1 + C \sum_{i=1}^N \log p(y_i|\mathbf{x}_i), \quad (5)$$

where $\|\cdot\|_1$ denotes the 1-norm. As we see, the first term in Eq. 5 penalizes large value of w_j in the weight vector $\mathbf{w}$, making some coefficients driven to zero, which leads to a sparse solution. A large number of the corresponding

features play no role for the prediction. However, the model becomes easy to
 240 understand and we can capture the most important features.

Support Vector Machine (SVM): SVM seeks to find the best linear
 hyperplane decision boundary that will separate the data into two sides, in our
 case, the benign apps class on one side and the malapps class on the other side.
 Among the infinite number of hyperplanes, SVM selects the one that results
 245 in the greatest margin between two classes as the optimal hyperplane. The
 problem of finding the optimal hyperplane can be formulated as a quadratic
 programming problem.

Briefly, the detection model in SVM corresponds to the direction of the
 hyperplane, i.e., a vector $\mathbf{w} \in \mathbb{R}^d$ that can be seen as the weight of features. More
 250 generally, by using a kernel function, SVM allows one to project the original
 features to a higher dimensional feature space. However, in our study, we use
 the linear kernel to stay in the original feature space in order to understand the
 practical meaning of the model straightforwardly.

Decision Tree (DT): Decision tree is a classification model defined by
 255 recursively partitioning the training data into a tree structure. In such a tree
 structure, nodes represent features, leaves represent class labels, and branches
 emanating from nodes to nodes or nodes to leaves represent conjunctions of
 features that generate the class labels. Inducing a decision tree is a multistage
 or sequential process. In the experiments, we firstly put all the training samples
 260 at the root node, and then partition the training set depending on the chosen
 feature at this node. These two steps are then executed recursively at the child
 nodes from the previous step with the partitioned training subsets. During this
 process, the training data set is gradually split into homogeneous subsets. Many
 specific decision tree algorithms have been proposed. We employ C4.5 [28] in
 265 this work.

Random Forest (RF): Random forest is a combined classifier consisting
 of a collection of decision trees where each tree is learned independently on a
 randomly selected subset of training data. A subset for training each decision
 tree is selected by randomly sampling from both features and objects. The final

270 classification will be done by voting within all the generated trees.

2.3. Feature Selection

Apart from ℓ_1 -regularization used in the LR models, we also employ three filter-based feature selection methods in order to potentially avoid overfitting of our model and thus improve the performance of learned models. Specifically, 275 we use Mutual Information (MuInfo), Chi-Squared Test (Chi2) and One-way Analysis of Variance (ANOVA) to find the most relevant and informative subsets of features. We introduce the three feature selection methods respectively after given the formal notation.

1) *Mutual Information*: Let X denote a feature variable and C be the class 280 variable. The relevance of X and C can be measured by mutual information of them as

$$I(X, C) = \sum_{x_i} \sum_{c_j} P(X = x_i, C = c_j) \times \log \frac{P(X = x_i, C = c_j)}{P(X = x_i)P(C = c_j)}, \quad (6)$$

where $P(C = c_j)$ is the frequency count of class C with value c_j , $P(X = x_i)$ is the frequency count of feature X with value x_i , and $P(X = x_i, C = c_j)$ is the frequency count of X with value x_i in class c_j . In this paper, the class 285 C has binary values, c_0 for *benign* apps and c_1 for *malicious* apps. $I(X, C)$ is nonnegative in $[0, 1]$. $I(X, C) = 0$ indicates no correlation, while $I(X, C) = 1$ means that C is completely inferable by knowing X .

2) *Chi-Squared Test*: By computing the chi-squared statistics between each feature X and class C , the chi-squared test measures divergence from the distribution expected if one assumes the feature occurrence is actually independent 290 of the class value. The chi-squared statistic is given by the equation:

$$\chi^2 = \sum_{x_i} \sum_{c_j} \frac{(x_{i,j} - \bar{t}_{i,j})^2}{\bar{t}_{i,j}}, \quad (7)$$

where $x_{i,j}$ is the number of instances with value x_i in class c_j , and $\bar{t}_{i,j}$ is the expected number of instances with the value x_i and class c_j . This statistic value

χ^2 can be used to indicate the correlation of X and C . The larger χ^2 shows a stronger correlation between them.

3) *One-way ANalysis Of VAriance (ANOVA)*: The purpose of One-way ANOVA is to test for significant differences between class means. In One-way ANOVA, the F-test is used to quantitatively determine whether class means are equal by computing the F-statistic as follows:

$$F = \frac{SS_b/(M-1)}{SS_w/(\sum_{c_j} N_j - M)} \quad (8)$$

where M is the number of classes, N_j is the sample number of class c_j , the between class variance $SS_b = \sum_{c_j} N_j (\bar{x}_j - \bar{x})^2$ is the sum of square variance among classes, where $\bar{x}_j$ and $\bar{x}$ are the class and the overall sample means, respectively. The within-class variance $SS_w = \sum_{c_j} \sum_{x_i} (x_{i,j} - \bar{x}_j)^2$ is the total within-class sum of square variance. Thus F can be used to determine whether the classes are actually different by the measured feature and provide useful information for feature selection.

3. Data Sets

We collect a very large data set in order to comprehensively evaluate our methods. The data sets consist of two parts, benign apps collected from six third-party app markets and malicious apps collected from different sources.

3.1. Benign Apps

For benign apps, we crawled six alternative app markets (i.e., AnZhi [4], AppChina [5], MyApp [29], LenovoMM [30], GFan [31], NDuo [32]) in China and downloaded 287,631 APK files from November 2013 through January 2014. All the APK files have been confirmed as benign by an online service named *VirusTotal* [33], which provides detection service with over 50 antivirus (AV) scan engines. We determine whether the APK is benign based on the individual AV scan report and a sample is labeled as benign only if all AV scanners identify it as benign. Consequently, after filtering out APKs which are identified as

malicious by one or more AV scanners, there are 166,365 benign apps left in our benign apps set, denoted as *benign_2014*.

3.2. Malicious Apps

In order to build an acceptable ground truth data set, we construct the malapp data set by selecting the relatively reliable data sources, as well as taking rigorous threshold values of positive results from VirusTotal. The malicious app sets in this work come from four sources — 1,260 samples from the *Android Malware Genome Project (AMGP)* [21], 3,417 malicious apps downloaded from the malware repository *VirusShare* [34], the *Drebin* data set with 5,560 applications published in [17], and 401 samples provided by two antivirus companies. We uploaded all these samples to VirusTotal and the output of more than 50 AV vendors are inspected. Then a large threshold (i.e., 20) for the number of positive results is set to decide whether to keep a sample within the malapp data set. Consequently, after removing the duplicate samples, we have 8701 malicious apps from 239 different families left in our malicious data set, denoted as *malapp_2014*. There are 21 families, each of which comprises more than 50 samples, in this set, and they constitute 82.9% (7,217 samples in total) of the whole malapp data set.

3.3. New Data Sets

In order to evaluate the persistence of different features on classification performance (Section 4.2.2), we crawl AnZhi app market again and download APK files from January to March, 2015 to form a new benign app set. For new malapps, we get 14,763 samples from *VirusShare*. After removing duplicated instances and labeling with *VirusTotal*, we have 46,891 benign apps and 9,662 malapps. These data sets are treated as *new* in this experiment, which are named as *benign_2015* and *malapp_2015*, respectively.

We also crawl 4,500 popular apps from the official Google Play market in 2016 and conduct experiments with this new data set. By labeling with *VirusTotal*, we at last get 4,363 new benign apps.

