Exploring Permission-induced Risk in Android Applications for Malicious Application Detection

Wei Wang, Xing Wang, Dawei Feng, Jiqiang Liu, Zhen Han, and Xiangliang Zhang, *Member, IEEE*

***Abstract*—Android has been a major target of malicious applications (malapps). How to detect and keep the malapps out of the app markets is an ongoing challenge. One of the central design points of Android security mechanism is *permission control* that restricts the access of apps to core facilities of devices. However, it imparts a significant responsibility to the app developers with regard to accurately specifying the requested permissions and to the users with regard to fully understanding the risk of granting certain combinations of permissions.**

Android permissions requested by an app depict the app's behavioral patterns. In order to help understanding Android permissions, in this work, we explore the permission-induced risk in Android apps on three levels in a systematic manner. First, we thoroughly analyze the risk of individual permission and the risk of a group of collaborative permissions. We employ three feature ranking methods, namely, mutual information, Correlation Coefficient (CorrCoef), and T-test to rank Android individual permissions w.r.t. their risk. We then use Sequential Forward Selection (SFS) as well as Principal Component Analysis (PCA) to identify risky permission subsets. Second, we evaluate the usefulness of risky permissions for malapp detection with Support Vector Machine (SVM), decision trees as well as random forest. Third, we in depth analyze the detection results and discuss the feasibility as well as the limitations of malapp detection based on permission requests. We evaluate our methods on a very large official app set consisting of 310,926 benign apps and 4,868 real-world malapps and on a third-party app sets. The empirical results show that our malapp detectors built on risky permissions give satisfied performance (a detection rate as 94.62% with a false positive rate as 0.6%), catch the malapps' essential patterns on violating permission access regulations, and are universally applicable to unknown malapps (detection rate as 74.03%).

***Index Terms*—Android security; permission usage analysis; malware detection; intrusion detection**

I. Introduction

Smartphones and mobile devices have become explosively popular for personal or business use in recent years. As reported by Digitimes research [1], global smartphone shipments are expected to reach around 1.24 billion in 2014. This number has increased 30% over the last year. Meantime, smartphone platforms have seen a massive surge in malwares. With Android accounting for 81 percent of all smartphone shipments globally in the third quarter of 2013 [2], it has unsurprisingly become the major target for mobile malware. The volume of Android malware families and samples has been growing explosively. Symantec [3] indicated that the number of known malware samples increased almost four times between June 2012 and June 2013 and was up to about 273,000.

As the official application (or app) market, Google's Play store provides a platform of delivering apps for Android smartphones and mobile devices. There are many third-party app markets providing similar platforms. App developers publish their apps on the Google's play or on the third-party app markets, where end users download and install their interested apps on their Android smartphones. Obviously, how to detect and keep the large number of malware out of the application (or app) markets is an emerging, crucial, but challenging issue. Previous work on the detection of malapps mainly focused on permissions [4], [5], [6], static [7], [8], [9], [10], [11], [12], [13] and dynamic analysis [14], [15], [16], [17].

Permission control is one of the major Android security mechanisms. Android permissions provide fine-grained security features by enforcing restrictions on the specific operations that a particular process can perform [18]. However, it imparts a significant responsibility to the app developers with regard to declaring the least-privileged set of permissions needed by designed apps, and to the app users with regard to fully understanding the risk of granting certain combinations of permissions. Android provides developers documentation, but its permission information is limited. On the one hand, the lack of reliable permission information may let developers request unnecessary permissions, resulting in overprivileged applications [19] that users may cancel the installation. In addition, the unnecessarily risky permissions may be leaked to other malapps [20], leading to the permission re-delegation attacks [21]. On the other hand, the lack of risk information of permissions confuses the users with regard to determining whether to install the app or not. Current Android permission warnings do not help most users make correct security decisions [22]. It is feasible to identify malapps through analyzing the permission usage patterns, as intuitively an app's behavior is characterized by the permissions it requests. We thus see that exploring the permission-induced risk is beneficial to three parties, the Android app developers, the users, as well as the malapp detectors. Curiosities are aroused on understanding the following questions: (1) what is the ranking of the permissions

W. Wang, X. Wang, J. Liu and Z. Han are with the School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044 China (e-mail: wangwei1@bjtu.edu.cn; 10112071@bjtu.edu.cn; jqliu@bjtu.edu.cn; zhan@bjtu.edu.cn). The work reported in this paper is supported by the Fundamental Research Funds for the Central Universities of China (No. K13JB00160, No. 2012JBZ010), the Ph.D. Programs Foundation of Ministry of Education of China (No. 20120009120010), the Scientific Research Foundation for the Returned Overseas Chinese Scholars, State Education Ministry (No. K14C300020), the Program for New Century Excellent Talents in University (NCET-11-0565), and the 111 Project (No. B14005).

D. Feng is with Laboratoire de Recherche en Informatique, CNRS, Université Paris-Sud, France (e-mail: dawei.feng@lri.fr).

X. Zhang (corresponding author) is with Division of CEMSE, King Abdullah University of Science and Technology (KAUST), Saudi Arabia (e-mail: xiangliang.zhang@kaust.edu.sa).

with respect to (w.r.t.) the risk to the Android system; (2) what is the subset of permissions that collaboratively cause security issues in malapps; (3) to what degree the Android malapps can be detected based on the permissions they requests; and (4) whether there exist fine-grained permission rules that can be used to identify unknown malapps (zero-day malapps), like the 9 detection rules with permissions called Kirin [23].

We are motivated to answer the above questions about the permission system of Android, in the vision of risk evaluation of Android permissions based on systematically quantitative analysis of Android apps on a very large scale (we collected 310,926 free apps from Google's play and 4,868 real-world malapps). To fulfill the goal of exploration, our study is performed on the following three levels. First, we systematically analyze the risk of each individual permission and the risk of a group of collaborative permissions by employing machine learning techniques, such as feature ranking with mutual information, Correlation Coefficient (CorrCoef) and T-test, subset selection and transformation with Sequential Forward Selection (SFS) as well as Principal Component Analysis (PCA). Second, we evaluate the usefulness of risky permissions for malapp detection using classification algorithms, suck as Support Vector Machine (SVM), decision tree as well as Random Forest. Last but not least, we discuss and analyze in depth the feasibility as well as the limitations of malapp detection based on permissions requests.

