MalPat: Mining Patterns of Malicious and Benign Android Apps via Permission-Related APIs

Guanhong Tao, Zibin Zheng, *Senior Member, IEEE*, Ziying Guo, and Michael R. Lyu, *Fellow, IEEE*

***Abstract*—The dramatic rise of Android application (app) marketplaces has significantly gained the success of convenience for mobile users. Consequently, with the advantage of numerous Android apps, Android malware seizes the opportunity to steal privacy-sensitive data by pretending to provide functionalities as benign apps do. To distinguish malware from millions of Android apps, researchers have proposed sophisticated static and dynamic analysis tools to automatically detect and classify malicious apps. Most of these tools, however, rely on manual configuration of lists of features based on permissions, sensitive resources, intents, etc., which are difficult to come by. To address this problem, we study real-world Android apps to mine hidden patterns of malware and are able to extract highly sensitive APIs that are widely used in Android malware. We also implement an automated malware detection system, MalPat, to fight against malware and assist Android app marketplaces to address unknown malicious apps. Comprehensive experiments are conducted on our dataset consisting of 31 185 benign apps and 15 336 malware samples. Experimental results show that MalPat is capable of detecting malware with a high F_1 score (98.24%) comparing with the state-of-the-art approaches.**

***Index Terms*—Android applications, malware detection, permission-related APIs, random forests, software security.**

I. Introduction

THE past few years have witnessed the drastic increase of mobile apps providing various facilities for personal and business use. The proliferation of mobile apps is due to billions of users who enable developers to earn revenue through advertisements, in-app purchases, etc. A multitude of apps developed by many independent developers, involving unfriendly ones, can be hard for users to determine the trustworthiness of these apps. Whenever users install a new app, they are under the risk of installing malware. Unlike desktop apps, mobile apps can have the privilege, after declared (e.g., in *Manifest* file of Android platform), to access sensitive information such as contact lists, SMS messages, GPS, etc. To make full use of resources of mobile devices and support abundant functionalities of mobile apps, such mechanism of permission declaration remarkably fulfills its job. Unexpectedly, however, it also provides the opportunity for malware to hijack and steal sensitive information. According the report released by McAfee [1] in March 2016, more than 13 million mobile malware samples were collected by 2015, and it recorded a 72% increase in new mobile malware samples as to the last quarter. Data leaks are the typical behaviors of malware to steal users' contact lists, personal information, even money. As more people change their payment habits from cash to mobile banking, it has been a serious challenge to protect mobile users from spiteful attackers who may steal bank account credentials. Rasthofer *et al.* [2] identified a new malware family Android/BadAccents that stole, within two months, the credentials of more than 20 000 bank accounts of users residing in Korea. There are some underground groups whose revenues are earned by trading stolen bank account credentials. Symantec reported [3] that one underground group had made $4.3 million in purchases using stolen credit cards over a two-year period.

Under such severe situation of the security and privacy of mobile users, identifying malicious apps and defending against them from stealing sensitive information have posted an especially important challenge. Rasthofer *et al.* [4] proposed a machine-learning approach, SUSI, to identify lists of *sources* of sensitive data (e.g., user location) and *sinks* of potential channels to leak such data to an adversary (e.g., network connection). The categories published by SUSI along with other new categories of sensitive sources were adopted by MUDFLOW [5] to identify malware as well as their abnormal usage of sensitive data. Their work both focused on the data flows that originate from sensitive sources, which were effective and efficient as to identify abnormal behaviors of malware, but when it comes to classifying malicious apps, these approaches can be inefficient and hard to assist Android app marketplaces fighting against malware. Permissions declared in *Manifest* file are easy to capture the intention of apps for data usage, which can be utilized to identify malicious behaviors of apps [6]–[8]. Permission-based methods avoid high cost of time and computation, which, however, only capture the coarse-grained features of apps. Application programming interfaces (APIs) provided by Android operating system, in contrast, profoundly demonstrate the full

Manuscript received September 9, 2016; revised April 22, 2017 and October 9, 2017; accepted November 24, 2017. Date of publication December 20, 2017; date of current version March 1, 2018. This work was supported in part by the National Basic Research Program of China (973 Project 2014CB347701), in part by the National Natural Science Foundation of China under Grant 61722214 and Grant 61472338, in part by the Program for Guangdong Introducing Innovative and Enterpreneurial Teams (2016ZT06D211), and in part by the Pearl River S&T Nova Program of Guangzhou (201710010046). Associate Editor: Y. Le Traon. *(Corresponding author: Zibin Zheng.)*

G. Tao, Z. Zheng, and Z. Guo are with the School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510006, China, and also with Collaborative Innovation Center of High Performance Computing, National University of Defense Technology, Changsha 410073, China (e-mail: gwinhen@gmail.com; zhzibin@mail.sysu.edu.cn; guozy96@gmail.com).

M. R. Lyu is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, Hong Kong, China (e-mail: lyu@cse.cuhk.edu.hk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2017.2778147

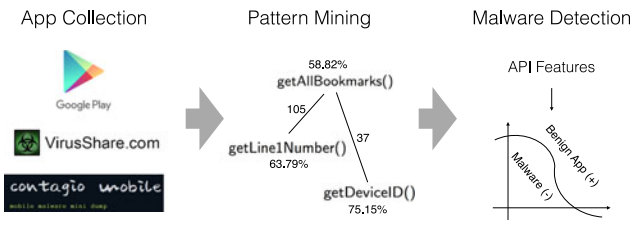

Fig. 1. Detecting malware via permission-related APIs. Starting from a collection of malicious and benign apps, we mine the patterns of malware. Based on extracted highly sensitive APIs, we train a two-class classifier to identify unknown apps.

picture of app behaviors in data usage. DroidAPIMiner [9] conducted an analysis to extract API-level features by statically counting the numbers of APIs used in malware and benign apps to capture different usage. However, the frequency analysis adopted by DroidAPIMiner may miss some key features. Furthermore, APIs that are not permission related can be less of value, and may introduce noise affecting feature extraction. In addition, DroidAPIMiner also utilized data flow analysis to obtain the frequent parameters, which exposes the similar issue as SUSI [4] and MUDFLOW [5] did. DREBIN [6] also adopted APIs as features but with more other types of features, such as hardware components, permissions, intents, etc. These features were all manually selected and required professional background knowledge. Besides, the final feature set contained about 545 000 features, which was a very large number of features and even more than the number of samples (123 453 benign apps and 5560 malware samples). It could require more time and efforts to extract these features as well as to train machine-learning models on them. Instead of using raw features of APIs, Zhang *et al.* [10] constructed a weighted dependency graph of APIs to represent the features of apps. Their approach was based on data flow analysis and the graph generation process was complex, which could be inefficient to assist app marketplaces. Therefore, we employ a machine learning approach to automatically extract permission-related APIs that can be utilized to distinguish malware from millions of apps.

In this paper, we study malicious and benign Android apps to mine hidden patterns of malware from fine-grained perspective. Android's main security mechanisms are based on sandboxes, which control physical and virtual resources of mobile devices to restrict the privilege of mobile apps, and permission mechanism is one of them. As permissions present the sensitive usage of Android resources, we focus on permission-related APIs currently employed in various Android apps. Differing from permission-based methods, our study captures fine-grained features extracting from APIs that contain more information of apps' behaviors. In addition, unlike API-level approaches, we neglect those APIs unprotected by permissions as they may introduce noise into the classification process and increase the computation cost. Our study includes three parts, illustrated in Fig. 1 and detailed later in the paper. First, we collect real-world Android apps consisting of 31 185 benign apps and 15 336 malware samples from the Internet. Second, based on the analysis of our collected datasets, we extract hidden patterns of malware with respect to benign apps in terms of Android APIs.[1] Finally, with the extracted features, malicious apps can be effectively detected and classified by malware detection systems. Consequently, we present MalPat, an automated malware detection system, which mines malware patterns from known malware samples automatically, and identifies malware efficiently.

The contributions of this paper are summarized in the following:

1) We present a thorough study on the different usage of permissions and APIs using statistical analysis techniques, and reveal their capabilities of differentiating malware and benign apps.
2) We mine the hidden patterns of malware by extracting highly sensitive APIs protected by permissions and analyzing the correlation of different APIs.
3) We make full use of characteristics of the machine learning approach by engaging it much earlier in the system architecture to learn the potential features from existing data.
4) We propose MalPat,[2] an automated malware detection system, which extracts crucial features from tens of thousands of apps automatically, to assist Android app marketplaces to fight against malware.
5) We conduct comprehensive experiments on a large-scale dataset consisting of 31 185 benign apps crawled from Google Play [11] and 15 336 malware samples collected from VirusShare [12] and Contagio [13]. The experimental results show that MalPat can detect malware with the F_1 score of 0.9824 using only 50 APIs, and outperforms the state-of-the-art approaches.

The rest of this paper is organized as follows. Section II discusses the related work addressing privacy and security problems on Android platform. Section III describes the dataset that we have collected. In Section IV, we statistically analyze the usage of permissions and APIs, mine the hidden patterns of malware, and discuss highly sensitive APIs. We then present our malware detection system, MalPat, in Section V. Comprehensive experiments on the real-world dataset are conducted in Section VI. Finally, the conclusion and future work are described in Section VII.

II. Related Work

Malware detection and classification are challenging problems, especially on mobile platforms. Researchers have paid great efforts to address these problems in various ways. In this section, we discuss the previous work addressing malware problems.

A. Dynamic Analysis

In-depth analysis of malware and app behaviors is carried out by many researchers from dynamic perspective. TaintDroid [14] identified sensitive information at a taint source, and tracked, dynamically, the impact of labeled data to other data that might

[1] In the rest of this paper, API and Android API are both referred to the permission-related API on Android platform.