Table 2: Breakdown of feature sets for benign and malicious data sets

Feature Set	Total	Benign Apps	Malicious Apps
Component names	1,295,664	1,277,560	20,877
URLs,IP,file paths,numbers	766,739	750,299	22,822
Certificate information	200,060	190,884	9,060
Intent filters	69,572	68,237	1,863
Restricted API calls	34,188	5,597	480
Requested permissions	7,900	7,796	320
Used permissions	96	68	51
Payload information	60	60	59
Hardware features	41	39	22
Suspicious API calls	15	15	15
Code patterns	5	5	5
App-specific features	2,339,710	2,294,553	54,782
Platform-defined features	34,630	6,007	792
All features	2,374,340	2,300,560	55,574

4. Evaluation

350 4.1. Overview

In this section, firstly we evaluate the effectiveness of the classifiers based on 14 different feature sets, including 11 single-type feature sets and three combined feature sets listed in Table 2. Secondly we discuss the performance of four classifiers. Thirdly we analyze in depth the relevant features generated by
 355 LR model based on the full feature set for the sake of finding the feature usage patterns of malapps and correlating them with malapps' actual behaviors. Finally, we discuss the limitations of malapp detection with only the static features extracted from APK files by analyzing false positives and false negatives.

We perform a 5-fold cross validation in the experiments. Specifically, we
 360 combine the benign app set and malapp set and randomly split it into five subsets consisting of the same number of apps. For each subset $k \in \{1, \dots, 5\}$, we train on the four subsets and test on the remaining subset k .

The experiments are run on an HP ProLiant DL380 G7 Server with two quad-core 2.40 GHz Xeon processors and 64 GB memory. We employ LIB-
 365 SVM [35] for SVM and LIBLINEAR [27] ℓ_1 -regularized Logistic Regression with Python. For Random Forest and Decision Tree, we use the implementation of *scikit-learn* [36]. In addition, we use *GNU Parallel* [37] to run the experiments in parallel so as to reduce the overhead.

4.2. Feature Set Comparison

370 In this section, we compare the feature sets from two aspects: classification performance and feature persistence.

4.2.1. Performance Comparison

In this work, True Positive Rates (TPR), False Positive Rates (FPR) as well as F-score are employed to compare the performances of different classifiers
 375 based on 14 feature sets. Accuracy is employed to compare the persistence of platform-defined and app-specific features.

Figure 2: Performance comparison between 11 single-type feature sets and 3 combined feature sets. The 11 feature sets are arranged from left to right in ascending order of the size of each feature set. The last three are the combined feature sets, i.e., the platform-defined feature set, the app-specific feature set and the full feature set. Overall, the performance becomes better with more features.

F-score is defined as the harmonic mean of precision and recall:

$$F\text{-score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}, \quad (9)$$

where Precision is the proportion of True Positive (TP) to all the positive results, and Recall is also called True Positive Rate (TPR) defined as the proportion of TP in all the positive instances. The F-score for an ideal classifier should be close to 1, indicating that the Precision and Recall are both close to 1.

Accuracy is the proportion of true results (both TP and True Negatives) to all the instances.

We list the feature sets in Figure 2 from left to right in ascending order of the number of features in each feature set. Features in the first five feature sets and feature set FS_5 (restricted API calls) are all platform-defined features. FS_2 (requested permissions) and FS_4 (filtered intents) contain app-specific features

and platform-defined features. FS_7 , FS_8 , FS_1 are all app-specific features. The last three feature sets combine all platform-defined features, all app-specific features and all features, respectively.

In general, from Figure 2 we have three observations: (1) app-specific features is more effective than platform-defined features for the detection of malapps; (2) more features lead to higher TPRs; (3) combining platform-defined and app-specific features achieve the best classification performance.

As shown in Figure 2, by using FS_{10} (code patterns), all the four classifiers (LR, SVM, DT, RF) almost identify all the instances as benign apps. It indicates that techniques such as Java reflection, dynamically loading, and encryption represented in FS_{10} features, have been widely utilized by both benign and malicious apps, therefore it is insufficient for malapp classification based on only code patterns.

With the following three platform-defined feature sets, namely FS_{11} (suspicious API calls), FS_3 (hardware features) and FS_6 (used permissions), the TPRs are only between 48% and 61%. This indicates that it is very hard to discriminate malapps from large amount of benign apps with only a few platform-defined features.

With FS_9 (payload information), the performances of LR and SVM classifiers are very poor, and the DT classifier yields a TPR of about 80% with a FPR of 1%. The RF classifier yields the similar but with a lower FPR of 0.2%. It can be seen that FS_5 is as effective as FS_2 . In general, the RF classifier achieves a TPR of 82.5% with a FPR of 0.1% with FS_5 . FS_4 (filtered intents) reveal the actions that an app is able to perform (e.g., showing the user a location on a map). With this feature set, all the four classifier produce similar detection results, i.e., a TPR of 65%-69% with a FPR of 0.2%-1.6%.

Another three feature sets (FS_7 certificate information, FS_8 URLs, IP addresses, file paths and numbers, FS_1 component names) are all app-specific. Classifiers based on these feature sets significantly reduce the FPR (e.g., 0.11%-0.32% using LR) while increasing the TPR. However, the number of features in these feature sets grows to over 200,000.

All the platform-defined feature sets are combined together to form the *full platform-defined* feature set. Similarly, combining all the app-specific feature sets we form the *full app-specific* feature set. Furthermore, all the platform-defined and app-specific feature sets constitute the *full* feature set. These combined feature sets result in better performance. As result, with the full feature set, the LR classifier yields the best F-score at 97.4% with a FPR of 0.05% and a TPR of 96.01%.

The above findings show that, a large feature set is potential to capture different characteristics of a large set of apps. Classifier based on a single-type feature set cannot detect more malwares than the classifier based on the full feature set. Moreover, by analyzing the relevant features in LR classifier in Section 4.6, we identify that in each feature set there are features making contributions to the detection. As Android malapps are becoming increasingly sophisticated, we need to characterize their behaviors from different aspects.

4.2.2. Feature Persistence

In order to discover the persistence of different feature sets on the detection performance, we use two data sets collected at different time periods, i.e., *benign_2014* and *malapp_2014* as the past data set and the *benign_2015* and *malapp_2015* as the new data set (see Section 3). As shown in Figure 3, the samples in these two data sets are almost separated by the time they were developed. We then build LR classifiers in different scenarios and compare their performance. The classifiers are built in three different settings: (1) training and testing based on the apps in the past data set; (2) training on the past data set and testing the apps in the new data set; (3) training and testing based on the new data set. The comparison results are presented in Table 3.

As shown in Table 3, when we build classifiers based on the past data set to detect the apps that also selected from the same data set, the accuracies are up to about 99%, regardless of the features we choose. Then when we use the classifiers built on the past data set to detect the apps from the new data set, the accuracies significantly reduce. Specifically, the accuracy of the classifier based

Figure 3: The Number of Samples over Time.

on platform-defined features reduces from 98.78% to 89.43%. Furthermore, the performance of classifier based on app-specific features decreases more significantly, reducing the accuracy from 99.59% to 77.29%. The decrease of the performance suggests that the behaviors of apps are evolving over time. The relevant features selected by the LR classifier from the past data set are no longer relevant in the new data set.

It is an interesting finding that the accuracy reduced with time passing. Actually, we guess this is caused by different reasons, such as (1) attacker improved their techniques; (2) new functionalities are introduced; (3) new attack models come out. However, platform-defined features are more persistent as time elapsed than app-specific features. So users who want to capture novel malicious apps based on new attack model may give these features more weights. Or retrain the classifier with new data set.

In addition, as shown in Table 5, we also detect apps from the official Google Play markets with LR models trained by the past data set. The accuracies are

Table 3: Persistence of feature sets

Feature Set	Accuracy		
	past-past	past-new	new-new
app-specific	99.59%	77.29%	98.63%
platform-defined	98.78%	89.43%	98.44%
full feature set	99.75%	89.35%	99.34%

high because all the tested apps are benign ones. However, the accuracy of
 465 app-specific features based classifier reduces more than that of platform-defined
 and full feature based classifiers.

The platform-defined features are inherently more persistent than the app-specific features. Therefore, the platform-defined features can have more stable performance than app-specific features.

470 As a linear classification model, the LR classifier gives a weight for each feature depending on how well the feature can be used to detect malapps. The *relevant* features, which receive non-zero weight from LR classifier, correspond to the behaviors that can discriminate malapps from benign apps.