The main contributions of this work are summarized level by level as follows:

- We systematically rank the permissions w.r.t. their risk to the Android system. Individual Android permissions are ranked regarding their risk-relevance measured by mutual information, CorrCoef and T-test. We also identify the subset of risky permissions that collaboratively cause security issues with SFS and PCA. This helps to monitor the misuse of risky permissions in practice, not only for the app users, but also for the app developers.
- We evaluate the feasibility of using permission requests for malapp detection with different subsets of risky permissions and classification algorithms like SVM, decision tree and random forest. We report top-40 risky permissions that can achieve a malapp detection rate as 94.62% with a false positive rate as 0.6%. We also construct a set of detection rules that catch the malapps' essential aspects on violating permission access regulations. They are able to detect unknown malapps with a detection rate of 74.03%.
- Based the empirical results on a very large scale, we comprehensively discuss and analyze the effectiveness as well as the limitations of malapp detection based on permission requests. The analysis provides a perspective regarding the use of permission requests for the analysis of Android applications.
- We collected a very large data set that consists of 310,926 benign apps and 4,868 malapps for the evaluation. We publish the permission vectors extracted from the data set on our website as a potential benchmark data for the research community.

As complementary parts to this introduction, Section II gives the background knowledge of Android permissions, while Section VII discusses related work on permission analysis and malapp detection techniques with permissions. Section III describes the data. Section IV describes the methodology of risk exploration. The extensive experiments are given in Section V. Section VI presents in-depth discussion and analysis of our findings and conclusion follows in Section VIII.

II. Background

In this section we first briefly describe the Android's security mechanisms and then elaborate the design of Android's permission control in the mechanism.

Android platform includes a multi-user operating system based on a Linux kernel, middleware, and a set of applications (apps). Users install apps acquired from app markets, e.g., official Google's play or alterative app markets. Android implements a number of security mechanisms of which the most prominent includes app sandbox and a permission framework that enforces access control to core functionalities. App sandbox is set up in a kernel lever [24]. It enforces security between apps and the system through identifying and isolating app resources. Each Android app is assigned a unique User ID (UID) and run as the user in a separate process. Under the app sandbox mechanism, apps cannot interact with each other and an app has limited access to the operating systems. While Android apps are mainly programmed in Java, native codes can also be integrated with Java apps. All types of apps, including Java, native or the hybrid are sandboxed in the same way and thus have the same degree of security [24].

One of the central design points of the Android security mechanism is permission control. As Android sandboxes apps from each other, apps must explicitly declare the permissions they need for additional capacities. Without a permission, an application by default is not able to do anything that could adversely impact the user experience, the network, or data on the device. The Android app developer statically declares the permissions the app requests in a manifest file (AndroidManifest.xml). When a user installs an app, a dialog will be displayed to indicate a permission list the app requests and asks the user whether to continue the installation. This is an all-or-nothing decision. If the user decides to install the app, all the requested permissions will be granted. The user is not able to grant or deny individual permission. The permissions are applied once the app is installed. The user will no longer be notified of the permissions granted in the running the app. While there exist some third-party apps (or rooting the device) that can help to manage the permissions on a per-app basis, normally if the user wants to block the permissions granted at the install time, the user needs to remove the app.

Android permissions provide a mechanism of access control to core facilities of the system. However, it imparts a significant responsibility to both the app developers and the app users. The developers need to accurately specify the requested permissions and the users need to understand the risk involved and thus make a rational decision regarding whether install the app or not. Ideally, the developer should follow the least-privileged set of permission requests and the user should understand the risk of granting certain combinations of permissions. Android itself provides an attribute called "protectionLevel" that characterizes the potential risk implied in the permissions [25]. The permission attributes can be categorized as "normal", "dangerous", "signature" and "signatureOrSystem". The last two attributes are system-granted only. The permission with the normal attributes is lower-risk (e.g., SET_WALLPAPER) and will be automatically granted, without asking for the user's explicit approval. The permission tagged as dangerous is higher-risk (e.g., READ_SMS) that would access to private user data or control over the device. However, the two permission categorizations provided by Android are very coarse for both the developers and the users. In this work, our goal is to systematically rank the permissions w.r.t. their risk for the users to have a better understanding of permissions, and to identify a subset of risky permissions that are most relevant to malapps for the developers to accordingly decide how to declare the permission requests. In addition, we are motivated to analyze and detect malapps with permission matrix and construct a decision rule set to universally detect unknown malapps. Our work would provide a whole picture of the relationships between the permission usage and their risk in Android apps, and a vision regarding the use of only permissions for the analysis and detection of malapps.

III. Data sets

In order to conduct extensive analysis on permission usage, we need to collect a large well-labeled app set. For benign apps, we collected a total number of 310,926 free apps from Google's play in June 2013 after removing malapps reported by an antivirus tool and by a free online service named *VirusTotal* [26] that helps to detect malware by antivirus engines (normally over 40) and website scanners for further examination.

Although a great number of malicious app samples have been reported, the collection of malapps is still a challenging task for research. Fortunately, we have been provided with two malicious app sets (named *Mal_Com1* and *Mal_Com2*) from two different antivirus companies. We got the malicious apps discovered by Zhou et al. [10] and named them as *Mal_Zhou*. In addition, we downloaded a total number of 3,417 malicious apps (named *Mal_VS*) from the website of *VirusShare* [27] that is a repository of malware samples. All the malapps in the *Mal_VS* were approved by *VirusTotal* [26]. After going through the *Mal_VS*, we found that there are duplicate samples that have been included in *Mal_com1*, *Mal_com2* and *Mal_Zhou*. After removing the duplicate samples, we have a total number of 3,207 malapps in *Mal_VS*.