[2] http://malpat.inpluslab.com

leak the original sensitive information. The impacted data were identified before they left the system at a taint sink. Comparing to TaintDroid, more information leaks were found by VetDroid [15]. It constructed permission use behaviors to examine the internal sensitive behaviors and find information leaks. VetDroid can help identify subtle vulnerabilities in some apps. To capture both the OS-level and Java-level semantics simultaneously and seamlessly, DroidScope [16] collected detailed native and Dalvik instruction traces to track information leakage through both Java and native components. These dynamic methods all aim to conduct taint analysis to detect suspicious behaviors during runtime. With the dramatically increasing number of Android apps, it restricts dynamic methods from high-speed detection. Besides, we aim to detect malware instead of capturing stealthy behaviors, dynamic analysis can be inefficient to assist mobile app marketplaces fighting against malware.

B. Static Analysis

Differing from dynamic analysis, static approaches can effectively deal with millions of apps and analyze patterns of malicious and benign behaviors without online running. Previous studies focused on malware detection and classification were conducted via various methods, such as static taint analysis [17], information flow analysis [18], [19], and probabilistic models [7], [20], etc. DroidRanger [8] utilized a permission-based behavioral footprinting scheme to detect new samples of known Android malware families and a heuristics-based filtering scheme was adopted to identify certain inherent behaviors of unknown malicious families. Permission-based method is coarse-granted and cannot capture more detailed behaviors of apps. We analyze and elaborate more details in the rest of this paper. To capture fine-grained information, Apposcopy [17] proposed a high-level language to capture the signatures describing semantic characteristics of malware families. Based on the extracted signatures, a static analysis was conducted to detect certain malware families. The main concern of Apposcopy is to identify certain family of malware, which is different from the purpose of our work. Based on static data-flow analysis, Flowdroid [21], ded [22], and CHEX [23] were able to precisely detect potential malicious behaviors of Android apps. Such technique can effectively capture the exactly stealthy and malicious behaviors of apps, but it can introduce a very high overhead and be inefficient to fight against malware.

C. Machine Learning Approach

Classifying malware automatically is an open problem commonly addressed by employing machine learning techniques. Permissions are designed to protect sensitive resources on Android platform, which directly demonstrates the sensitive behaviors of Android apps. Bartel *et al.* [24] found that there were a large part of apps suffering from permission gaps, i.e., not all the permissions they declared were actually used. By analyzing permission usage among millions of benign and malicious apps, they can effectively expose abnormal behaviors and finally distinguish malware from various apps. Peng *et al.* [7] utilized the advantage of permissions and was able to correctly classify malware. However, the permission declaration of Android apps is coarse grained, lacking the ability of capturing in-depth behaviors of malicious apps. To this point, Gorla *et al.* [25] selected a subset of APIs that were governed by Android permission setting naming sensitive APIs, which were identified by Felt *et al.* [26]. These sensitive APIs were used as binary features to train an OC-SVM to identify the outlier apps. Similarly, DREBIN [6] also utilized APIs as features but with more other types of features (hardware components, permissions, intents, etc.). These features were all manually selected and required professional background knowledge. Besides, the final feature set contained about 545 000 features, which was a very large number of features and even more than the number of samples (123 453 benign apps and 5560 malware samples). It could require more time and efforts to extract these features as well as to train machine-learning models on them. Utilizing the categorization of Android APIs published by SUSI [4], Avdiienko *et al.* [5] added three new categories to detect malware: sensitive resources, intents, nonsensitive sources, and sinks. With these categories as app features, Avdiienko *et al.* trained a ν-SVM [27] to detect malware (MUDFLOW). Differing from the features used in MUDFLOW, we only adopt the permission-related APIs used by apps without any other additional information to detect malware. Moreover, random forests are employed in our malware detection system as the classifier to identify malware. Random forest is a well-known machine learning approach in classification and regression, which consists of a set of binary decision trees [28]. By training multiple decision trees, random forests combine the results from these decision trees with voting approach. Details of the training process of random forests are illustrated in Section V.

Previous studies focused on data flow can be inefficient for malware detection and classification, resulting in high cost of time and resources in fighting against malware. In contrast, sensitive or critical API-based approaches are easy to construct and can adjust rapidly according to the change of malware. However, very few efforts have been made in conducting a thorough analysis of permission-related APIs, which can significantly improve the efficiency of malware detection with easy and practical approaches. In this paper, we try to address this problem and give an empirical study of permission-related APIs.

III. Dataset Collection

To mine the hidden patterns of malware, we study the behaviors of malicious and benign Android apps in the real world. In this section, we briefly describe two sets of Android apps and their characteristics.

Benign apps: To collect the benign apps, we used Google Play [11], a leading Android app marketplace in the world. BenignRan, the first dataset, consists of 28 787 apps randomly collected from May to December 2015. The category distribution of BenignRan is shown in Fig. 2. The right part in light gray are the categories belonging to *Games* category. Except for the *Tools* category, containing 3787 apps, other categories all include almost similar amount of apps. The randomly collected apps may have bias on categories and API usage. To ease such

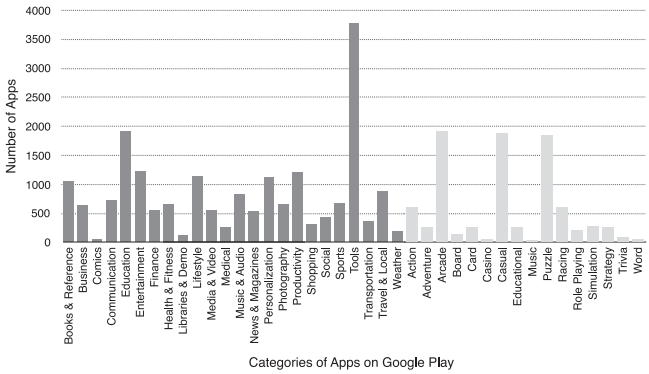

Fig. 2. Categories of Android Apps collected from Google Play, where the green histograms are the categories belonging to *Games* category.

potential influence, we also downloaded the top 100 most popular free apps for each of the 30 app categories based on the same list. Avdiienko *et al.* [5] had collected as of March 1st, 2014. Although some apps cannot be found due to unknown reasons, we were still able to collect 2398 apps, which formed our second benign dataset, BenignPop. These two benign datasets are all employed in our experiments comprising the whole benign dataset, BenignAll, with 31 185 Android apps, and the detailed discussions are presented in Section VI.

Malicious apps: Our malicious apps were collected from two sources:

1) A set consisting of 24 317 malicious apps was obtained from VirusShare [12] with the period from May 6th, 2013 to March 24th, 2014. As some of these apps could not be correctly decompiled or the `.apk` files were missing, 14 843 apps were left for further study and evaluation.
2) 493 Android malware samples provided by Contagio [13] were also adopted in our dataset. These malicious apps were collected from October 2010 to January 2016.

Hence, the malware dataset totally contains 15 336 malicious apps. It should be noted that there is no overlap between the benign and malicious app datasets.

IV. Pattern Mining

In this section, we first make a rough analysis on permission and API distribution of Android apps, respectively. In the API-permission mapping list extracted by Pscout [29], there are 32 304 APIs governed by 71 permissions. As the differences of APIs in different versions of Android operating systems, we adopt the API mapping list of Jelly Bean (Android 4.1) in the following analysis. The influence of API change, which has been studied by previous work [30], is out of the scope of this paper. By analyzing permissions and APIs used in malicious and benign apps, we can obtain a brief statistical result indicting the difference between these two types of apps.

In order to understand to what extent benign and malicious apps are of difference with respect to permissions and APIs, we study two research questions in the following:

1) $\mathbf{RQ}_1$: *Is the usage of permissions in malware significantly different from that of benign apps?* This research question aims at revealing the permission usage patterns of malware as compared to benign apps. The conjecture is that the usage of permissions would not significantly differentiate malware from benign apps. This conjecture is based on the previous work carried out by Felt *et al.* [26] pointing out that apps tend to request extra privileges (declare more permissions than they actually need). Thus, we test the following null hypothesis:
   H_{0p}*: There is no significant difference between the permissions used by malicious and benign apps.*
1) $\mathbf{RQ}_2$: *Is the usage of APIs in malware significantly different from that of benign apps?* This research question is similar to $\mathbf{RQ}_1$; however, it considers APIs instead of permissions as the main factor to analyze. It is intuitive that apps tend to request more permissions in case of further use of privileged resources (e.g., APIs), but never actually acquire corresponding APIs. Based on this reasonable fact, APIs could reveal the real behaviors of apps to some extent. Specifically, we test the null hypothesis as
   H_{0a}*: There is no significant difference between the APIs used by malicious and benign apps.*

In the following sections, detailed analysis on permissions and APIs is conducted by comparing their use in corresponding benign and malicious apps. Specifically in Section IV-C, we use Mann–Whitney test [31] so as to capture statistical significance of the different usage of permissions and APIs in malware and benign apps.

As to mining the hidden patterns of malware comparing to benign apps, we utilize machine learning approaches and successfully extract highly sensitive APIs.

A. Permission Distribution

An overview study of privacy and security is conducted on the benign and malicious app datasets, and Fig. 3 gives the result of the permission usage in these apps. The top 25 most frequently used permissions are illustrated in Fig. 3 and sorted by the order of usage percentage in the benign app dataset.