We compare the number of relevant features selected by the two LR classifiers
 475 built on the past as well as the new data set with the full feature set, denoted by LR2014Full and LR2015Full in Table 4. As shown in Table 4, there are a total of 1,321 features selected by both classifiers, in which 993 are app-specific features and 388 are platform-defined features. Though the absolute number of shared platform-defined features is less than that of shared app-specific features, we can see that 70.71% platform-defined relevant features of
 480 LR2014Full continue to be selected by LR2015Full, while this percentage is only 20% for app-specific relevant features. Consequently, this illustrates again that platform-defined features are more persistent than app-specific features.

Table 4: Intersection of relevant features of LR2014Full and LR2015Full

Classifier	Number of relevant features		
	Total	App-specific	Platform-defined
LR2014Full	5,443	4,965	478
LR2015Full	4,263	3,704	559
Intersection	1,321	993	388
Percentage	24.27%/30.99%	20%/26.81%	70.71%/60.47%

Table 5: Detection Results on Google Play Apps

Feature Set	app-specific	platform-defined	full feature set
Accuracy	98.07%	99.58%	99.85%

4.3. Performance of Classification Algorithms

Figure 4 shows the Receiver Operating Characteristic (ROC) curves for the four classifiers. It is seen that the DT classifier performs noticeably poorer than the other three classifiers. The LR, SVM and RF classifiers perform comparably, while LR is better with small FPR values. RF is *de facto* ensemble of a (large) number of DTs. RF thus outperforms DT as it corrects for DT's habit of overfitting to their training set. Although we have extracted millions of features, most of the features are zero, resulting a very sparse vector that represents an app. The linear SVM and LR are effective and efficient to process high dimensional sparse vectors, and thus they can achieve high detection accuracy.

When we only consider the range of small FPRs, LR is the best performing method, though it is a little poorer than RF method if we go beyond 0.4% of FPR or so. Moreover, the LR model is more interpretable than the other ones. Hence, in the following sections, we only give the results of LR for demonstrating the contribution of different feature sets (Section 4.6) and for understanding the

Figure 4: Mean ROC curves comparing the performance the Logistic Regression (LR), linear SVM (SVM), Random Forest (RF) and Decision Tree (DT) classifiers for detecting malapps with the full feature set.

limitation of our method (Section 4.7).

4.4. Feature Selection

In this section, in order to potentially avoid overfitting of our model and thus improve the detection accuracy, we firstly employed three filter-based feature selection methods, i.e., Mutual Information (MuInfo), Chi-Squared Test (Chi2) and one-way Analysis of Variance (ANOVA), to find the most relevant and informative subsets of features. We then train LR models with different numbers of selected top features, and thus compare their performances and the performance with full feature sets.

From Figure 5, we observe that the detection accuracy gradually increases as the number of selected features becomes bigger. Then the accuracy is beginning to level off when the number of features exceeds 12,000, and finally reaches the best, which is around 99.75% for ANOVA, when around 14,000 features are

Figure 5: The detection accuracy of 3 different methods when increasing the number of selected features used.

Figure 6: The F-score of 3 different methods when increasing the number of selected features used.

used for building the models. As shown in Figure 6, F-score is consistent with accuracy.

It is seen from Figure 5 that comparing with the full-feature-based LR model,

Figure 7: ROC curves for the full-feature-based model and models with only top-14000 features

the model with top-14000 features selected by ANOVA algorithm slightly improves the detection accuracy (see the dotted red lines in Figure 1 and 2) (i.e., improving the accuracy from 99.750% to 99.748%).

Figure 7 provides a comparison between the full-feature-based and top-14000-feature-based models in terms of ROC curves. From this figure, it can be seen that the full-feature-based model is showing a consistently higher true positive rate than other models.

4.5. Comparison with Related Work

We use most features suggested by the study of Drebin [17], and extend the feature sets with the additional three new feature sets, including the certification information, payload information and code patterns (i.e., FS_7 , FS_9 and FS_{10}), and some refinement of other feature sets. In addition, we apply these features on four different classifiers, among which the LR classifier based on the full feature set yields the best performance. In contrast, Drebin only use SVM for

Figure 8: Comparison with Drebin by ROC curves

Table 6: Comparison with Related Work

Approach	# of Benign Apps	# of Malapps	TPR	FPR
[38]	1500	238	87.39%	-
[18]	1,5000	732	90%	6.5%
[16]	135,792	12,158	93.5%	10%
[9]	310,926	4,868	94.62%	0.6%
[17]	123,453	5,560	94%	1%
Ours	166,365	8701	96%	0.06%

malapp detection.

530 We compare the classifiers based on the full feature set with Drebin using LR, SVM and RF algorithms in terms of ROC curves. As shown in Figure 8, after adding these three new feature sets, the classification performances of all these three classifiers (LR, SVM and RF) improve, which means that these three new feature sets make their contributions to the detection, without respect to
535 the algorithms used.

We also compare our work with several notable related work, including DroidMat [38], MAST [18], Gascon [16] and the work in [9]. As shown in Table 6, our result is the best among these work regarding TRP and FPR.

Table 7: Breakdown of relevant features in LR

Feature Set	Total	Relevant	Benign	Malicious
Component names	1,295,664	1,525	303	1,222
URL,IP,file path,number	766,739	2,765	957	1,808
Certificate information	200,060	412	103	309
Intent filters	69,572	215	91	124
Restricted API calls	34,188	292	150	142
Requested permissions	7,900	152	73	79
Used permissions	96	40	22	18
Payload information	60	7	3	4
Hardware features	41	16	10	6
Suspicious API calls	15	14	7	7
Code patterns	5	5	3	2
App-specific features	2,339,710	4,965	1,460	3,505
Platform-defined features	34,630	478	262	216
All features	2,374,340	5,443	1,722	3,721

4.6. Composition of Relevant Features

540 A linear model such as LR yields a parameter vector w . As the weights of features, w is depended on how well a feature can be used to distinguish the benign and malicious apps. Each weight in w can be a positive or negative value. The positive values shift the sample to the malicious side, conversely, the negative ones shift the sample to the benign side. Hence, the features with positive weights or negative weights are treated as “malicious features”
545 or “benign features”, respectively. In addition, ℓ_1 -norm regularization makes w a *sparse* vector. It implies that most of the weights in w are zeros. The corresponding features of these zeros are useless in the classification, thus can be ignored in analysis. With a sparse w , we are selecting only a few *relevant*
550 features (i.e., features with non-zero weights) to characterize the apps.

In the following paragraphs, the LR model based on the full feature set trained on one of the five splits of our data set is analyzed. Although the total

number of features in the full feature set is over 2,000,000, there are only 5,443 non-zero features. In details, 1,722 features are benign, while 3,721 features are malicious. We list the relevant features in different feature sets in Table 7, and analyze them below in order to reveal their contributions to the detection.

The feature set *URLs, IP addresses, file paths, and numbers* counts for the most relevant features (50.8%). In the relevant strings, the largest portion are 1,183 HTTP/HTTPS addresses, including 789 malicious ones. The addresses shared by malapps include the following subset: (1) SMS subscription sites (e.g., “http://smsbankomat.ru” for *SMSreg* family); (2) command-and-control (C&C) servers (e.g., “http://svr.xmstsv.com/Notice/” for *jSMShider* family); (3) servers for saving collected user information; (4) Web Service API over HTTP used by both benign apps and malapps, such as social network API, Google Web Search API or Map API; (5) other kinds of HTTP address used by apps.

The feature set *component names* contributes as many as 1,525 relevant features (28%) consisting of 303 benign and 1,222 malicious ones. The top one malicious feature by weight is a Service component named “com.airpush.ad.Update-Check” appearing in 24 variants from *DroidKungFu* family. It masquerades as a legitimate ad library component. However, the component passes the device ID and app’s package name to a function implemented in a native library which dynamically generates a randomly named malicious executable file. Among the selected 1,222 malicious component names, 1,054 (86.3%) ones only appear in the malapps data set, showing that the classifier can effectively identify the malicious component shared by malapps for detection.