In this work, we only consider the permissions provided by Android system, although an app can also define its own permissions. To analyze the permission usage of apps, we mainly extract the Android permission list from the Manifest file of each app. The total number of distinct permissions requested by all the apps (including benign and malicious) in our data sets is 135. However, the permissions requested by an app may be over-privileged, since 47 out of the 135 permission (e.g., permission INSTALL_PACKAGES) are not for use by third-party applications [28]. We then remove these 47 permissions and the total number of distinct permissions is thus 88. Therefore, each app can be represented by a 88-dimensional Boolean vector, where 1 denotes that the app requests the permission and 0 otherwise. The vectors are then sent as the data input to the methods described in next Section for analysis.

It is not rare that different apps request the same set of permissions. The number of distinct permission vectors accounts for 9.25% of the total number of apps. However, duplications imply the popularity of permission sets, which is a part of permissions' nature. Our exploration of permissions is thus based on the full vector sets and we leave the discussion of its impact in Section VI. The statistics of data[1] used in our work are given in Table I.

IV. Methods

In this section, we introduce our methodology for exploring permission-induced risk in Android apps. First, we employ three feature ranking techniques to evaluate the risk of granting each permission, based on which the permissions are ranked from most to least risky. Second, permission sets, instead of individual permission, are evaluated by feature subset selection methods for investigating the risk introduced by the collaboration of several permissions. Third, the detection of malapps based on risky permissions is formulated as a classification problem and executed by building classifiers. Last, in order to explicitly characterize the risk caused by permission requests and use it to report malapps, detection rules are extracted from malapps detectors. We then employ the detection rules to detect unknown malapps. The process of our methods is illustrated in Figure 1.

A. Ranking Permissions w.r.t. Risk

Permissions can be considered as features that describe the functionalities of apps, or the app's behavior indicating its attempt to interact with the system, the data or other apps, as described in Section II. The malapps are basically different from benign apps on requesting different permissions, i.e., having different values of permission/feature variables. Defining a class variable that indicates the label of an app, *benign* or *malicious*, the risk of granting a permission can be evaluated by measuring the relevance between this permission/feature variable and the class variable. A strong correlation of them indicates the high risk of granting this permission.

Measuring the relevance of a feature and class variables is known as feature ranking in machine learning, which has a goal of selecting the most informative features and improving the performance of learned models [29]. In this paper, we employ three ranking methods, namely, mutual information, Pearson correlation coefficient (CorrCoef), and T-test. We introduce the three ranking methods respectively after giving the formal notation.

[1]The permission vector sets used in this work are available to download at http://infosec.bjtu.edu.cn/wangwei/?page_id=85. The malapps are available upon request to the first author.

TABLE I
THE NUMBER OF BENIGN APPS AS WELL AS MALAPP SAMPLES IN OUR STUDY

Data set	# of Benign apps	# of Malapps				# of Overall
	Google's play	Mal_Zhou	Mal_Com1	Mal_Com2	Mal_VS	
Full	310,926	1,260	247	154	3,207	315,794
Distinct	28,536	332	76	31	504	29,216

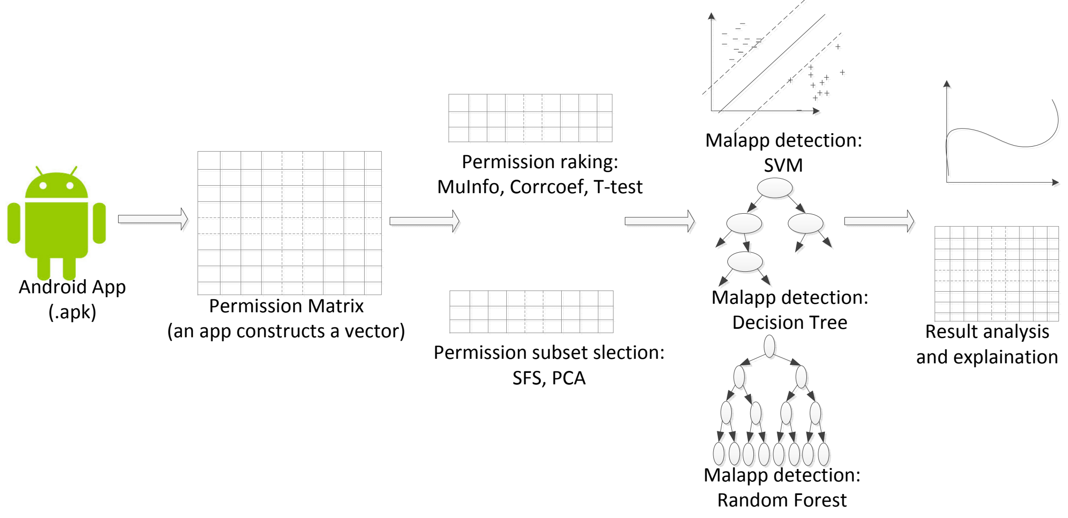


Fig. 1. The methods and process for exploring permission-reduced risks and the detection of malicious applications in three levels.

Mutual Information. Let X denote a permission variable and C be the class variable. The relevance of X and C can be measured by mutual information of them as

$$I(X,C)=\sum_{x_i}\sum_{c_j}P(X=x_i,C=c_j)\log\frac{P(X=x_i,C=c_j)}{P(X=x_i)P(C=c_j)} \tag{1}$$

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 permission 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 C has binary values, c_0 for *benign* apps and c_1 for *malicious* apps. Each permission X is a boolean variable with value 1 or 0. $I(X,C)$ 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.