As shown in Fig. 3, some of the highly utilized permissions are both adopted by benign and malicious apps, which can hardly be distinguished between these two types. Nevertheless, there are a few permissions that are frequently used by malicious apps, such as ACCESS_WIFI_STATE, SEND_SMS and GET_TASKS, etc. WiFi state can be utilized to obtain the network information as well as location context of users, which may be the reasons why it is highly used. As to SEND_SMS, it is obvious that bank verification and other sensitive information are protected via messages. The last permission seems unreasonable that it is widely adopted in malicious apps. However, if we take a look into the APIs protected by the permission GET_TASKS, we can find one of them, i.e., getRecentTasks(). As illustrated in the API documents [32], this method returns a list of the tasks that the user has recently launched, which can be used by malware for malicious purposes, such as monitoring user behaviors, attacking certain tasks, etc. This method is no longer available to third party applications, but there may still remain

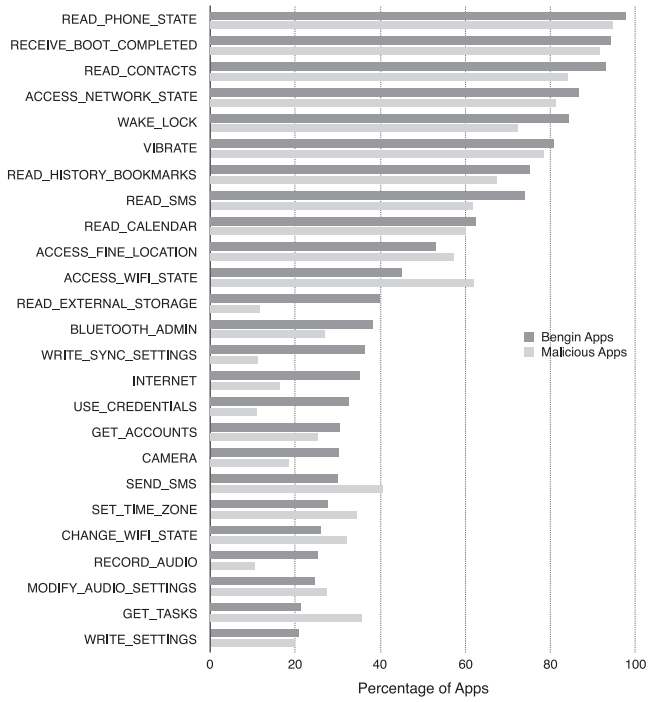

Fig. 3. Top 25 most used permissions in benign and malicious app dataset.

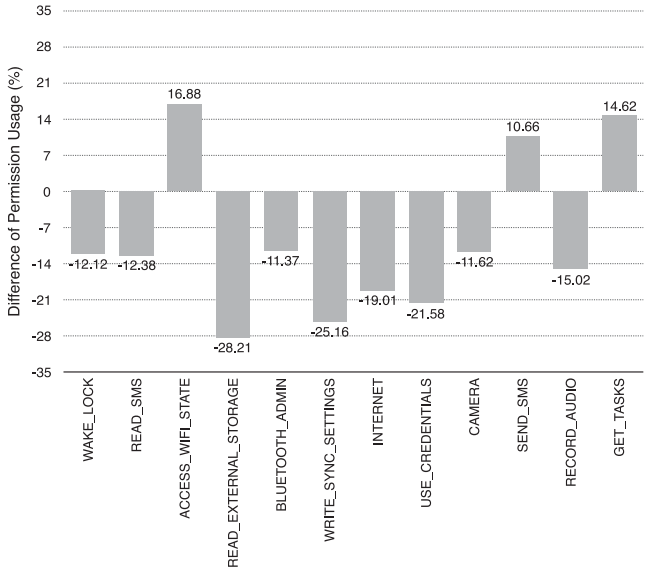

Fig. 4. Difference of permission usage between malicious and benign apps, where the percentage of the differences is larger than 10%.

such methods that can be used for malicious purposes, which is studied in this paper.

To obtain a deeper analysis of the difference of permission usage between malicious and benign apps, we compare the percentage of two datasets. Fig. 4 shows the results that the percentage of the differences is larger than 10%, where the positive values represent that more malicious apps use the current permission than benign apps. There are 12 permissions denoting a

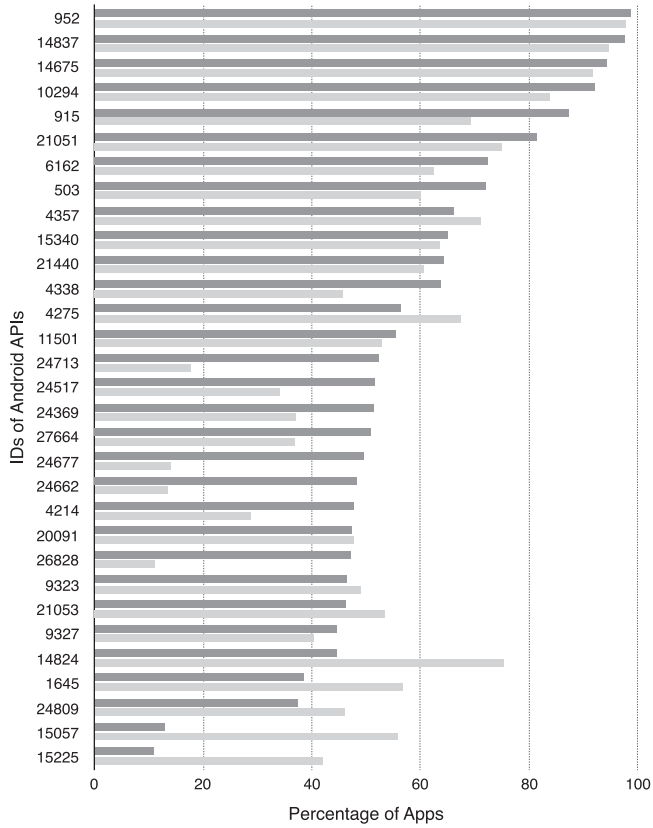

Fig. 5. Top 25 most used APIs in benign and malicious app dataset.

big difference between malicious and benign apps. The differences of three permissions, READ_EXTERNAL_STORAGE, WRITE_SYNC_SETTINGS, and USE_CREDENTIALS, even surpass 20%, but these three permissions are more used by benign apps rather than malware, which is very different from the study conducted by Peng *et al.* [7]. The malware dataset used in their study contained only 378 apps, which may cause unpredictable deviations. Based on our datasets, three permissions expose significant difference between malicious and benign apps, which can be utilized to distinguish their types. Such coarse-grained features, however, cannot effectively differentiate malicious and benign apps which is discussed in Section VI. Therefore, regarding the research question $\mathbf{RQ}_1$, it can be observed that some permissions present the ability of distinguishing malware and benign apps, but it is still hard to determine whether permissions actually possess such feature in the big picture. We try to answer this question in Section IV-C. In the following section, we study the difference of these two types of apps from a fine-grained perspective, i.e., API usage.

B. API Distribution

Statistical patterns of API usage are studied and the top 25 most frequently used APIs in both benign and malicious app datasets are illustrated in Fig. 5. There are 31 APIs in the figure, where the whole mapping of API IDs ranging from 0 to 32303 is listed on our website [33]. We can

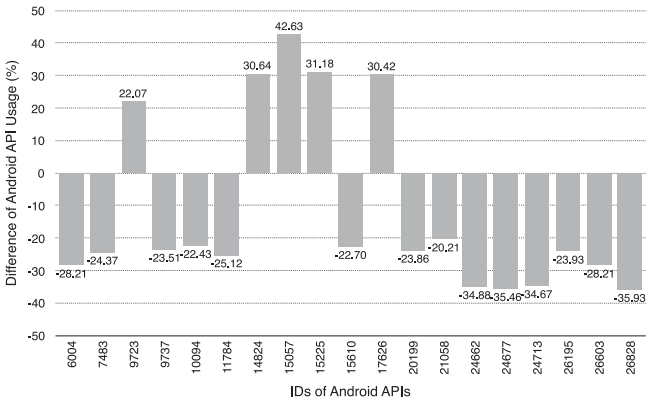

Fig. 6. Difference of API usage between malicious and benign apps, where the percentage of the difference is larger than 20%.

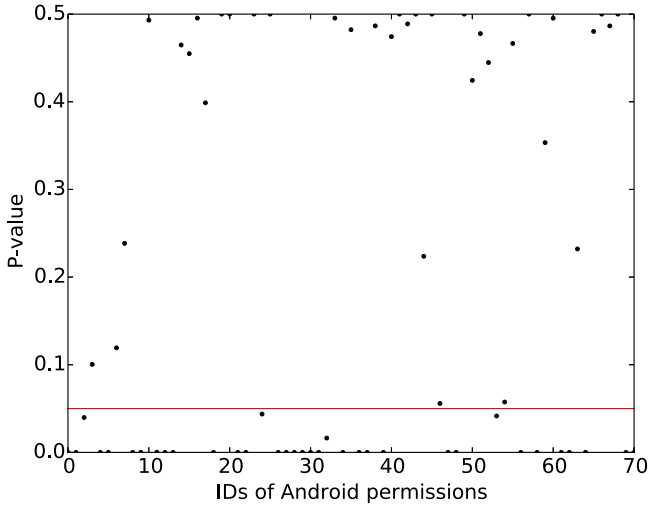

Fig. 7. Mann–Whitney test (p-value) on Android permissions. The red line represents the significant level ($\alpha = 0.05$).

observe that most of the top used APIs are not distinguishable as they are all widely employed in both benign and malicious apps. However, API IDs, 14824, 15057, and 15225, are more frequently used in malware comparing with benign apps. 75.15% malicious apps use the method, getDeviceID(), from android.telephony.TelephonyManager package. The method getDeviceID() returns the unique device ID, e.g., the IMEI for GSM and the MEID or ESN for CDMA phones [34], which can be used to identify mobile devices as the tag to trace anything occurring on those devices. If such sensitive identifications are utilized for malicious purposes, it can be dangerous and risky to the privacy and security of mobile users. Therefore, methods that may lead to unsecure information leaks should be paid close attention, especially the ones that are frequently employed in malware rather than benign apps.