Certificate information also contributes a lot to the relevant features (7.57%). We find two phenomenons about the usage of certificate. First, some developers sign their multiple apps with the same key. In the malapps, several family use the same key to sign lots of variants in batches, such as the *FakeInstall*, *Boxer*, *Opfake* families. The LR classifier automatically selects these certificates commonly used by benign apps or malapps as benign or malicious features. Another phenomenon is the widespread use of publicly available keys, for example the

AOSP² keys and Eclipse debug keys. Because the classifier treats these keys
 585 as benign features, resulting in possible false positives. So we suggest that the
 developers should never use publicly available keys for their apps.

In the selected *filtered intents*, the top 10 malicious third-party app-defined
 Intent Action Strings are only used by specific malapps. It means that the
 model can automatically select relevant malicious features that the experts
 590 cannot know in advance. Besides creating their own Intents to trigger an ac-
 tion, malapps also register for listening specific system-defined actions. The
 top-5 malicious system intents selected by the classifier are “REBOOT”, “IN-
 PUT_METHOD_CHANGED”, “SIG_STR”, “SIM_FULL” and “BATTERY_CH-
 ANGED_ACTION”.

There are no relevant restricted API calls used by only malapps. How-
 595 ever, the LR classifier treats a number of API calls as malicious features. Be-
 cause these API calls reflect the malicious attempts of the malapps. These
 API calls can be used for (1) gathering user and device information such as
`getSubscriberId()`, `getAllVisitedUrls()`, `getAllBookmarks()`, (2) send-
 600 ing SMS message and (3) getting user location (e.g., `requestLocationUpdates()`).

The top 10 most commonly *requested permissions* selected by the classifier
 are listed in Table 8 and Table 9, respectively. We can see that “READ_PHONE_-
 STATE” is the most requested malicious features. “SEND_SMS” and “RE-
 CEIVE_SMS” are the two malicious permissions which distinguish malapps from
 605 benign apps in terms of frequency of usage.

For the *used permissions*, the LR classifier selects 40 out of 96 features,
 including 22 benign features and 18 malicious ones. The top 10 malicious mostly
 used by malapps can be classified into 3 categories: (1) permissions used for
 stealing user or system information; (2) permissions used for changing system
 610 state; (3) permissions used for sending SMS message; and (4) permissions used
 for restarting or stopping other apps.

²AOSP stands for Android Open Source Project, an open source project led by Google for
 offering the Android software stack. See <https://source.android.com/>

Table 8: Top-10 most commonly requested malicious permissions and their occurrence percentage in benign apps and malapps

	Permission	% in benign apps	% in malapps
1	READ_PHONE_STATE	64.52%	90.36%
2	ACCESS_WIFI_STATE	52.40%	35.43%
3	ACCESS_FINE_LOCATION	33.24%	25.62%
4	WAKE_LOCK	30.70%	43.11%
5	RECEIVE_BOOT_COMPLETED	18.37%	41.24%
6	READ_LOGS	16.97%	8.22%
7	INSTALL_SHORTCUT	11.65%	24.35%
8	SYSTEM_ALERT_WINDOW	12.24%	7.70%
9	SEND_SMS	8.05%	63.54%
10	RECEIVE_SMS	6.01%	51.18%

Table 9: Top-10 most commonly requested benign permissions and their occurrence percentage in benign apps and malapps

	Permission	% in benign apps	% in malapps
1	INTERNET	81.53%	96.99%
2	ACCESS_NETWORK_STATE	76.38%	55.64%
3	WRITE_EXTERNAL_STORAGE	75.33%	69.75%
4	VIBRATE	34.63%	24.62%
5	MOUNT_UNMOUNT_FILESYSTEMS	27.39%	3.74%
6	CHANGE_WIFI_STATE	26.00%	14.15%
7	GET_TASKS	22.36%	11.30%
8	CALL_PHONE	15.34%	11.99%
9	WRITE_SETTINGS	14.48%	13.21%
10	CAMERA	13.73%	3.37%

Table 10: Relevant Features of Payload Information, Hardware Features, Suspicious API calls and Code Patterns feature sets

FS	Benign Features	Malicious Features
<i>Payload Information</i>	# of .PNG file	# of PE executable file
	# of .APK file	# of non-type file
	# of .DEX file	# of ELF executable file mismatch of .APK file
<i>Code Patterns</i>	Loading DEX files	Reflection Used
	Loading Native Code	Execute Shell Command
	Cryptographic Functions	
<i>Hardware Features</i>	live.wallpaper	location
	touchscreen	microphone
	screen.landscape	camera.autofocus
	screen.portrait	sensor.light
	touchscreen.multitouch	telephony
	location.gps	touchscreen.multitouch.distinct
	camera.front	
	bluetooth	
	location.network	
	wifi	
<i>Suspicious API calls</i>	printStackTrace ()	Cipher-related
	getSystemService ()	HTTPPost-related
	Read/Write External Storage	Read-SMS-related
	getPackageInfo ()	getSubscriberId ()
	setWifiEnabled ()	getDeviceId ()
	execHttpRequest	getSimCountryIso ()
	getWifiState ()	Send-SMS-related

The relevant features selected by the classifier among the last four feature sets, namely *payload information*, *hardware features*, *suspicious API calls* and *code patterns* feature sets, are listed in Table 10. For the *payload information* feature set, having an APK file but with no ‘.apk’ extension is treated by the LR classifier as the most malicious feature. This is often used by malapps to hide their malicious payload. Also, having ELF executable files in the app is treated as malicious.

Overall, app-specific features dominate the relevant features in terms of number of features due to the inherent characteristics of these two feature sets. However, the discriminatory features from both app-specific and platform-defined feature sets can be automatically selected by the classifier. Furthermore, the selected relevant features reflect the difference between benign apps and the malapps according to the above analysis, which demonstrates the effectiveness of the classifier.

4.7. Limitations

4.7.1. Effect of Obfuscated Malapps

We are motivated to provide a general framework to detect various Android malapps. Although it is not specifically designed to detect zero-day attacks or obfuscated malapps, our framework is at the same level of the state-of-art tools in detecting obfuscated malapps, and better than those in detecting zero-day malapps. We have checked our data sets and found many obfuscated malapps. The reason is that Android malapps are always heavily repackaged and obfuscated. Thus, even though we did not purposely select obfuscated ones, the data sets contain a large percentage of those.

Specifically, out of 8701 apps in the data set `malapp_2014`, 8,692 apps use identifier renaming to obfuscate their class names or method names. 4,109 apps use reflection technique to hide their API usage. 2,367 apps use cryptographic APIs thus they have possibility to encrypt constant strings in their code such as C&C commands or server addresses. The `DroidKungFu` and `AnserverBot` are aggressively obfuscated. But the LR classifier detects 692 out of 696 variants

of DroidKungFu, as well as all the AnserverBot variants in our experiment. The fact that our test results are not downgraded by the presence of a lot of obfuscated malapps shows that our framework can resist common obfuscation in the detection.

4.7.2. Effect of Errors in Ground Truth

Errors in ground truth affect estimates of classification accuracy. Carlotto [39] developed an error model and proved that the accuracy of binary classifier A is:

$$\gamma = \rho\alpha + (1 - \rho)(1 - \alpha), \quad (10)$$

where ρ is the accuracy of the “ground truth” and α is the true accuracy of classifier A which is measured against the real ground truth. For a classifier with true accuracy $\alpha = 0.99$, its relative accuracy will be $\gamma = 0.98\rho + 0.01$. It is a linear function of ρ .

The proposed work rely on VirusTotal to build ground truth data sets. However, VirusTotal suffers from its own errors.