Pearson CorrCoef. Pearson CorrCoef measures the relevance of X and C by

$$R(X,C)=\frac{cov(X,C)}{\sqrt{var(X)var(C)}} \tag{2}$$

which in our case of binary class and boolean variable becomes

$$R(X,C)=\frac{\sum_{n=1}^{N}(X_n-\bar{X})(C_n-\bar{C})}{\sqrt{\sum_{n=1}^{N}(X_n-\bar{X})^2\sum_{n=1}^{N}(C_n-\bar{C})^2}} \tag{3}$$

where $\bar{X}$ (resp. $\bar{C}$) is the average of all sample values of X (resp. C), X_n (resp. C_n), $n=1...N$. $R(X,C)$ has a value in $[-1,\ 1]$, where $R(X,C)=0$ indicates the independency of X and C, $R(X,C)=1$ indicates the strongest positive correlation of them and $R(X,C)=-1$ indicates the strongest negative correlation. In this paper, $R(X,C)=1$ means that permission request of X makes apps highest risky, while $R(X,C)=-1$ means that permission request of X makes apps lowest risky.

T-test. T-test is similar to CorrCoef. Given a variable, it measures the statistical significance of its value difference between two classes. Let N_0, μ_0 and σ_0 be the number, the mean and the standard deviation of X in *benign* samples (with class $C=c_0$), respectively, and N_1, μ_1 and σ_1 be the number, the mean and the standard deviation of X in *malicious* samples (with class $C=c_1$), respectively. The null hypothesis is that X and C are independent, i.e., $\mu_0=\mu_1$. T-test is performed by

$$t=\frac{\mu_0-\mu_1}{\sigma_{01}\sqrt{\frac{1}{N_0}+\frac{1}{N_1}}}\ \text{with}\ \sigma_{01}=\sqrt{\frac{(N_0-1){\sigma_0}^2+(N_1-1){\sigma_1}^2}{N_0+N_1-2}} \tag{4}$$

The absolute statistic value t can be used to indicate the correlation of X and C. A large value shows a strong correlation. The p-value of t can be somehow considered as the false positive rate on reporting the correlation. A threshold, e.g., 0.05, is usually set to reject the null hypothesis. The correlation with a p-value lower than the given threshold is considered statistically significant.

For each permissions X_i, its relevance to class C can be evaluated by the above-mentioned methods. We are especially interested in the permissions that are strongly correlated with C. Thus, under each criteria, permissions are ranked according to the value of $I(X_i,C)$, $|R(X_i,C)|$ or $|t_i|$ in decreasing order.

The top permissions are the most sensible ones that malapps often manipulate.

B. Identifying Risky Permission Sets

The ranking from individual evaluation in the previous subsection helps on selecting the most relevant permissions for distinguishing malapps from benign apps. In this subsection, we identify the risky permission subsets that are risky either because of their combinations or their cooperations with each other.

An interesting permission set is supposed to be useful for reporting malapps. We thus employ feature subset selection methods to identify such risky permission sets. Feature subset selection searches for the best combination of feature subsets that can achieve the optimal prediction performance by using the least number of features. However, it is not practical to search the whole space of $2^M - 1$ possible feature subsets given M features ($\sum_{k=1}^{M} \binom{M}{k}$), which in our case is $M = 88$. The search strategies usually used in feature selection include exhaustive enumeration (when M is small), forward selection (bottom-up), backward elimination (top-down), and best first (forward/backward with a stopping criterion) [30]. In this paper, we employ sequential forward selection (SFS) as well as Principal Component Analysis (PCA) to for feature subset selection.

Sequential Forward Selection (SFS). SFS sequentially adds features to an empty candidate set until the addition of further features cannot improve the prediction performance. In this greedy algorithm, the feature added at each step is the one from all unselected ones, which can best improve the predictive power. Thus, the selected subset contains features that are highly correlated with the class variable but uncorrelated with each other.

Principal Component Analysis (PCA). Both above feature ranking and subset selection algorithms are concerned with analyzing original features. Feature extraction algorithms construct a set of new features by applying either linear or non-linear functions on the original features. New features can reveal latent variables that better clarify relationships among studying objects. Principal Component Analysis (PCA) is a method for representing original data by new-defined variables. It aims at finding a set of orthogonal (uncorrelated) Principal Components (PCs) from the original variable space. PCs are linear combinations of the original variables. Constructing a N by M data matrix $\mathbf{X}$ whose columns are permissions X_i, $i = 1...M$, PCs are actually eigenvectors of the gamma matrix of data $\mathbf{X}$, defined as

$$\Sigma = \frac{1}{N}(\mathbf{X} - \bar{\mathbf{X}})^{\mathbf{T}}(\mathbf{X} - \bar{\mathbf{X}}) \tag{5}$$

where $\bar{\mathbf{X}}$ is the vector containing the mean of each permission.

For each PC, its importance is measured by its corresponding eigenvalue, which indicates the variance it captures. The most important PC captures the largest variance in the data, the second most important PC captures the second largest variance in the data, and so on. PCs with low importance can be eliminated. The remaining PCs then represent $\mathbf{X}$ in a lower dimensional space than the original one, but capture the underlying pattern with little loss. In this paper, we employ PCA to select the top k-PCs (k<88) that can represent the original permission set.

C. Building Detectors of Malapps Using Risky Permissions

From the above two subsections, we obtain 1) top-k permissions that are most relevant to class label of apps; 2) k-permission sets that include k permissions cooperating to make apps risky; and 3) k PCs representing apps in a new space. In order to distinguish malapps from benign apps, classifiers are built on the basis of these identified risky permissions. We employ three classification algorithms, Support Vector Machine (SVM), Decision Tree (DTree) and Random Forest (RF), due to their good performance in prediction accuracy.

Support Vector Machine (SVM). SVM seeks the best hyperplane that separates data objects from one class on one side, while others on the other side. The optimal separating hyperplane is defined as the one resulting in the maximal margin. Finding the maximal margin separating hyperplane is formulated as a quadratic programming problem with the help of Lagrange multipliers and duality [31].

Decision Tree (DTree). DTree [32] learns a classification model with a tree structure, where nodes are features/permissions, leaves are determined class labels and branches from nodes to nodes or nodes to leaves are associated with specified conditions, e.g., feature at parent node has a specified value. Training objects are all at the root in the beginning, and then are partitioned recursively based on selected features. The selection of features is on the basis of a heuristic or statistical measure. The selected best feature at each recursive step maximally reduce the uncertainty for classification. Therefore, DTree algorithm inherently possesses the function of feature selection.