To find the big difference of API usage between two types of apps, we select those APIs whose differences are larger than 20% in Fig. 6. The methods whose API IDs are 9723, 14824, 15057, 15225, and 17626 are more frequently engaged in malicious apps. Specifically, the difference of the method 15057 is larger than 40%. We list these APIs with large differences in Table I. The method with API ID 15057 is getSubscriberId() from the same package of getDeviceID(). Differing from getDeviceID(), the method getSubscriberId() can be much more dangerous as it returns the unique subscriber ID, e.g., the IMSI for a GSM phone [34]. According to Wikipedia [35], International Mobile Subscriber Identity (IMSI) is used in any mobile network that interconnects with other networks. Especially, for GSM, UMTS and LTE networks, the IMSI number is provisioned in the SIM card, which means that it can be utilized to mark any SIM card, i.e., mobile user. Hence, it is essential and important to mine the API usage patterns from malicious and benign apps. Table I also lists the permissions that govern those APIs. Some methods, such as API IDs of 6004, 9723, etc., are protected by more than one permissions. As the protection of APIs are overlapped via permissions, the features based on permissions can be hard to clearly and correctly reflect the patterns of apps and to detect malware. Therefore, we employ fine-grained features, i.e., permission-related APIs, to mine hidden patterns of malicious apps for efficient malware detection. The use of APIs do show a certain level of difference between malware and benign apps. However, similar to the analysis in the last section, we still cannot conclude the significant difference regarding research question $\mathbf{RQ}_2$ without more detailed and thorough analysis, which is carried out in the Section IV-C.

C. Permissions Versus APIs

In this section, we use Mann–Whitney test [31] to analyze statistical significance of the different usage between malicious and benign apps. The significance level to reject the null hypothesis is set as $\alpha = 0.05$. As the API-permission mapping list [29] with 71 permissions and 32 304 APIs are employed in the study, we apply tests on each permission and each API of malicious and benign apps, as well as the average usage of them in the following three sections.

1) Permission test: We separate the malicious and benign apps into two groups, and use the Mann–Whitney test to analyze statistical significance on every permission. More specifically, for one permission, if an app (malicious or benign one) declared this permission, then the sample value is set to 1; otherwise, it will be set to 0. Therefore, in one test, two sets of samples represent one specific permission usage of malicious and benign apps, respectively, and the range of each sample value is $\{0, 1\}$. Fig. 7 shows the p-values of all the tests on permissions and the red line represents the cutoff level ($\alpha = 0.05$). Since we apply multiple tests on permissions, we adjust these p-values using Holm's correction procedure [36]. The adjusting procedure sorts the p-values from n test in ascending orders, and then multiply them with n to 1, respectively. For instance, let $p^{(1)} \leqslant p^{(2)} \leqslant \cdots \leqslant p^{(n)}$ be the p-values of n test on permissions, and $0 < \alpha < 1$ is the significant level. The assessment on the n hypotheses is performed as follows:

$$p^{(1)} > \frac{\alpha}{n}, \quad p^{(2)} > \frac{\alpha}{n-1}, \quad \ldots, \quad p^{(n)} > \frac{\alpha}{1}. \tag{1}$$

TABLE I
API-PERMISSION MAPPINGS

API ID	Package Name	Method Name	Permission
6004	com.android.providers.media. MediaProvider	openFileAndEnforce PathPermissionsHelper()	READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE
7483	com.android.browser. GoogleAccountLogin	<init>()	INTERNET
9723	android.telephony.TelephonyManager	getCellLocation()	ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION
9737	android.accounts.AccountManagerService	getAuthToken()	USE_CREDENTIALS
10094	android.accounts.AccountManagerService	invalidateAuthToken()	USE_CREDENTIALS MANAGE_ACCOUNTS
11784	android.content.ContentService	removePeriodicSync()	WRITE_SYNC_SETTINGS
14824	android.telephony.TelephonyManager	getDeviceId()	READ_PHONE_STATE
15057	android.telephony.TelephonyManager	getSubscriberId()	READ_PHONE_STATE
15225	android.telephony.TelephonyManager	getLine1Number()	READ_PHONE_STATE
15610	com.android.contacts.activities. ContactDetailActivity	onAttachFragment()	READ_PHONE_STATE GET_ACCOUNTS READ_SYNC_SETTINGS
17626	android.telephony.SmsManager	sendTextMessage()	SEND_SMS
20199	com.android.calendar.DayView	<init>()	READ_CALENDAR
21058	android.net.ConnectivityManager	isActiveNetworkMetered()	ACCESS_NETWORK_STATE
24662	android.widget.VideoView	pause()	WAKE_LOCK
24677	android.widget.VideoView	stopPlayback()	WAKE_LOCK
24713	android.widget.VideoView	start()	WAKE_LOCK
26195	android.webkit.WebViewClassic	drawContent()	WAKE_LOCK
26603	android.webkit.WebViewClassic	onPause()	WAKE_LOCK
26828	android.widget.VideoView	setVideoPath()	WAKE_LOCK

After the adjustment, we can notice that the usage of 31 (out of 71) permissions in malware exhibits a statistically significant difference as compared to benign apps (p-values < 0.05). From the above analysis, we cannot directly accept or reject the null hypothesis H_{0p} as not all the tests presents the statistical significance. However, the 31 permissions that shows statistical significance can be employed as major features in analyzing malware and worth further study.

2) API test: Similar to the last section, this section focuses on the tests on APIs instead of permissions. For one specific API, *call sites* of this API used in the app (malicious or benign one) indicate the sample value of this app. The sample value is set as 0 if this API is never used in the app. Therefore, in each test, two sets of samples represent one specific API usage of malicious and benign apps, respectively, and the range of each sample value is $[0, +\infty)$. Fig. 8 illustrates the p-values of all the tests on APIs and the red line represents the cutoff level ($\alpha = 0.05$). From the figure, we can observe that most tests on APIs have p-values larger than α, which indicates that most test results cannot reject the null hypothesis H_{0a}. Since we apply multiple tests on APIs as well, the p-values are also adjusted with Holm's correction procedure. After adjustment, we notice that the usage of 106 (out of 32 304) APIs in malware exhibits a statistically significant difference as compared to benign apps (p-values < 0.05). From the above analysis, it can be noticed that only a very small part of APIs usage have the statistical significance between malware and benign apps. In addition, with these test results, we cannot directly accept or reject the null hypothesis H_{0a}, but the usage of 106 APIs in malware is worth further study as it provides fine granted and more features than permissions.

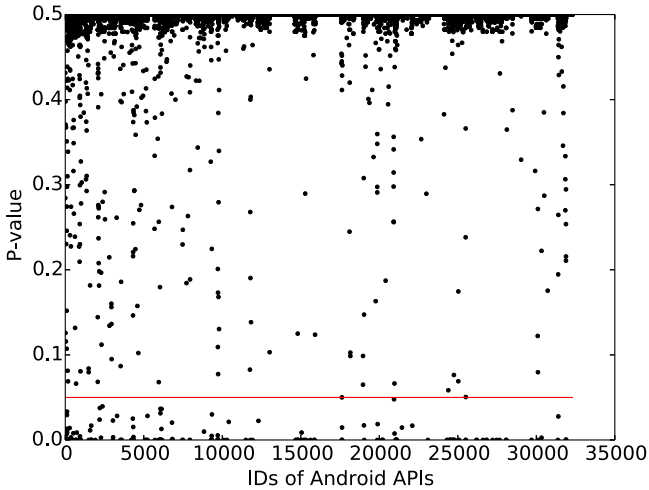

Fig. 8. Mann–Whitney test (p-value) on Android APIs. The red line represents the significant level ($\alpha = 0.05$).

3) Comparison test: The previous two sections study the statistical significance of each permission and API one by one, and the conclusions are based on intuitive analysis. In this section, we investigate the different usage of permissions and APIs between malicious and benign apps by considering all the employment of these features. For each permission, we compute the average declaration in all malicious and benign apps, respectively. Thus, two sets of samples represent the average declaration of permissions in malware (malware_perm) and benign apps (benign_perm), and the range of sample value lies in $[0, 1]$. The first line of Table II reports the result of the Mann–Whitney

TABLE II
USE OF PERMISSIONS AND APIS BY MALICIOUS AND BENIGN APPS: MANN–WHITNEY TEST (p-VALUE)

Test	p-value
malware_perm versus benign_perm	0.3442
malware_API versus benign_API	$<$0.0001

test (p-value) on average permission usage. Apparently, the average usage of permissions in malware does not show a statistically significant difference comparing to benign apps (p-value > 0.05). Hence, from the perspective of average usage of permissions, we cannot reject the null hypothesis H_{0p}, and the conclusion is that *there is no significant difference between the permissions on average used by malicious and benign apps*. For each API, we compute the average *call sites* in all malicious and benign apps, respectively. Thus, two sets of samples represent the average *call sites* of APIs in malware (malware_API) and benign apps (benign_API), and the range of sample value lies in $[0, +\infty)$. The second line in Table II shows the p-value of the Mann–Whitney test on average usage of APIs. As we can notice from the table, the usage of APIs in malware exhibits a statistically significant difference as compared to benign apps (p-value <0.05). Therefore, from the perspective of average usage of APIs, we can reject the null hypothesis H_{0a}, and come into the conclusion that *APIs used by malware are on average significantly different from APIs used by benign apps*.

Summarizing, the usage of APIs in malware demonstrates statistical significance as compared to benign apps. In analyzing behaviors of malicious and benign apps, APIs can be a nonnegligible factor with respect to permissions. Therefore, in this paper, we mainly focus on API usage patterns of malicious and benign apps. More details are analyzed in the following sections.