To evaluate this, we request VirusTotal to completely re-scan our data set including all the 8,701 malicious and 166,365 benign samples with the latest version of AV engines from 10 December to 25 December 2016. According to the criterion mentioned earlier, the 8,701 malicious samples can be labeled as malicious as before, because more than 20 AV engines still give positive results to each of them. However, for the 166,365 benign samples, the scan results change. In fact, after about two years, there are only 44,551 samples (about 26.78%) that trigger no positive signals. 25.47% samples are detected as malicious or potentially harmful by one AV engine from VirusTotal. Further, 5.90% samples (9,809) are labeled as malicious by more than 20 AV engines from VirusTotal. Among the 9,809 samples, 68.27% samples are categorized as AdWare. Furthermore, except for the Adwo family, the other nine families in the top 10 families do not appear in the top 21 families of our malapp data set. In

other words, the samples of these families can be treated as zero-day malwares
 670 for the classifiers trained based on the original data sets.

4.7.3. *FP and FN Analysis*

Since there are nearly 6% wrongly labeled samples in the original benign
 data set, we may, to some extent, underestimate false negatives in practice, and
 overestimate the false positives. Thus, we carefully investigate in detail the FNs
 675 and FPs the LR classifier produces in the first round of a 5-fold cross-validation.

When the LR classifier successfully detects 96.58% malapps (59 malapps
 undetected), simultaneously it produces 0.1% False Positives (33 benign apps
 falsely reported). In these 33 false positives, 20 samples are in fact malicious
 ones that trigger more than 20 positive results from latest AV engines. These
 680 20 malicious samples include 6 **Adwo** variants, 4 **AirPush**, 2 **Ksapp**, 3 **Kuguo** and
 5 **Waps** variants. **Ksapp** is a kind of trojan, and other 4 families are categorized
 as AdWare (shown in Table 11).

The reasons for other 13 false positive are (1) the original legitimate apps
 of some repackaged malapps; (2) using the same certificate as that of numerous
 685 malapps; (3) using app automatic creation tools used by malapps; (4) containing
 APK file as payload but without **.apk** extension name.

The false negatives can be divided into the following categories: (1) malapps
 using anti-decompiling techniques to prevent feature extraction; (2) malapps
 that rely on native code to realize their malicious purpose; (3) malicious com-
 690 ponents repackaged in a feature-rich legitimate app; (4) simple malapps that
 just send premium SMS or just send collected contact information to remote
 server.

4.7.4. *Evolving Behaviors of Benign and Malicious Apps*

Evolving behaviors of both benign and malicious apps may lead to overfit-
 695 ting of the models and performance reduction. In order to verify the absence
 of overfitting, in this section, we construct a completely separate data set in
 which the samples are distinct from those in the previous data sets we use (i.e.,

Table 11: 20 new found malicious samples

Digest	Type/Family	Package Name	Market
8b4846	AdWare/adwo	com.finger2finger.games.motobikegrassland.lite	NDuoa
d43511	AdWare/adwo	com.shuqi.paid.controller	AnZhi
9f5ef0	AdWare/adwo	com.ilovemdev.android.jinpinmeichatuban	Lenovo
35de20	AdWare/adwo	com.mmc.mohuan29	Nduoa
91892f	AdWare/adwo	cn.hongxiu.singlebook288328	AnZhi
3d8317	AdWare/adwo	com.droidhen.game.ciittyjumper	Lenovo
590922	AdWare/airpush	com.som4tress	Nduoa
5f5ada	AdWare/airpush	com.ifeng.xiaohua	AnZhi
10f14e	AdWare/airpush	com.gamefang.lwp.LSweetBaby	AnZhi
6d8ea1	AdWare/airpush	com.tim.apps.taiji	AnZhi
535713	Trojan/ksapp	baltorogames.kartmaniac	Lenovo
7ec2cf	Trojan/ksapp	com.it.cxy.xln	gFan
f41b35	AdWare/kuguo	com.Zeeplox.PoppingStars	gFan
454e84	AdWare/kuguo	com.fengle.game.addencircle	AnZhi
5c4af5	AdWare/kuguo	com.android.mmreader602	gFan
bc1ed9	AdWare/waps	com.dangerb.pedometer	Nduoa
0fd093	AdWare/waps	net.rjt.livejpmv	gFan
8e7aff	AdWare/waps	com.androidemu.harvemariosc	AppChina
04b539	AdWare/waps	org.youhu.lyyou	AnZhi
a0f875	AdWare/waps	com.supergame.game.zhizhupaiGM	Lenovo

Table 12: Accuracy comparison of different set-ups

	Train14-Test15	Train15-Test16	Train14-Test16
with full features	89.35%	88.47%	55.74%
with platform-defined features	89.43%	97.02%	63.58%
with top14000 features	89.23%	84.16%	53.46%

benign_2014, *malapp_2014*, *benign_2015*, *malapp_2015*) without overlapping in time. In this data set, the benign samples were downloaded from AnZhi app market from January 2016 to February 2016, and the malapps were collected from various sources in 2016. All the samples in this data set are labelled as benign or malicious according to the same rules as before by checking the output of *VirusTotal* scan reports of these samples and applying a majority voting. Finally, we obtain 7,931 benign apps and 9,854 malicious ones to construct the separate data set named as *dataset_2016*.

With this new data set, we conducted additional experiments as follows:

- **Train14-Test15:** Train LR models based on *benign_2014* and *malapp_2014* data sets. Then test the samples in *benign_2015* and *malapp_2015* data sets.
- **Train14-Test16:** Train LR modes based on *benign_2014* and *malapp_2014* data sets. Then test the samples in *dataset_2016*.
- **Train15-Test16:** Train LR modes based on *benign_2015* and *malapp_2015* data sets. Then test the samples in *dataset_2016*.

The above set-ups are repeated three times by combining with different feature sets, namely, the full-features, the platform-defined features, and the top-14000 features selected with ANOVA algorithm. Thus we have 9 experiments with different set-ups and the corresponding detection accuracy are shown in Table 12.

For simplicity, let $cell_{x,y}$ denote the cell of row x and column y in Table 12. For instance, $cell_{2,2}$ stands for the cell of row 2, column 3. The value in $cell_{2,2}$, 89.35%, is the full-feature-based models accuracy on data set 2015 while training the model on data set 2014.

From $cell_{2,2}$ and $cell_{2,3}$ in Table 12, it can be seen that, while using the full-feature-based model trained with the data collected one year earlier to detect the up-to-dated apps, the model keeps an accuracy being over 88%, which indicates that the full-feature-based model has a degree of generability while detecting

the unknown app sets. Additionally, it is worth noting that in these set-ups, the platform-feature-based models hold on a slight and steady advantage over the full-feature-based models, due to the persistence of platform-defined features over time. The top14000-feature-based models slightly underperform, which indicates that this kind of feature selection method does not automatically select persistent features like platform-defined features. Defining and demonstrating the persistence of platform-defined features is thus one of our major contributions, as these features are not difficult to extract but effective for vetting apps.

It can also be seen from Table 12 that with a model trained on old data sets, the detection accuracy of new app samples significantly decreases, especially the detection accuracy of *dataset_2016* with models trained on data set of 2014 (see the forth column of Table 12). This is because both the benign/malicious apps and the Android platform itself evolve over time. Some behaviors of apps or functions of the platform change or even disappear.

In general, an ideal behavior-based detection model is dynamic in characterizing apps behaviors. Thus, one needs to periodically update the model to keep up the rapid pace of evolving platform and the malware threats.

4.8. Overhead Evaluation

In this section, we conduct experiments to measure the overhead of feature extraction and classification. We randomly select 10,000 app samples from benign and from malicious apps, respectively. The DEX file size distribution of the selected apps is shown in Figure 9. We record the feature extraction time of these samples and the results are shown in Figure 10. In general, the time used for feature extraction goes up with the size of DEX file, but the overhead is acceptable. After building classification models, the classification time for an app can be ignored (less than 0.1 seconds with SVM and LR models). The total time required to extract features and classification for an app is around 40 seconds. That means with the mainstream hardware settings, our methods are scalable to millions of apps.

Figure 9: DEX File Size Distribution of 10000 Randomly Selected Samples.

Figure 10: Feature Extraction Time in seconds for 10,000 randomly selected samples.

5. Related Work

The related work can be summarized as the features and the methods used in malapp detection.