Random Forest. Random Forest [33] is an ensemble of a set of decision trees independently learned on reduced training sets. A reduced training set is formed by randomly sampling with replacement from both features and objects. The final decision of classification is made by voting among all learned trees. Like other ensemble methods, Random Forest often outperforms a single tree on classification accuracy.

D. Extracting Rules for Explicitly Outlining MalApps

Rules represent the knowledge in the form of IF-THEN, which is easy for users to interpret. A rule IF (*Condition*) - THEN (*c*) makes reasoning explicit, where *Condition* is a conjunctions of feature-value pairs and *c* is the class label. For example, IF (*BloodType=Warm* & *LayEggs=Yes*) - THEN (*AnimalClass=Birds*).

In this paper, we are especially interested in rules because rules with permissions as condition and *malicious* as class label explicitly define the profile of malapps. Nine permission rules are used in [23] for the detection of malapps. Unlike these 9 rules formed mainly based on the security requirement engineering, our extracted rules are from empirically quantitative analysis of permissions requested by a very large app sets, and are thus more comprehensive and larger on scale. Due to its easiness of interpretation, these rules can be further used for reporting malapps in a straightforward manner, with no requirement on users's classification expertise. Among the three learned classifiers, DTree is more interpretable than others. Rules are therefore extracted from well-founded decision tree classifiers.

Given a decision tree, one rule is created for each path from the root to a leaf. Each feature-value pair along a path forms a conjunction, while the leaf holds the class prediction. A set of rules extracted in such a way contains as much information as the tree. These rules are mutually exclusive (rules are independent and each data object is covered by at most one rule) and exhaustive (each object is covered by at least one rule).

V. Experimental results

This section reports the results of permission ranking, permission subsets identified, malapps detector performance evaluation, and the extracted explicit decision rules.

A. Permissions Ranking

We use Mutual Information (MuInfo), CorrCoef as well as T-test methods for ranking the permissions w.r.t. their capability of discriminating malapps from benign apps. As the permission ranking needs benign apps as well as malapps, in the experiments, we use all the app data for permission ranking. The ranking results for the top 40 risky permissions are presented in Table II. It is observed that CorrCoef and T-test produce the same top risky set, although the ranking order for the first 10 permissions is different. The number of intersection of the top 40 risky permissions generated by MuInfo and by CorrCoef is 39, indicating that the ranking results are consistent. The only one different permission in the ranking results is shown in bold.

Figure 2 shows the occurrence rate of each top ranked permission with CorrCoef in the benign apps as well as in the malapps. It is clear that the top risky permissions well discriminate malapps from benign ones by the frequency of their appearance. The SMS related permissions are consistently ranked very top by the three ranking methods. The difference of occurrence rate between benign apps and malapps is above 50% for permissions READ_SMS, RECEIVE_SMS, and SEND_SMS, and is more than 15% for permission WRITE_SMS. We thus see that the usage pattern of SMS related permissions is quite different between the benign apps and the malapps and many malapps attempt to request SMS related permissions. This is consistent with the recent report [34] indicating that the total number of premium service abuser and data stealer accounts for 68% mobile threats. SMS-related activities mainly contribute to these two types of dominant threats. INSTALL_PACKAGES is also a risky permission that is more likely requested by malapps. The permission INTERNET is a sensitive permission. However, it is not ranked on the top, as it is a common permission often requested by both benign apps and malapps.

Android provides "protectionLevel" that categories permissions as "normal", "dangerous", "signature" and "signatureOrSystem" [25]. The permissions attributed as "signature" and "signatureOrSystem" are system-granted only and are not considered in our study. Our ranking results for "normal" and "dangerous" thus provide fine-grained risks for permission requests. While our ranking results are roughly consistent with the risk signals that Android "protectionLevel" identifies (accounts for 31/40), there exist differences between the two risk indicators. First, a number of "normal" permissions appear in our top-40 risky permission lists, as underlined in Table II. As Android permissions evolve with the new versions of Andriod, we use GingerBread (Android version 2.3.7) as an example to illustrate the main differences. There are 9 "normal" permissions appearing in our top-40 risky permission list. For example, permission SET_ALARM is attributed as "normal" by Android. However, it is ranked 9th in top-40 risky permissions in our analysis. With in-depth analysis, we find that there are only 238 benign apps requesting permission SET_ALARM (account for 0.077%) that is requested by 221 malapps (account for 4.54%). Most of these 221 malapps induce users to send SMS that is triggered by an alarm (clock), in order to get profit with the unawareness of users. Another example is permission RECEIVE_BOOT_COMPLETED. While it is attributed as "normal", it is requested by 35.6% of malapps in order to re-boot the system to trigger the malicious behaviors. Our study indicates that even "normal" permissions can be exploited by malapps. Second, a number of "dangerous" permissions do not appear in the top-40 risky list. For example, in GingerBread, 19 out of the total 46 "dangerous" permissions are not ranked as top-40 by our study. For example, permission SET_TIME_ZONE is attributed as "dangerous". However, it does not indicate risk and thus ranked as 80/88 in our study, since it is never exploited by any malapp in our study.

Android intuitively attributes a permission by considering whether it can access directly the resources of the system. However, "normal" permissions can be exploited by sophisticated malapps, as discovered by our work that ranks the risk of permissions based on the quantitative analysis of benign apps as well as malapps on a very large-scale with machine learning techniques. Our study thus provides empirical view on the risks of Android permissions.

B. Risky Permission Sets

As introduced in Section IV, we can obtain risky permission subsets in three different ways, 1) taking top-k permissions from the ranking results; 2) selecting k-permission sets by SFS; and 3) extracting k PCs by PCA method. Varying k, we have risky permission sets in different sizes. In order to evaluate the capability of different risky permission sets for malapps detection, we apply SVM with RBF kernel on three ranking sets, one SFS subset and one PCs set. The classification results on these five different permission sets are compared on Accuracy (ACC) and F-score.