D. Hidden Patterns

To mine the hidden patterns of malware, we study the APIs extracted by a trained classifier that are highly sensitive in malware classification, and analyze the co-used APIs in both malicious and benign app datasets. Some specific APIs have the capabilities to distinguish malware from millions of Android apps. Therefore, by training a classifier, it gives a full picture of the API usage in different types of apps and identify malicious patterns with highly sensitive APIs. As co-used APIs can represent the usage patterns of malware stealing users' sensitive information, the difference of co-used APIs between malicious and benign apps is an effective and efficient way to attract the problem. In the following sections, highly sensitive APIs extracted through a trained classifier are presented, and co-used APIs are studied based on both malicious and benign app datasets.

1) Highly sensitive APIs: The permission mechanism is utilized to protect sensitive information on Android platform so as to prevent unpredictable data leaking by malware. However, there are hundreds and thousands of APIs governed by only one permission, which makes it hard to distinguish the behaviors

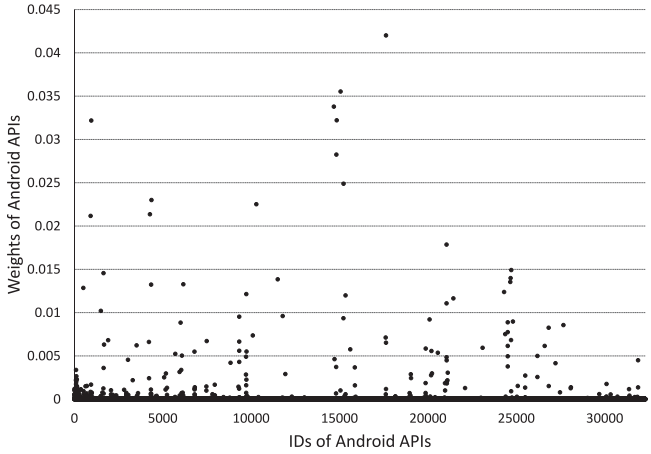

Fig. 9. Weights of Android APIs.

TABLE III
API WEIGHT DISTRIBUTION

Weight	$>$0.01	$>$0.001	$>$0.0001	$>$0	All
APIs	26	137	315	2939	32 304

of apps through one permission as the requirements of functionalities. Permission-related APIs, on the other hand, can not only hold the features of permissions, but also be more distinguishable among millions of apps. Therefore, we take these permission-related APIs as the features of apps, and trained a random forests classifier. The random forests classifier can learn the importance of different APIs in the training process and output the weights of APIs. The training process is presented in Algorithm 1 of Section V. The weights given by the classifier indicate the importance of APIs in classifying malicious and benign apps. As shown in Fig. 9, the weights of different APIs trained and tested on real-world datasets range from 0 to 0.045. Most of the APIs cannot be used to classify malicious and benign apps as their weights are all **zero**. Only a few of these APIs have the ability to distinguish the types of apps. The weight distribution of APIs is listed in Table III. Among the 32 304 permission-related APIs, only 2939 APIs have nonzero weights. The weights of 26 APIs listed on our website [33] are over 0.01, which shows the importance of them to identify malware. We compared the 26 APIs with the most different API usage listed in Table I and found that 8 of them are the same. The API 17626 with the highest weight of 0.042 is also among the eight highly sensitive APIs. As listed in Table I, the API 17626 is the method sendTextMessage() from package android.telephony.SmsManager, which is also identified as a suspicious API call in DREBIN [6]. Other highly sensitive APIs such as getDeviceID(), getSubscriberId(), and getLine1Number(), etc., are all highly dangerous APIs which have discussed in the previous sections. These APIs with high weights are especially important in identifying malware. Hence, we utilize this kind of APIs to retrain our classifier, which is discussed in Section V.

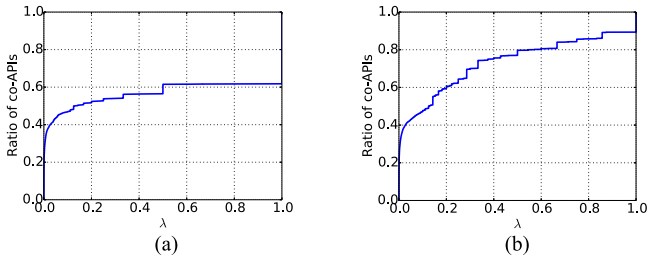

Fig. 10. Distribution of the correlation of co-APIs λ in datasets. (a) Distribution of λ value in malicious app dataset. (b) Distribution of λ value in benign app dataset.

2) Coused APIs: To investigate the correlation between different APIs, we first define the correlation value of co-used APIs (co-APIs) in the following.

Definition 1 (Co-API): Given two APIs a_i and a_j, if a_i and a_j are both employed in the same (benign or malicious) app, these two APIs are called co-API. The correlation of co-APIs is the possibility how they are likely to be used together in the app dataset, which is to calculate the co-API value λ as Jaccard measure [37]:

$$\lambda = \frac{\mathbb{D}(a_i \cap a_j)}{\mathbb{D}(a_i \cup a_j)} \tag{2}$$

where $\mathbb{D}(a_i \cap a_j)$ indicates the number of unique apps that used API a_i and a_j together, and the number of apps that used either API a_i or a_j is $\mathbb{D}(a_i \cup a_j)$.

Fig. 10 shows the co-API usage in malicious and benign apps, respectively. In Fig. 10(a), it shows the correlation value λ against the ratio of co-APIs. From the figure, we can notice that about 40% of co-APIs have the correlation value of 1.0. But we found that most of these co-APIs with $\lambda = 1.0$ were only employed in one malicious app. As to benign apps shown in Fig. 10(b), there is not a large part of the dataset with λ value of 1.0. About only 10% of API pairs have the correlation of 1.0 in benign apps. To compare the difference of co-API usage of malicious and benign apps, we select top 20 most different co-APIs employed in malicious and benign app datasets. As illustrated in Fig. 11, all the 20 co-APIs have the difference larger than 35%, which are the significant patterns indicating app behaviors. The positive value of difference denotes that the current co-API is employed in malware more than benign apps. The largest difference is achieved by the co-API <14837, 15057>, which is employed by 43.08% more apps in malicious app dataset. The API 14837 is the method <init>() from package com.android.emailcommon.service.EmailServiceProxy. We cannot find the official document about this API, but from the name of the method, we can conjecture that this API is used to set email services, more specifically, the proxy of email services. Obviously, this is a sensitive API that can be hijacked by malicious behaviors. The co-used API 15057 discussed in Section IV-A is the method getSubscriberId(), which is also a very dangerous API. From the top 20 most different co-APIs, we observe that 8 of them contain the API 15057, and these co-APIs are all employed in malicious apps more than

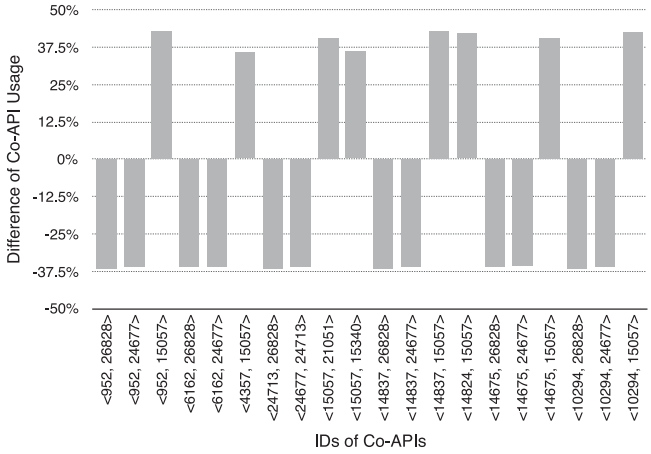

Fig. 11. Top 20 most different co-API usage between malicious and benign apps. The positive percentage denotes that more malicious apps employ the current co-API comparing with benign apps, and vice versa.

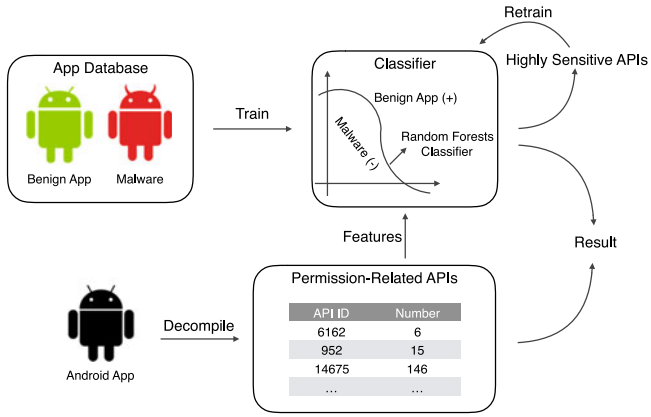

Fig. 12. Architecture of MalPat.

benign apps. This observation indicates the hidden patterns of malware and can be utilized to identify malicious apps. As the correlation of APIs is considered in the training process of random forests, the co-APIs are not regarded as features in the classifier.

V. Malware Detection

To assist Android app marketplaces to fight against malware, we proposed an automated malware detection system, MalPat,[3] to detect any suspicious Android apps. Permission-related APIs are adopted as main features in MalPat to classify malicious and benign apps based on their unique usage patterns. As shown in Fig. 12, there are two main parts of MalPat to detect malware: model training and malware detection. A newly coming Android app is decompiled first to extract permission-related APIs and highly sensitive APIs are selected as the features. Apps in the database consisting of malicious and benign apps are engaged to train the classifier. The detection process is based on the trained classifier. Detailed explanations are presented in the following.