5.1. Features Used for Android Malapp Detection

App meta-data, static features and run-time information are commonly used in machine learning based Android malapp detection.

Chia *et al.* [40] focused on the effectiveness of app meta-data used as risk signals for the detection of privacy leakage. It is indicated that app ratings are not reliable signals because the ratings are based on app functionality rather than

on privacy risks. In WHYPER [41], by examining the app description with the technique of Natural Language Processing (NLP), Pandita *et al.* attempted to identify sentences in the description as indications regarding why the app needs a permission. Similarly, Gorla *et al.* [42] clustered apps by their descriptions to
 770 verify whether an app's description matched its functionality. After clustering, the outliers in each cluster could indicate potential malapps.

However, app meta-data are not always available. For example, there are no user ratings and reviews for a newly submitted app. Moreover, these information are market-specific, making the detection system tied to the specific market.

775 Permissions are widely used to detect malapps. Frank *et al.* [43] used permission request patterns and build probabilistic models to evaluate the quality of apps. Sarma *et al.* [14] compared the requested permissions of an app with the permissions requested by other apps in the same categories to determine its risk. Peng *et al.* [13] ranked permission risks with Bayesian models. In our
 780 previous work [9], we ranked all the individual permissions with respect to their potential risk and identified subsets of risky permissions with three methods. The identified subsets of risky permissions were then used to build classifiers for malapp detection.

Besides permission, other static features were also explored. In [44, 45], the
 785 authors used permissions as well as API calls for malapp detection. Canfora *et al.* [46] investigated the effectiveness of n-gram opcodes for Android malapp detection. In DroidMat [38], Wu *et al.* took into consideration static information including permissions, intent messages and deployment of components. DenDroid [47] used code structures as features and converted them into text se-
 790 quences to classify malapp families. In MAST [18] and Drebin [17], the authors used even more static features (e.g., used permissions, hardware components and network addresses) to detect malapps at a large scale.

In Canfora *et al.*'s work, 2-grams of opcode achieve the best detection accuracy and they can be considered as a kind of general feature. General features,
 795 or platform-defined features, have been widely used for the detection of malapps. In this work, we categorize all the features into two types, platform-defined and

app-specific. The platform-defined features are inherently related to Android system, and thus can be extracted from all the Android apps. In contrast, app-specific features are defined by individual apps, and thus can only be extracted from a few specific apps that own this type of feature. In this work, we define platform-defined and app-specific features and investigate the detection performance based on these two types of features.

In addition, our work extends the previous efforts by exploring as many as 11 types of static features and testing them on larger data sets. Additionally, we focus on comparing the discriminative power and persistence of different features. We also analyze the composition of relevant features selected by LR classifier to discover the contribution of different features to the detection.

By running and monitoring the apps, apps' run-time information such as CPU usage, memory usage, network connectivity and the interaction between an app and Android system or the users or other apps can be captured as features to profile the apps [48, 49, 12, 50, 51, 52].

In Andromaly [12], the authors realized a *host-based* Android malware detection system. By continuously monitoring the activities of the whole device and extracting features (such as CPU or memory consumption, number of running processes, etc.) that describe the device's state to feed into a supervised anomaly detector, the system can thus identify suspicious and abnormal activities.

While sharing the similarity of using supervised machine learning framework, our work differs from Andromaly in both motivations and the features concerned. As a host-based approach, Andromaly was aimed at detecting anomaly activities right on the mobile devices. However, we are motivated to provide maintainers of app markets and virus analysts with an approach that can effectively detect malapps, typically, in a large scale, by exploring the most discriminative features. The features we employed are directly extracted from static APK files of benign and malicious apps, whereas the major concerns of Andromaly are dynamic activities of the whole device.

Like Andromaly, MADAM [48, 52] was also a host-based anomaly detection

system and therefore was different from our work in the application scope. The main novelty of MADAM is that it retrieves features at four different levels: the kernel-level, app-level, user-level and package-level, and that it correlates the analysis of these features. In comparison, our work focuses on comparing the discriminative power and persistence of features defined by the platform or third-party apps.

As another behavior-based malware detector, Crowddroid [49] was distinguished by the way that it obtained application running data from users based on crowdsourcing. Crowddroid was also the name of a lightweight client installed on real users' devices for monitoring and sending behavior-related data of each application. However, restricted by the limit resources the data collection application could demand for, the author only considered system calls as features to characterize an application in their work.

Networkprofiler [50] focused on fingerprinting Android apps to re-identify their presence in the HTTP traffic. Network traffic can be combined with static features from APK files and run-time activities of application to provide a comprehensive characterization.

CopperDroid [51] reconstructed OS- and Android-specific behaviors of malapps by tracking and analyzing system calls. The features in our work are different from the meaning and usage of Android-specific behaviors defined in CopperDroid. First of all, we extract the features using static analysis, contrary to the dynamic approach in CopperDroid. Secondly, the representation and semantics of the features in our work is different from what is used in CopperDroid. The set of features in our work is predefined and used to train the classifiers, however, in CopperDroid, the behaviors of apps are reconstructed after running. Finally, we categorize all the features according to their generality and specificity.

Basically, the above approaches are orthogonal to static feature based systems such as ours, because they analyze the app behavior at run-time, while our approach performs a risk analysis by, for example, market maintainers, before the apps are distributed to end users. Because capturing run-time features is relatively time consuming compared to static analysis, we did not consider

run-time information features in this work in order to deal with the large app
 860 set.

5.2. Methods of Android Malapp Detection

There are policy based methods for the detection of malicious behavior. Enck *et al.* presented Kirin [53], in which they defined some permission-based rules to prevent dangerous app configuration during app installation. Apex [54]
 865 allowed users to define permission related policies after app installation to limit potential malicious behaviors. XManDroid [55] maintained an overall view of the whole system by building a system monitoring mechanism to control inter-component communication (ICC) and possible covert channels. Saint [56] prevented ICC-based attacks by regulating inter app ICC. SEAndroid [57] imple-
 870 mented a mandatory access control (MAC) enforcement layer that allowed for install-time or run-time MAC.

These approaches help users to enforce their security policy and shield them from malicious behavior of apps.

By means of static or dynamic information flow analysis, one knows how
 875 private and confidential data flows among the apps, the third-party libraries and the system [19, 58, 20, 59, 60, 61]. AppIntent [58] ran a static taint analysis to help checking the data transmission of an app. TaintDroid [19] was the first system to perform a dynamic taint tracing to find privacy leaks in Android smartphone. FlowDroid [20] implemented a static interprocedural data flow
 880 analysis. Amandroid [60] improved FlowDroid by linking the sources and the targets of ICC. In our previous work [61], we investigated what, why and how embedded sensors on Android devices are used in apps from both the official and third-party app markets. In [62], we designed and implemented a tool called “Alde” that can detect the data collected by analytics libraries and potential
 885 privacy leakage.

Due to the large volume of Android apps, machine learning based methods are introduced to deal with the malapp detection problem at a large scale. Our work also falls into this scope.

Andromaly [12] applied machine learning based anomaly detectors to help
 890 users to detect suspicious activities. Crowddroid [49] monitored system calls in-
 voked by an app as features and used k-means method to classify malicious apps.
 Rasthofer *et al.* [63] used machine learning approach for categorizing sources
 and sinks in Android information flow analysis. In our previous work [64],
 we capitalized on term frequency-inverse document frequency (tf-idf) and em-
 895 ployed k-Nearest Neighbor (k-NN) and Principal Component Analysis (PCA)
 for anomaly detection. In [65], we employed an ensemble learning method to
 detect malapps. In MAST[18], Chakradeo *et al.* used Multiple Correspondence
 Analysis (MCA) to measure the correlation between multiple app fea-
 tures. In [16], Gascon *et al.* proposed a method to use graph kernel based SVM
 900 classifier to find similarities between function call graphs of samples. In [17],
 Arp *et al.* presented Drebin, which was the most closely related to our work.
 They conducted a broad static analysis and extracted features to build a linear
 SVM classifier.