Accuracy is the proportion of correctly classified data object (both True Positives (TP) and True Negatives (TN)) in the population (that is, the sum of TP, TN, as well as False Positive (FP) and False Negative (FN)):

$$ACC = \frac{TP + TN}{TP + TN + FP + FN}. \tag{6}$$

TABLE II
TOP 40 RISKY PERMISSIONS RANKED BY CORRCOEF, MUTUAL INFORMATION AND T-TEST.

Rank	Corrcoef		Mutual Information	T-test
	Score	CorrCoef		
1	0.4428	READ_SMS	SEND_SMS	RESTART_PACKAGES
2	0.42	RECEIVE_SMS	RECEIVE_SMS	READ_SMS
3	0.3961	SEND_SMS	READ_SMS	READ_PHONE_STATE
4	0.1988	WRITE_SMS	READ_PHONE_STATE	SEND_SMS
5	0.1443	SET_ALARM	READ_EXTERNAL_STORAGE	READ_EXTERNAL_STORAGE
6	0.1403	RECEIVE_WAP_PUSH	WRITE_SMS	SYSTEM_ALERT_WINDOW
7	0.114	READ_PHONE_STATE	SET_ALARM	SET_ALARM
8	0.1044	READ_EXTERNAL_STORAGE	RECEIVE_BOOT_COMPLETED	RECEIVE_WAP_PUSH
9	0.0804	RESTART_PACKAGES	RECEIVE_WAP_PUSH	WRITE_SMS
10	0.0711	SYSTEM_ALERT_WINDOW	RESTART_PACKAGES	RECEIVE_SMS
11	0.0668	RECEIVE_BOOT_COMPLETED	WAKE_LOCK	RECEIVE_BOOT_COMPLETED
12	0.063	CHANGE_WIFI_STATE	ACCESS_NETWORK_STATE	CHANGE_WIFI_STATE
13	0.0611	WAKE_LOCK	WRITE_EXTERNAL_STORAGE	WAKE_LOCK
14	0.0562	DISABLE_KEYGUARD	INTERNET	DISABLE_KEYGUARD
15	0.0553	ACCESS_NETWORK_STATE	CHANGE_WIFI_STATE	ACCESS_NETWORK_STATE
16	0.0551	WRITE_SETTINGS	SYSTEM_ALERT_WINDOW	WRITE_SETTINGS
17	0.0535	READ_CONTACTS	READ_CONTACTS	READ_CONTACTS
18	0.053	RECEIVE_MMS	READ_CALL_LOG	RECEIVE_MMS
19	0.0506	WRITE_EXTERNAL_STORAGE	WRITE_SETTINGS	WRITE_EXTERNAL_STORAGE
20	0.045	EXPAND_STATUS_BAR	DISABLE_KEYGUARD	EXPAND_STATUS_BAR
21	0.0444	WRITE_CONTACTS	WRITE_CONTACTS	WRITE_CONTACTS
22	0.0415	CHANGE_NETWORK_STATE	CAMERA	CHANGE_NETWORK_STATE
23	0.0413	INTERNET	WRITE_CALL_LOG	INTERNET
24	0.0365	READ_HISTORY_BOOKMARKS	CHANGE_NETWORK_STATE	READ_HISTORY_BOOKMARKS
25	0.0346	CHANGE_CONFIGURATION	ACCESS_WIFI_STATE	CHANGE_CONFIGURATION
26	0.0344	PROCESS_OUTGOING_CALLS	RECEIVE_MMS	PROCESS_OUTGOING_CALLS
27	0.0339	GET_PACKAGE_SIZE	READ_HISTORY_BOOKMARKS	GET_PACKAGE_SIZE
28	0.0338	PERSISTENT_ACTIVITY	EXPAND_STATUS_BAR	PERSISTENT_ACTIVITY
29	0.0334	ACCESS_WIFI_STATE	GET_ACCOUNTS	ACCESS_WIFI_STATE
30	0.0329	READ_CALL_LOG	CHANGE_CONFIGURATION	READ_CALL_LOG
31	0.0309	CAMERA	PROCESS_OUTGOING_CALLS	CAMERA
32	0.0287	WRITE_HISTORY_BOOKMARKS	CALL_PHONE	WRITE_HISTORY_BOOKMARKS
33	0.0273	CALL_PHONE	WRITE_HISTORY_BOOKMARKS	CALL_PHONE
34	0.0252	SET_WALLPAPER_HINTS	GET_PACKAGE_SIZE	SET_WALLPAPER_HINTS
35	0.0249	GET_ACCOUNTS	GET_TASKS	GET_ACCOUNTS
36	0.0237	GET_TASKS	ACCESS_MOCK_LOCATION	GET_TASKS
37	0.0232	WRITE_CALL_LOG	PERSISTENT_ACTIVITY	WRITE_CALL_LOG
38	0.019	**ADD_SYSTEM_SERVICE**	ACCESS_FINE_LOCATION	ADD_SYSTEM_SERVICE
39	0.0182	ACCESS_FINE_LOCATION	SET_WALLPAPER_HINTS	ACCESS_FINE_LOCATION
40	0.0168	ACCESS_MOCK_LOCATION	**USE_CREDENTIALS**	ACCESS_MOCK_LOCATION

F-score is a compromised measure between Precision and Recall, where Precision can be referred as Positive predictive value (defined as $Precision = \frac{TP}{TP+FP}$), and Recall is also referred to as the TPR (calculated by $Recall = \frac{TP}{TP+FN}$). It is defined as

$$F\text{-}score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}. \tag{7}$$

F-score is usually used to evaluate the performance of unbalanced binary classification problem, like our malapps detection problem involving a relatively small number of malapps comparing to that of benign apps. It can be interpreted as a weighted average of the precision and recall. A F-score close to 1 indicates the good performance on correctly classifying the minority class, which in our case is the malapp.