[3]http://malpat.inpluslab.com

A. Feature Extraction

Data samples are the base of training a model. As described in Section III, we crawled two sets of Android apps, malicious and benign apps, comprising the datasets of our training process. There are 31 185 apps in the benign app dataset and 15 336 malware samples in the malicious app dataset. With all these malicious and benign apps, we can extract features from the source codes of decompiled files. The installation package of Android apps is the `.apk` file, which can be disassembled by the well-known decompiling tool, *Apktool* [38]. It can recover main files organizing source codes into a particular way in the $smali$ folder, and the methods implemented in the source codes are in the following format after being decompiled:

$$\text{android/net/ConnectivityManager;}$$
$$-> \text{getActiveNetworkInfo}()$$

The first part android/net/ConnectivityManager presents the package of the invoked method, and the second part is the target method, getActiveNetworkInfo(), used in the app. Based on this, we can traverse all the decompiled source codes to extract employed APIs of the target app, which form the initial feature set. For all the apps in both malicious and benign app datasets, the numbers of permission-related APIs are extracted as the features to train the classifier. For each app, the feature consists of 32 304 items, where each item represents the number of *call sites* of the current API used in the target app. The features we extracted for classification could resist to the code obfuscation that does not obscure API calls of Android operating system. The manual of an official Android obfuscation tool, ProGuard [39], explicitly confirms this. Benign apps are regarded as negative samples and malicious apps as positive samples. With the extracted features, we utilize random forests to train the malware classifier, and details are illustrated in the following section.

B. Malware Classifier

Random forests, proposed by Amit *et al.* [40] and Ho [41] independently, have been widely utilized in classification and regression. To construct random forests is to train a set of decision trees [28], [42] separately, and to combine them with the voting approach. We formalize our problem as a binary classification of apps. Each app a is described by the API features $\mathbf{f} = (f_1, f_2, \ldots, f_i, \ldots, f_n)$, where f_i denotes the *call sites* of API i. In the training step, a set of labels is given to determine the type of each app, and 1 denotes malware and 0 denotes benign apps. The construction of random forests consists of a collection of decision trees and the number of decision trees is set manually. Each decision tree is constructed in a top-down fashion starting from the root. At each node of the decision tree, it splits the training set into two subsets with different labels by minimizing the uncertainty of the class labels. The uncertainty is evaluated in our classifier by computing the Gini impurity

Algorithm 1: Classifier Training.

Require:
$\mathcal{A}$: App Set
$\mathcal{F}$: Feature Set
$\mathcal{L}$: Label Set
k : Number of Decision Trees
Ensure:
$\mathcal{C}$: Random Forests Classifier
1: **function** Train $\mathcal{A}$, $\mathcal{F}$, $\mathcal{L}$, k
2: **for** $i \leftarrow 0, k$ **do**
3: $\mathcal{A}_i \leftarrow$ Randomly selected N apps
4: $\mathcal{L}_i \leftarrow$ Labels of $\mathcal{A}_i$
5: **for** each node n in decision tree $\mathcal{T}_i$ **do**
6: $\mathcal{F}_i \leftarrow$ Randomly selected m features
7: $f = \arg\max Gini(\mathcal{A}_i, \mathcal{L}_i, \mathcal{F}_i)$
8: Generate n using feature f
9: **end for**
10: **end for**
11: $\mathcal{C} = \sum_0^k \mathcal{T}_i$
12: **return** $\mathcal{C}$
13: **end function**

[43]. Specifically, the subject function is defined as follows:

$$Gini(\mathcal{A}, \mathbf{f}) = \frac{|\mathcal{A}_0|}{|\mathcal{A}|}\left(1 - \sum_{k=0,1}\left(\frac{|C_k|}{|\mathcal{A}_0|}\right)^2\right) + \frac{|\mathcal{A}_1|}{|\mathcal{A}|}\left(1 - \sum_{k=0,1}\left(\frac{|C_k|}{|\mathcal{A}_1|}\right)^2\right) \quad (3)$$

where $\mathcal{A}$ denotes the sample set containing malicious and benign apps at a specific tree node. $\mathcal{A}_0$ and $\mathcal{A}_1$ represent the subsets of $\mathcal{A}$ ($\mathcal{A}_0 \cup \mathcal{A}_1 = \mathcal{A}$) that are classified as benign and malicious apps, respectively. The feature $\mathbf{f}$ is adopted to split the sample set at the node. C_k is the sample set belonging to the class k. Each trained decision tree outputs a classified result and the final result of random forests is combined with all the results from these decision trees using voting process. The voting process is based on majority rule, i.e., selecting alternatives with more than half of the votes. Algorithm 1 shows the training process of the random forests classifier, which includes the following steps:

1) *Step 1 (lines 2–4):* Select N apps from the full app dataset $\mathcal{A}$ randomly as the initial dataset for each decision tree $\mathcal{T}_i$ of random forests.
2) *Step 2 (lines 5–9):* Let M be the size of the feature set $\mathcal{F}$. For each node n of the decision tree $\mathcal{T}_i$, m APIs are selected randomly as the feature set $\mathcal{F}_i$ to compute the Gini impurity [43], where $m \ll M$. The feature f with the min value of Gini impurity is selected as the best feature to generate the node of decision tree. It should be noted that the value of m is the same during the construction of each decision tree.

3) *Step 3 (lines 10–11):* Construct k decision trees from steps 1–2, and the result is decided by the voting approach, i.e., the type of each app is decided by the major result of all the outputs of these decision trees.

After the training process, the parameters of random forests at each node of each decision tree are set and have the capability of classifying apps. Therefore, in testing process, each app with a feature vector can be determined into a certain type with the trained malware classifier. Our full dataset is split into two parts for extracting highly sensitive APIs and training final classifier, respectively. The full set of APIs is employed as the features in the training process on the first dataset part. After the first process, we are able to extract highly sensitive APIs based on the importance of APIs, i.e., the weights of APIs learned from the first training process. Therefore, the APIs with large weights are adopted to retrain the classifier, which is subsequently used to detect malware.

The detection process is based on the trained classifier. When a new app comes, it is decompiled by *Apktool* [38]. As described above, the source codes of the app are all stored in the $smali$ folder. We extract all the permission-related APIs, including the number of *call sites* for each API, as the initial features. Based on the app database, highly sensitive APIs are able to be mined from thousands of permission-related APIs by training the random forests classifier. Therefore, only these highly sensitive APIs are utilized as the final features of the newly coming app. These features are then employed as the input to obtain the result of the app type.

VI. Evaluation

In this section, we present the evaluation metrics adopted in our experiments. With the datasets of benign and malicious apps described in Section III, comprehensive experiments are conducted in our malware detection system, MalPat. The comparison with the state-of-the-art approaches is also illustrated in this section. Details of experiments are given in the following.

A. Evaluation Metrics

To evaluate the performance of malware detection, we use *precision* and *recall* metrics. As malicious apps are positive samples and benign apps are negative samples in our evaluation, we first present three types of values:

1) *(tp*: true positive): The number of malicious apps that are correctly identified as malicious apps.
2) (*fp*: false positive): The number of benign apps that are incorrectly identified as malicious apps.
3) (*fn*: false negative): The number of malicious apps that are incorrectly identified as benign apps.

Therefore, the metrics *precision* and *recall* can be calculated as follows:

$$precision = \frac{tp}{tp + fp} \tag{4}$$

$$recall = \frac{tp}{tp + fn} \tag{5}$$

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}. \tag{6}$$

Equation (4) denotes that how many of the true malicious apps are correctly identified. The value of *precision* is in the interval of $[0, 1]$, and the large value indicates the correctness of the malware detection system. Equation (5) denotes that how many of the malicious apps identified by the detection system are true malware. The value of *recall* is also in the interval of $[0, 1]$. Equation (6) is the F_1 score, which is the harmonic average of *precision* and *recall*. The value of F_1 score is also in the interval of $[0, 1]$. In order to fight against malware, we focus on the correctness of the identification of malicious apps instead of benign apps. Therefore, *precision*, *recall*, and F_1 score of malware are adopted as evaluation metrics in the experiments.

B. Experimental Setup

In the experiments, MUDFLOW [5], DREBIN [6], and DroidAPIMiner [9] are employed as the state-of-the-art approaches to compare with our MalPat. In comparison with these methods, we select the intersection dataset of the dataset used in article [5] and our dataset, which contains 2398 benign apps and 13 840 malware samples. Therefore, our remaining dataset consists of 28 787 benign apps and 1496 malicious apps, and there is no interaction with the dataset adopted to compare with MUDFLOW, DREBIN, and DroidAPIMiner. This remaining dataset is employed in our MalPat to extract highly sensitive APIs. Except the experiments comparing with MUDFLOW, DREBIN, and DroidAPIMiner, the datasets used in the later experiments comparing with the baseline methods are our full malicious and benign datasets consisting of 15 336 malicious apps and 31 185 benign apps. Besides, in extracting highly sensitive APIs, the dataset is split into two parts with the partition of 1:1, which is illustrated in Section V. For all the experiments, we randomly select from 50% to 90% of both malicious and benign datasets as the training set, and the remaining part is regarded as the test set. We repeat each experiment for ten times and average the results. In addition, the number of decision trees trained in the random forests classifier is 200, and remains the same. In Section VI-D, we also study the impact of highly sensitive APIs, i.e., how the number of APIs used to retrain the classifier influences the final result. The partition of training and test set in the study of the impact of highly sensitive APIs is 9:1. Details of experiments and discussions are presented in the following sections.

C. Comparison With State-of-the-Art Methods

To demonstrate the effectiveness and efficiency of our automated malware detection system, MalPat, we compare it with existing state-of-the-art approaches. We test on two versions of our MalPat system: the one using the full set of 32 304 APIs (MalPat) and the one only employing top 50 highly sensitive APIs (MalPat50).