Our work differs from existing machine learning based methods in the fol-
 905 lowing four aspects. First, we explore much more feature sets and compare the
 classification performance and persistence of each feature set based on exten-
 sive experiments. Second, we categorize the feature sets into app-specific and
 platform-defined to evaluate the features and methods. Third, we employ four
 machine learning methods on a large app set and find that Logistic Regression
 910 outperforms others based on the full feature set. Finally, we analyze the con-
 tribution of each type of feature set for the detection and discover the usage
 patterns of features in malapps.

6. Conclusion

In this work, we conduct an empirical study on the discriminative power
 915 and persistence of eleven types of feature sets extracted from APK files for
 analyzing Android malapps at a large scale. We explore three types of new
 features and combine them with other related features to characterize the be-

haviors of malapps. We categorize these feature sets into platform-defined and app-specific features according to how these features are correlated with. We
 920 propose to employ four methods, namely, Logistic Regression (LR), linear Support Vector Machine (SVM), Decision Tree and Random Forest (RF) and compare their classification performance based on single and compound feature sets. Extensive experimental results indicate that (1) more features lead to better classification performance; (2) app-specific feature sets are more effective than
 925 platform-defined feature sets; (3) combining app-specific features and platform-defined features yields the best accurate classifier. Among the four classifiers, the LR classifier outperforms others with the full feature set.

To evaluate the persistence of feature sets, we employ app-specific and platform-defined features with two data sets, namely, the past and the new
 930 data set, to build classifiers. Evaluation results indicate that platform-defined features are more persistent on classification performance than app-specific features. By carefully analyzing the composition of relevant features selected by the LR classifier, we discover the usage patterns of features in the malapps. These patterns help to understand the behaviors of malapps with the most suitable
 935 features for automated malapp detection. The methods and models are evaluated based on a very large data set consisting of real-world benign (217,619) and malicious (18,363) apps collected in different time periods. The experimental results demonstrate the effectiveness of our methods and models (a TPR as 96% with a low FPR) at a large scale.

940 For the future work, we are exploring more features that effectively characterize apps' behaviors to further improve the detection performance. The ways of detecting malapps at a large scale based on labeled and unlabeled samples with semi-supervised learning methods will be investigated as well.

7. Acknowledgments

945 The work reported in this paper was supported in part by the Scientific Research Foundation through the Returned Overseas Chinese Scholars, Ministry

of Education of China, under Grant K14C300020, in part by Shanghai Key
 Laboratory of Integrated Administration Technologies for Information Security,
 under Grant AGK2015002, in part by ZTE Corporation Foundation, and in
 950 part by National Natural Science Foundation of China, under Grant 61672092.

References

- [1] Smartphone OS Market Share, Q2 2016.
 URL <http://www.idc.com/prodserv/smartphone-os-market-share.jsp>
- 955 [2] Android Apps on Google Play.
 URL <https://play.google.com/store?hl=en>
- [3] Number of available Android applications - AppBrain.
 URL <http://www.appbrain.com/stats/number-of-Android-apps>
- [4] Anzhi market.
 960 URL <http://www.anzhi.com/>
- [5] Appchina market.
 URL <http://www.appchina.com/>
- [6] Kaspersky, INTERPOL, Mobile Cyber Threats - Kaspersky Lab and
 INTERPOL Joint Report, Tech. rep., Kaspersky and INTERPOL (2014).
 965 URL [http://media.kaspersky.com/pdf/
 Kaspersky-Lab-KSN-Report-mobile-cyberthreats-web.pdf](http://media.kaspersky.com/pdf/Kaspersky-Lab-KSN-Report-mobile-cyberthreats-web.pdf)
- [7] D. Barrera, H. G. Kayacik, P. C. van Oorschot, A. Somayaji, A methodol-
 ogy for empirical analysis of permission-based security models and its appli-
 cation to Android, in: CCS '10, 2010, pp. 73–84. doi:10.1145/1866307.
 970 1866317.
- [8] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, D. Wagner, Android
 permissions: User attention, comprehension, and behavior, in: SOUPS
 '12, 2012. doi:10.1145/2335356.2335360.

- [9] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, X. Zhang, Exploring permission-induced risk in Android applications for malicious application detection, *TIFS* 9 (11) (2014) 1869–1882. doi:10.1109/TIFS.2014.2353996.
- [10] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, E. Chin, Permission re-delegation: attacks and defenses, in: *SEC '11*.
- [11] L. Lu, Z. Li, Z. Wu, W. Lee, G. Jiang, CHEX: statically vetting Android apps for component hijacking vulnerabilities, in: *CCS '12*. doi:10.1145/2382196.2382223.
- [12] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. Weiss, Andromaly: A behavioral malware detection framework for Android devices, *JHIS* 38 (1) (2012) 161–190. doi:10.1007/s10844-010-0148-x.
- [13] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, I. Molloy, Using probabilistic generative models for ranking risks of Android apps, in: *CCS '12*. doi:10.1145/2382196.2382224.
- [14] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, I. Molloy, Android permissions: a perspective combining risks and benefits, in: *SACMAT '12*. doi:10.1145/2295136.2295141.
- [15] Y. Zhou, W. Zhi, Z. Wu, X. Jiang, Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets, in: *NDSS '12*.
- [16] H. Gascon, F. Yamaguchi, D. Arp, K. Rieck, Structural detection of Android malware using embedded call graphs, in: *AISec '13*. doi:10.1145/2517312.2517315.
- [17] D. Arp, M. Spreitzenbarth, M. Hbner, H. Gascon, K. Rieck, Drebin: Effective and explainable detection of Android malware in your pocket, in: *NDSS '14*.

- 1000 [18] S. Chakradeo, B. Reaves, P. Traynor, W. Enck, MAST: Triage for market-scale mobile malware analysis, in: WiSec '13. doi:10.1145/2462096.2462100.
- [19] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, A. N. Sheth, TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones, in: OSDI '10.
1005
- [20] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, P. McDaniel, Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps, in: PLDI '14. doi:10.1145/2594291.2594299.
- 1010 [21] Y. Zhou, X. Jiang, Dissecting Android malware: Characterization and evolution, in: S&P '12. doi:10.1109/SP.2012.16.
- [22] Z. Xu, K. Bai, S. Zhu, Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors, in: WiSec' 13. doi:10.1145/2185448.2185465.
- 1015 [23] Y. Michalevsky, D. Boneh, G. Nakibly, Gyrophone: Recognizing speech from gyroscope signals, in: SEC '14.
- [24] Y. Aafer, W. Du, H. Yin, Droidapiminer: Mining api-level features for robust malware detection in Android, in: Security and Privacy in Communication Networks, Springer, 2013, pp. 86–103. doi:10.1007/978-3-319-04283-1_6.
1020
- [25] <uses-feature> | Android Developers.
URL <http://developer.Android.com/guide/topics/manifest/uses-feature-element.html>
- [26] K. W. Y. Au, Y. F. Zhou, Z. Huang, D. Lie, PScout: analyzing the Android permission specification, in: CCS '12. doi:10.1145/2382196.2382222.
1025

- [27] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, C.-J. Lin, LIBLINEAR: A library for large linear classification, JMLR 9.
- [28] J. R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
- 1030 [29] Myapp market.
URL <http://Android.myapp.com/>
- [30] Lenovomm market.
URL <http://www.lenovomm.com/>
- [31] Gfan market.
1035 URL <http://app.gfan.com/>
- [32] Nduoa market.
URL <http://nduoa.com/>
- [33] VirusTotal - free online virus, malware and URL scanner.
URL <https://www.virustotal.com/>
- 1040 [34] VirusShare.com.
URL <http://virusshare.com/>
- [35] C.-C. Chang, C.-J. Lin, LIBSVM: A library for support vector machines, ACM Transactions on Intelligent Systems and Technology 2, software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>. doi:10.1145/1961189.1961199.
1045
- [36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, JMLR 12 (2011) 2825–2830.
- 1050 [37] O. Tange, Gnu parallel-the command-line power tool, The USENIX Magazine 36 (1) (2011) 42–47.