Decision Tree (DTree) and Random Forest (RF) inherently possesses the function of feature selection. In other words, they perform feature selection and classification simultaneously. We therefore compare their results with that of five SVM outputs. In order to have a reliable performance measure and fair comparison, all results reported in this paper are from the implementation with a 5-fold cross-validation. In detail, we randomly partition the malapp set as well as the benign app set into 5 equal size of subsets. 4 subsets from malapps and another 4 subsets from benign apps are retained for training the models. The remaining single subset from malapps and single subset from benign apps are used for the testing. The cross-validation process is then repeated 5 times, with each of the 5 subsets used exactly once as the test data. The performance measures from the 5 folds are then averaged to produce a single estimation. Varying the number k of permissions used for detecting malapps, we show the ACC and F-score obtained with 7 methods in Figure 3-4.

From Figure 3, we observe that the detection accuracy for most methods almost reaches the best when around 40 permissions are used for building the models, except Random Forest and DTree. DTree maintains a stable rate when over 40 permissions are used. Random Forest has the best performance when 10 permissions are used for sampling and tree construction. This is explainable because Random Forest is an ensemble method and prefer to have a set of trees as diverse as possible. The results of F-score are consistent with those of accuracy. The top 40 permissions listed in Table

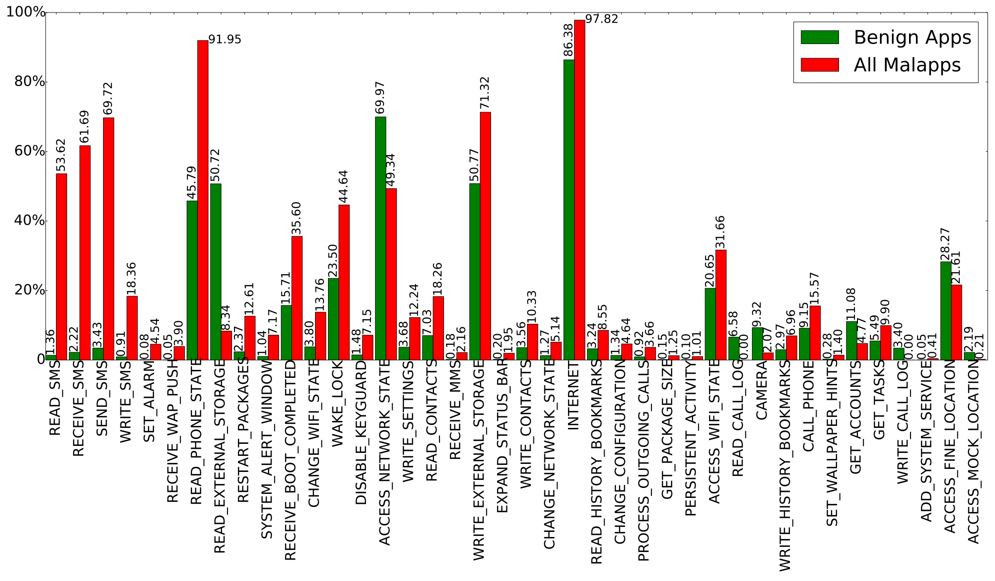


Fig. 2. Occurrence percentage of top 40 ranked risky permissions (with CorrCoef) in benign apps and malapps. The permissions are ranked from the left to the right in the figure.

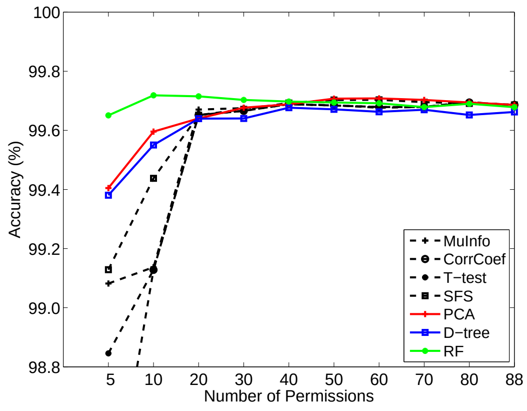


Fig. 3. The detection accuracy of 7 different methods when increasing the number of permissions.

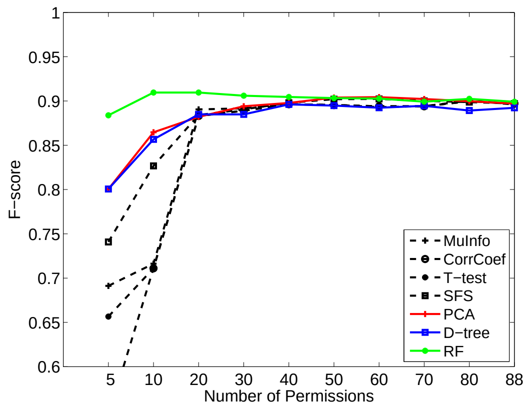


Fig. 4. The F-score of 7 different methods when increasing the number of permissions.

2 and the 40-permission sets by SFS can be considered as the most useful risky subset, as they have better performance than permissions sets with other k values, considering that we normally tend to select the smallest subset of permissions. Adding more permissions does not help on improving but may introduce redundant, noisy and irrelevant information. In the next Section, we thus evaluate to what degree the detection rates can be achieved with only the 40 risky permissions selected by each corresponding method.

Different methods may result in different subsets of relevant permissions. While CorrCoef and T-test select exactly the same subsets of risky permissions, MuInfo selects only one different risky permissions, as shown in Table 2. We examine the subset of permissions selected by SFS method and find that there are 25 permissions overlapped with the top-40 ranking results. PCA is not considered for overlap checking, because it does not directly select the 40 original permissions, but produces 40 PCs instead to represent the permissions. It's worth noting that permissions may be co-related: one of a pair co-related permissions can be represented by the other. This explains why the ACC and F-score curves almost remain after using more than 40 permissions, even the permission subsets may be different.

C. Comparative Detection Results with 7 Methods, with Kirin's Rules [23] as well as with RCP [5]

We employed the 7 methods to detect malapps based on the risky permission subsets (PCA is an exception, as it is based on linear combinations of permissions). The comparative results are shown in Figure 5 by ROC (Receiver Operating Characteristic) Curves that are the plot of TPR against FPR.