The first method we utilize to compare with MalPat is MUDFLOW [5]. We downloaded the source code from the website of MUDFLOW [44] and reran the scripts with its optimal settings on our intersection dataset. The experimental results are shown in Fig. 13. It is obvious that MUDFLOW cannot compete with MalPat under all the measures. Especially, MalPat50 outperforms MUDFLOW with 3% precision value, 2% recall

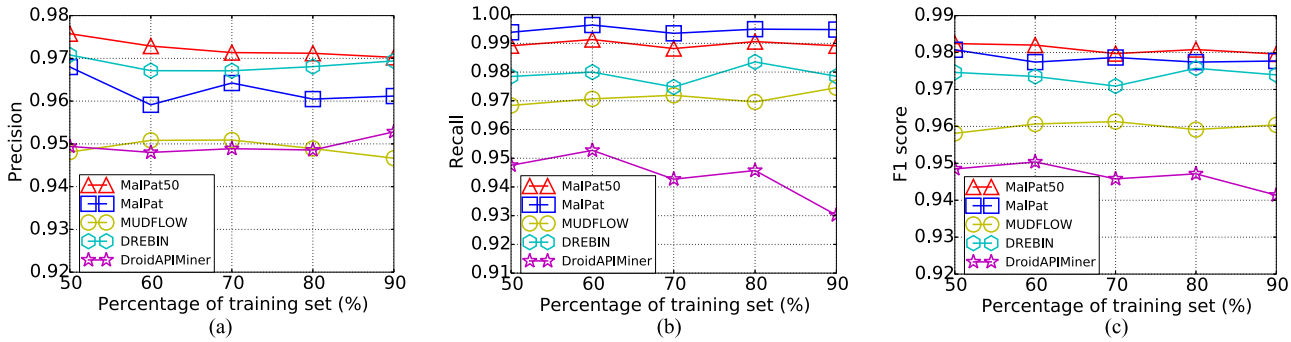

Fig. 13. Comparison with state-of-the-art methods. (a) Precision results. (b) Recall results. (c) F1 score results.

rate, and 2% F_1 score. The classifier employed in MUDFLOW is support vector machine (SVM) [45], and the features of apps are extracted according to a manually selected list. MalPat, on the other hand, takes full advantages of the random forests classifier by engaging it during the feature set construction, which manages to capture more information of disparate behaviors between benign and malicious apps.

The second state-of-the-art method to compare with is DREBIN [6], which used APIs as well as other types of features to detect malware. There are two sources of features utilized in DREBIN. The first source of feature sets is from the manifest and another one is from disassembled code. Especially, the feature sets from the disassembled code include the similar API features as we used in MalPat from PScout [26]. From Fig. 13(a), it can be observed that DREBIN's precision results are between MalPat and MalPat50, which means DREBIN has similar capability of correctly identifying malicious apps to MalPat. Its recall rates in Fig. 13(b) are all lower than both versions of MalPat, and the largest difference can be more than 0.02. The recall rate demonstrates the capability of detecting malware from millions of apps. Apparently, DREBIN is worse than MalPat on this functionality. The reason can be the same to MUDFLOW, where DREBIN uses a manually selected features, but according to the fair results of DREBIN that are better than MUDFLOW, the features employed in DREBIN do show their effectiveness in identifying malware. We have an assumption that the API features employed in DREBIN, which are similar to the ones used in our MalPat, are the key to detect malware. Thus, we modify the original DREBIN by reducing its feature sets to only the one that contains restricted API calls. We test this method on 90% training set for 10 times, and the averaged results are 0.950910 precision value and 0.972977 recall rate. These results are almost the same as the ones generated by MUDFLOW. It is unexpected but also reasonable. Because our MalPat also uses the similar API features and achieves the best results.

The last state-of-the-art approach is DroidAPIMiner [9]. It also employs Android system calls as its major features. Besides, DroidAPIMiner adds APIs with similar support whose parameters are more frequent in the malware set as well, but the later part does not make a difference on the final results. Therefore, we extract the APIs that have a usage difference of more than 6% between malicious and benign apps. These features are then employed in DroidAPIMiner to identify malware, and the results are shown in Fig. 13. Clearly, DroidAPIMiner could not compete with other approaches including MalPat. Especially, all the recall rates and F_1 scores are worse than any of other methods. We carry out the experiment on DREBIN with restricted API calls in the previous paragraph, and it is observed a surprising result. By comparing with the results of DroidAPIMiner, it can be explained that restricted API calls or permission-relate APIs do have better capability to distinguish malware from millions of apps. This proves our statement in Section I that APIs that are not permission-related may introduce noise affecting feature extraction.

By comparing with existing state-of-the-art approaches, it demonstrates the effectiveness of MalPat in identifying malware. The highest recall rate achieved by MalPat is 0.9963, which means that only 51 out of 13 840 malicious apps are not detected. In the following section, we study how different features of apps can affect the results of MalPat on our full dataset.

D. Comparison With Baseline

To study the influence of different features of apps on MalPat, we experiment on different sets of API features like the previous section, where MalPat includes the full 32 304 APIs and MalPat50 contains the top 50 highly sensitive APIs. Besides, a baseline method we compared with is the one using permissions declared in the *Manifest* file of Android platform (Perm). Permission-based methods can avoid high cost of time and computation, which, however, can only capture the coarse-grained features of Android apps. Because there are thousands of Android APIs governed by each permission. Hence, using permissions as the features of apps can miss the full view of app behaviors due to the lack of in-depth information. To demonstrate the weakness of using permissions as features, we conduct the experiments on our MalPat with both APIs and permissions, respectively.

As illustrated above, MalPat surpasses the state-of-the-art approaches, MUDFLOW, DREBIN, and DroidAPIMiner, on malware detection, so the experiments on MalPat can directly show the effectiveness of using APIs instead of permissions. Fig. 14 shows the results of MalPat and Perm methods. According to

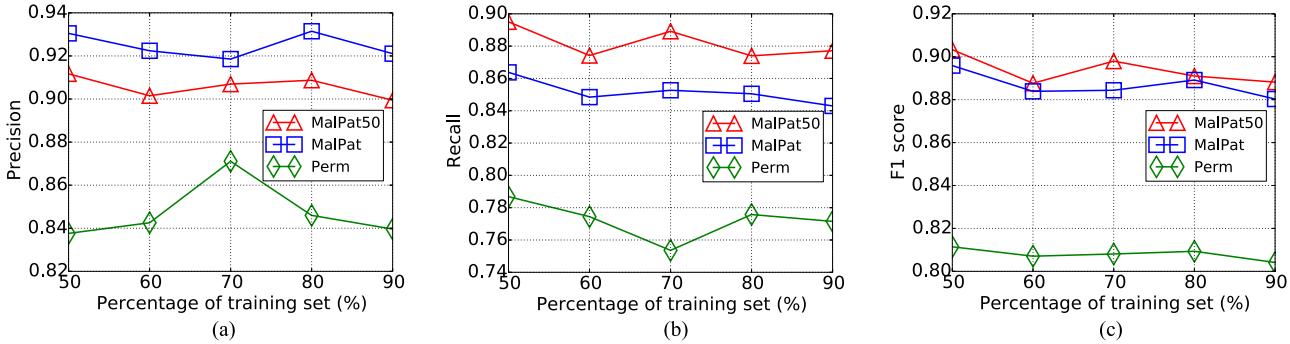

Fig. 14. Comparison with baseline methods on the full datasets. (a) Precision results. (b) Recall results. (c) F1 score results.

the results shown in Fig. 14, it is obvious that MalPat with APIs as features outperforms the one using permissions under all percentage of training set. The precision results of MalPat, shown in Fig. 14(a) are all between 0.90 and 0.94, and there is no big change with the increase of the training set. The difference between MalPat and Perm is near 0.1 when 50% of apps are used for training. There is one observation that the precision results of MalPat are better than that of MalPat50, but MalPat50 surpasses MalPat on the results of recall. It is interesting that fewer APIs employed as features can improve the recall rate, which means that highly sensitive APIs actually represent the malicious behaviors of malware and have the capability to identify malware. As observed, permissions of Android apps lack the ability to capture the fine-grained features, which can be addressed by adopting the APIs governed by them. Permission-related APIs not only capture more detailed behaviors of apps, but also have the similar features as permissions do.

E. Impact of Highly Sensitive APIs

In our model training process, we aim to extract highly sensitive APIs so as to detect malware with as few features as possible. As discussed in Section IV-B-1, different APIs have different importance in capturing malicious behaviors and identifying malware. Therefore, to study the impact of different number of APIs as features, we conduct experiments on MalPat with the number of APIs ranging from 10 to 200. As illustrated in Section VI-B, we split our full dataset into two parts with partition of 1:1. The first part is used to train the classifier to extract the importance of different APIs. In the second part, we utilize the importance of all the APIs obtained in the first part to retain MalPat. Training on different apps can ease the possibility that MalPat has the prior knowledge of features of benign and malicious apps. The results of MalPat with different number of APIs are shown in Fig. 15. According to the figure, we can see that the largest difference between the maximum and minimum values of precision is larger than 0.025, which means that the number of APIs as features has influence on the precision of MalPat. MalPat with 50 APIs as features peaks at 0.9172 of precision result, and even with only 20 APIs has the precision result of 0.9024. As to recall results, the top result is achieved with 20 APIs, and the change of recall rates of MalPat with the increase of API number is larger than that of precision results.

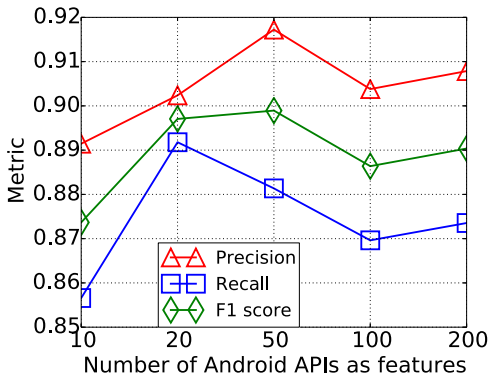

Fig. 15. Impact of highly sensitive APIs on the efficiency of MalPat.