- [38] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, K.-P. Wu, Droidmat: Android malware detection through manifest and api calls tracing, in: Asia JCIS '12. doi:10.1109/AsiaJCIS.2012.18.
- 1055 [39] M. J. Carlotto, Effect of errors in ground truth on classification accuracy, International Journal of Remote Sensing 30 (18) (2009) 4831–4849.
- [40] P. H. Chia, Y. Yamamoto, N. Asokan, Is this app safe?: A large scale study on application permissions and risk signals, in: WWW '12. doi:10.1145/2187836.2187879.
- 1060 [41] R. Pandita, X. Xiao, W. Yang, W. Enck, T. Xie, WHYPER: Towards automating risk assessment of mobile applications, in: SEC '13.
- [42] A. Gorla, I. Tavecchia, F. Gross, A. Zeller, Checking app behavior against app descriptions, in: ICSE '14. doi:10.1145/2568225.2568276.
- 1065 [43] M. Frank, B. Dong, A. P. Felt, D. Song, Mining permission request patterns from Android and facebook applications, in: ICDM '12. doi:10.1109/ICDM.2012.86.
- [44] N. Peiravian, X. Zhu, Machine learning for Android malware detection using permission and API calls, in: ICTAI '13. doi:10.1109/ICTAI.2013.53.
- 1070 [45] S. Rosen, Z. Qian, Z. M. Mao, AppProfiler: A flexible method of exposing privacy-related behavior in Android applications to end users, in: CO-DASPY '13. doi:10.1145/2435349.2435380.
- 1075 [46] G. Canfora, A. D. Lorenzo, E. Medvet, F. Mercaldo, C. A. Visaggio, Effectiveness of opcode ngrams for detection of multi family Android malware, in: Availability, Reliability and Security (ARES), 2015 10th International Conference on, 2015, pp. 333–340. doi:10.1109/ARES.2015.57.
- [47] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, J. Blasco, Dendroid: A text mining approach to analyzing and classifying code structures in An-

- droid malware families, *Expert Systems with Applications* 41 (4) (2014) 1104–1117.
- [48] G. Dini, F. Martinelli, A. Saracino, D. Sgandurra, Madam: A multi-level anomaly detector for Android malware., in: *MMM-ACNS*, Vol. 12, Springer, 2012, pp. 240–253. doi:10.1007/978-3-642-33704-8_21.
- [49] I. Burguera, U. Zurutuza, S. Nadjm-Tehrani, Crowdroid: behavior-based malware detection system for Android, in: *SPSM '11*. doi:10.1145/2046614.2046619.
- [50] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, D. Song, Networkprofiler: Towards automatic fingerprinting of Android apps, in: *INFOCOM '13*. doi:10.1109/INFOCOM.2013.6566868.
- [51] K. Tam, S. J. Khan, A. Fattori, L. Cavallaro, Copperdroid: Automatic reconstruction of Android malware behaviors., in: *NDSS*, 2015.
- [52] A. Saracino, D. Sgandurra, G. Dini, F. Martinelli, Madam: Effective and efficient behavior-based Android malware detection and prevention, *IEEE Transactions on Dependable and Secure Computing* PP (99) (2016) 1–1. doi:10.1109/TDSC.2016.2536605.
- [53] W. Enck, M. Ongtang, P. McDaniel, On lightweight mobile phone application certification, in: *CCS '09*. doi:10.1145/1653662.1653691.
- [54] M. Nauman, S. Khan, X. Zhang, Apex: extending Android permission model and enforcement with user-defined runtime constraints, in: *AsiaCCS '10*. doi:10.1145/1755688.1755732.
- [55] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, Xmandroid: A new Android evolution to mitigate privilege escalation attacks, *Technische Universität Darmstadt*, Technical Report TR-2011-04.
- [56] M. Ongtang, S. McLaughlin, W. Enck, P. McDaniel, Semantically rich application-centric security in Android, *Security and Communication Networks* 5 (6) (2012) 658–673. doi:10.1002/sec.360.

- [57] S. Smalley, R. Craig, Security enhanced (se) Android: Bringing flexible mac to Android., in: NDSS '13.
- [58] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, X. S. Wang, Appintent: Analyzing sensitive data transmission in Android for privacy leakage detection, in: CCS '13. doi:10.1145/2508859.2516676.
- [59] L. Li, A. Bartel, T. F. D. A. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, P. McDaniel, Ictta: detecting inter-component privacy leaks in Android apps, in: ICSE '15.
- [60] F. Wei, S. Roy, X. Ou, et al., Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps, in: CCS '14. doi:10.1145/2660267.2660357.
- [61] X. Liu, J. Liu, W. Wang, Y. He, X. Zhang, Discovering and understanding Android sensor usage behaviors with data flow analysis, World Wide Web (2017) 1–22doi:10.1007/s11280-017-0446-0.
- [62] X. Liu, S. Zhu, W. Wang, J. Liu, Alde: privacy risk analysis of analytics libraries in the Android ecosystem, in: 12th EAI International Conference on Security and Privacy in Communication Networks (SecureComm), 2016.
- [63] S. Rasthofer, S. Arzt, E. Bodden, A machine-learning approach for classifying and categorizing Android sources and sinks, in: NDSS '14.
- [64] D. Su, W. Wang, X. Wang, J. Liu, Anomadroid: Profiling Android applications' behaviors for identifying unknown malapps, in: 2016 IEEE Trustcom/BigDataSE/ISPA, 2016, pp. 691–698. doi:10.1109/TrustCom.2016.0127.
- [65] W. Wang, Y. Li, X. Wang, J. Liu, X. Zhang, Detecting Android malicious apps and categorizing benign apps with ensemble of classifiers, Future Generation Computer Systemsdoi:10.1016/j.future.2017.01.019.

Biography of the authors

Xing Wang is currently pursuing the Ph.D. degree with the School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China. He received the B.S. degree from Beijing Jiaotong University in 2009. He visited the King Abdullah University of Science and Technology, Thuwal, Saudi Arabia, in 2014. His main research interests lie in mobile security.

Wei Wang is currently an Associate Professor with the School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China. He earned his Ph.D. degree from Xi'an Jiaotong University, in 2006. He was a postdoctoral researcher in University of Trento, Italy, during 2005–2006. He was guest editor of IEEE Network and currently young AE of Frontiers of Computer Science Journal. He has authored or coauthored over 40 peer-reviewed papers in various journals and international conferences. His main research interests include network security and data mining.

Jiqiang Liu received the B.S. and Ph.D. degrees from Beijing Normal University, Beijing, China, in 1994 and 1999, respectively. He is currently a Professor with the School of Computer and Information Technology, Beijing Jiaotong University, Beijing. He has authored over 60 scientific papers in various journals and international conferences. His main research interests are trusted computing, cryptographic protocols, privacy preserving, and network security.

Zhen Han received the Ph.D. degree from the China Academy of Engineering Physics, Beijing, China, in 1991. He is currently a Professor with the School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China. He has authored or coauthored over 100 papers in various journals and international conferences. His main research interests are information security architecture and trusted computing.

Xiangliang Zhang is currently an Assistant Professor and directs the Machine Intelligence and Knowledge Engineering Laboratory with the Division of Computer, Electrical, and Mathematical Sciences and Engineering, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia. She received the Ph.D. (Hons.) degree in computer science from INRIA-University Paris-Sud 11, France, in 2010. She serves on the Program Committees of premier conferences, such as KDD, ICDE, ICDM, AAAI. Her main research interests and experiences are in diverse areas of machine intelligence and knowledge engineering, such as complex system modeling, computer security, and big data processing.

MANUSCRIPT

xing_wang.pdf

USCRIPT

jiquiang_liu.pdf

zhen_han.pdf

- First work defining app-specific and platform-defined features to characterize apps.
- A set of algorithms based on these features is employed to detect malapps.
- The relevant features are thoroughly analyzed to discover the patterns of apps.
- 18,363 malapps and 217,619 benign apps in real-world are used to validate our methods.