Moreover, there is one special case that should be noted. The recall rate reaches the largest value with 20 APIs, and then it drops from 0.8918 to 0.8696 as the number of APIs increases from 20 to 200. Although there is a small increase when the number of APIs increases from 100 to 200, but from the blue line, we can observe that MalPat with full set of APIs as features has lower recall rate than the one with 50 APIs. The numbers of malicious and benign apps are both similar to the numbers of all the APIs. Therefore, if all the APIs are adopted as the features to train the classifier, it may overfit on the training set and can misclassify malicious apps. On the other hand, too few features can also affect the classification results. According the F_1 score results in Fig. 15, MalPat with 50 APIs achieves the best results, which means that this number of APIs is able to distinguish malicious and benign apps with high precision and recall. Moreover, based on the 50 highly sensitive APIs, MalPat can be trained within 1 min on the whole dataset.

VII. Conclusion

To fight against malware, we study malicious and benign Android apps in the real world to mine hidden patterns of malware. Previous research work mainly focused on permissions, sensitive resources, intents, etc., and very few efforts have been proposed for addressing the malware detection problem from API perspective. To fill this gap, we analyze the behaviors of malicious apps on API usage comparing with benign apps. Utilizing fine-grained features, we are able to mine the patterns

of malware and extract highly sensitive APIs by training the random forests classifier. To assist Android app marketplaces, we propose an automated malware detection system, MalPat. Comprehensive experiments are conducted on the large scale dataset we collected from the Internet consisting of 31 185 benign apps and 15 336 malicious apps. Compared with the state-of-the-art approaches, MUDFLOW, DREBIN, and DroidAPIMiner, MalPat outperforms them in both precision and recall. Based on the small feature set of 50 highly sensitive APIs, MalPat achieves the F_1 score of 98.24%. Experimental results show the effectiveness and efficiency of MalPat, and highly sensitive APIs mined from malicious and benign apps reflect the patterns of malware that are able to identify malware.

Despite the efficiency of MalPat, there still remains a great deal of work to improve the pattern mining and malware detection. Therefore, we will focus on the following topics in the future work. First, a larger scale dataset of apps need to be collected so as to avoid overfitting problem, which leads to the second topic. Due to the limitation of the number of existing malware collected on the Internet, mining patterns of malicious apps can be hard to expand. In order to address this problem, we will mainly focus on benign apps to study their behavior patterns to exclude malware. MalPat only considers the difference of malicious and benign apps but neglects the categories of benign apps, which may affect the identification of benign apps due to their category features. It is the third topic we are going to overcome by taking categories of benign apps into consideration in the detection of malware. In future work, we will improve the capability of MalPat and assist Android app marketplaces to fight against malware efficiently.

ACKNOWLEDGMENT

The authors would like to thank L. Huang for his assistance in building the MalPat system.

REFERENCES

[1] "McAfee Labs Threats Report March 2016," [Online]. Available: http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2016.pdf
[2] S. Rasthofer, I. Asrar, S. Huber, and E. Bodden, "How current android malware seeks to evade automated code analysis," in *Proc. 9th Int. Conf. Inf. Security Theory Practice*, 2015, pp. 187–202.
[3] "Symantec Report on the Underground Economy July 07-June 08," 2008. [Online]. Available: https://www.symantec.com/content/en/us/about/media/pdfs/Underground_Eco n_Report.pdf
[4] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," in *Proc. 21st Annu. Netw. Distrib. Syst. Security Symp.*, 2014, pp. 1–15.
[5] V. Avdiienko *et al.*, "Mining apps for abnormal usage of sensitive data," in *Proc. 37th IEEE Int. Conf. Softw. Eng*, 2015, pp. 426–436.
[6] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and explainable detection of android malware in your pocket," in *Proc. 21st Annu. Netw. Distrib. Syst. Security Symp.*, 2014, pp. 1–12.
[7] H. Peng *et al.*, "Using probabilistic generative models for ranking risks of android apps," in *Proc. 19th ACM Conf. Comput. Commun. Security*, 2012, pp. 241–252.
[8] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proc. 19th Annu. Netw. Distrib. Syst. Security Symp.*, 2012, pp. 50–62.
[9] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in android," in *Proc. 9th Int. Conf. Security Privacy Commun. Netw.*, 2013, pp. 86–103.
[10] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual API dependency graphs," in *Proc. 21st ACM Conf. Comput. Commun. Security*, 2014, pp. 1105–1116.
[11] "Google Play," [Online]. Available: https://play.google.com/store/apps
[12] "VirusShare.com," [Online]. Available: https://virusshare.com/
[13] "Contagio Mobile," [Online]. Available: http://contagiominidump.blogspot.com/
[14] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst*, vol. 32, no. 2, p. 5, 2014.
[15] Y. Zhang *et al.*, "Vetting undesirable behaviors in android apps with permission use analysis," in *Proc. 20th ACM Conf. Comput. Commun. Security*, 2013, pp. 611–622.
[16] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic android Malware analysis," in *Proc. 21st USENIX Security Symp.*, 2012, pp. 569–584.
[17] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware through static analysis," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng*, 2014, pp. 576–587.
[18] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar: Practical fine-grained decentralized information flow control," in *Proc. 30th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2009, pp. 63–74.
[19] X. Xiao, N. Tillmann, M. Fähndrich, J. de Halleux, and M. Moskal, "User-aware privacy control via extended static-information-flow analysis," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng*, 2012, pp. 80–89.
[20] O. Tripp and J. Rubin, "A Bayesian approach to privacy enforcement in smartphones," in *Proc. 23rd USENIX Security Symp.*, 2014, pp. 175–190.
[21] S. Arzt *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 259–269.
[22] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proc. 20th USENIX Security Symp.*, 2011, pp. 21–21.
[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting android apps for component hijacking vulnerabilities," in *Proc. 19th ACM Conf. Comput. Commun. Security*, 2012, pp. 229–240.
[24] A. Bartel, J. Klein, Y. L. Traon, and M. Monperrus, "Automatically securing permission-based software by reducing the attack surface: An application to android," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2012, pp. 274–277.
[25] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1025–1035.
[26] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. Comput. Commun. Security*, 2011, pp. 627–638.
[27] P.-H. Chen, C.-J. Lin, and B. Schölkopf, "A tutorial on ν-support vector machines," *Appl. Stoch. Models Bus. Ind.*, vol. 21, no. 2, pp. 111–136, 2005.
[28] J. R. Quinlan, "Induction of decision trees," *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.
[29] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proc. 19th ACM Conf. Comput. Commun. Security*, 2012, pp. 217–228.
[30] M. L. Vásquez, G. Bavota, C. Bernal-C árdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "API change and fault proneness: A threat to the success of android apps," in *Proc. 9th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2013, pp. 477–487.
[31] W. J. Conover, *Practical Nonparametric Statistics*. 3rd ed. Hoboken, NJ, USA: Wiley, 1998.
[32] "ActivityManager," [Online]. Available: http://developer.android.com/reference/android/app/ActivityManager.html
[33] "API Mappings," [Online]. Available: http://www.inpluslab.com/mappings.html
[34] "TelephonyManager," [Online]. Available: http://developer.android.com/reference/android/telephony/TelephonyManag er.html
[35] "IMSI," [Online]. Available: https://en.wikipedia.org/wiki/International_mobile_subscriber_identity
[36] S. Holm, "A simple sequentially rejective multiple test procedure," *Scand. J. Statist.*, pp. 65–70, 1979.

[37] P. Jaccard, "Etude Comparative de la Distribution Florale Dans Une Portion Des Alpes et du Jura," *Bulletin del la Socit Vaudoise des Sciences Naturelles*, vol. 37, pp. 547–579, 1901.
[38] C. Tumbleson and R. Winiewski, "Apktool," [Online]. Available: http://ibotpeaches.github.io/Apktool/
[39] "ProGuard," [Online]. Available: http://developer.android.com/tools/help/proguard.html
[40] Y. Amit and D. Geman, "Shape quantization and recognition with randomized trees," *Neural Comput.*, vol. 9, no. 7, pp. 1545–1588, 1997.
[41] T. K. Ho, "Random decision forests," in *Proc. 3rd Int. Conf. Document Anal. Recognit.*, 1995, pp. 278–282.
[42] J. Ross Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA, USA: Morgan Kaufmann, 1993.
[43] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*. Belmont, CA, USA: Wadsworth, 1984.
[44] V. Avdiienko *et al.*, "Mudflow," [Online]. Available: https://www.st.cs.uni-saarland.de/appmining/mudflow/
[45] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.

Guanhong Tao received the B.Eng. degree in computer science and technology from Zhejiang University, Hangzhou, China, in 2014. He is currently working the Master's degree in the School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China.

His research interests include mobile computing, program analysis, and software security.

Zibin Zheng (SM'16) received the Ph.D. degree in computer science and engineering from the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Sha Tin, Hong Kong, in 2010.

He is currently an Associate Professor with the School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China. His research interests include services computing, software engineering, and blockchain.

Dr. Zheng received the Outstanding Thesis Award of CUHK in 2012, the ACM SIGSOFT Distinguished Paper Award at ICSE2010, and the Best Student Paper Award at ICWS 2010.

Ziying Guo is currently working toward the undergraduate degree at the School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China.

Her research interests include mobile computing and data mining.

Michael R. Lyu (F'04) received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1981; the M.S. degree in computer engineering from the University of California, Santa Barbara, CA, USA, in 1985; and the Ph.D. degree in computer science from the University of California, Los Angeles, CA, USA, in 1988.

He is currently a Professor with the Department of Computer Science and Engineering, Chinese University of Hong Kong. He is also the Director of the Video over Internet and Wireless (VIEW) Technologies Laboratory. His research interests include software reliability engineering, distributed systems, fault-tolerant computing, mobile networks, web technologies, multimedia information processing, and e-commerce systems.

Dr. Lyu is a fellow of the ACM, AAAS, and Croucher Senior Research.