From Figure 5, we observe that all the 7 methods achieve high detection rates with low false positive rates based on only

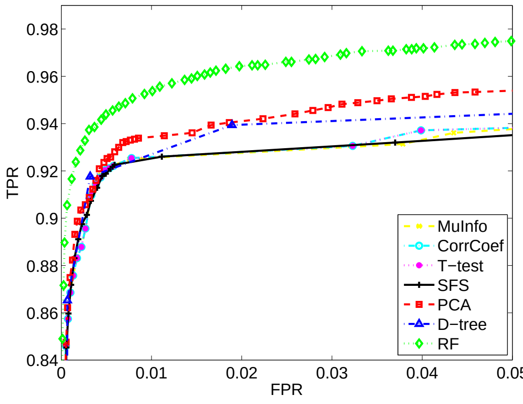


Fig. 5. The ROC of 7 methods when using only 40 relevant risky permissions.

40 relevant risky permissions. With Random Forest, the TPR rate reaches as high as 0.9462 with a FPR as low as 0.006. To make further comparison, we also use Kirin [23] as well as RCP [5] to detect malapps in our data. The TPR and FPR with 7 methods as well as results of Kirin and RCP are described in Table III. It is clear that all the 7 methods are superior to Kirin and to RCP in terms of both TPR and FPR.

TABLE III
COMPARISON OF TPR AND FPR WITH OUR 7 METHODS AND WITH KIRIN'S RULES [23] AS WELL AS WITH RCP METHOD [5]

Method	TPR	FPR
Muinfo	0.9228	0.0059
CorrCoef	0.9232	0.0060
T-test	0.9230	0.0060
SFS	0.9226	0.0059
PCA	0.9258	0.0056
D-tree	0.9281	0.0059
RF	0.9462	0.0060
Kirin [23]	0.2734	0.1390
RCP [5]	0.5596	0.0527

D. Extracted Rules and Their Performance

Malapps in different sets may have different characteristics, e.g., different distribution of permission requests. The detection of malapps with different characteristics is a big challenge. We are motivated to detect malapps with varied characteristics with a set of detection rules extracted from the permission sets with DTree. In the experiments, 95% of apps randomly selected from all the benign samples as well as three sets of malapps are used as data input to construct the rules, in order to detect the other single set of malapps. We used the remaining 5% of benign apps to evaluate the FPR. Note that only DTree can produce explicit rules. Randomly Forest ensembles a set of independently learned decision trees and it cannot construct explicit rules. To facilitate the comparison, we also include Random Forest in evaluation because it gives the best performance in our previous results.

107 decision rules are constructed with the 40 relevant permissions using all the 95% benign apps as well as sets of *Mal_Zhou* and *MAL_VS* with DTree. Each of such IF-THEN rules has *condition* as a conjunction of permission values and *consequence* as a class label, which is -1 (benign) and 1 (malicious) in our case. Note that in a rule the conjunction of permission values forms a sufficient condition for detecting malapps, but not a necessary condition. As an example, detection rule 281and 19 are depicted as

```
Rule 281:
READ_SMS = 1
INTERNET = 1
READ_EXTERNAL_STORAGE = 0
WRITE_EXTERNAL_STORAGE = 1)
class = 1 [99.7%]

Rule 19:
IF (READ_CALL_LOG = 1)
THEN class = -1 [100.0%]
```

While rule 19 indicates a benign app, rule 281 describes a kind of malicious behavior. The rule 281 can be interpreted as follows. An app requests to write but not to read external storage. In addition, it requests to connect the Internet and read SMS. This implies that the app may transfer the privacy information from the device to the external storage and then may find some means to send out the information by Internet stealingly. Apps that match (sufficient condition of) rule 281 are malicious with a probability of 99.7% in the training data. With in-depth analysis, we find that there are 415 malapps that matches rule 281. These malapps belong to the families of ADRD, AnserverBot, BaseBridge, etc. in the set of *Mal_Zhou*. In contrast, there are only 4 benign app samples matching the rule. This indicates that the rules well discriminate malapps from benign apps, and thus are effective for the detection of malapps.

The TPRs of our rules are visually compared with those of Random Forest and Kirin in Fig. 6, and the FPRs are given in Table IV. It is observed that our detection rules outperform Kirin's in terms of TPRs as well as FPRs. With the rules constructed from the set of *Mal_Zhou, Mal_COM2* and *Mal_VS* as well as 95% of all the benign apps, we detect 74.03% malapps in the *Mal_Com2* at FPR of only 0.12%.

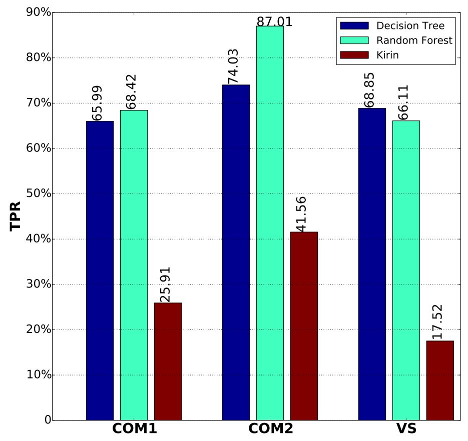


Fig. 6. The TPR for the detection of unknown malapps using the decision rules.

TABLE IV
THE FPRS USING THE DECISION RULES

Method	DTree	RF	Kirin
FPR	0.0012	0.0007	0.1414

E. Evaluation on a third-party app store

We also implement our methods on a set of apps collected from *AppChina* that is a widely used third-party app store in China. We collected 10,737 apps, in which 652 malapps are identified by at least two of ten anti-virus softwares (e.g., AntiVir, F-Secure, McAfee and Panda). The malapps thus account for approximately 5.56% of all the apps in the app store.

We used our 7 methods to detect malapps based on the permission requests, and the ROC curves are shown in Fig. 7. It is observed that our methods are effective as well to identify malapps in *AppChina*. As an example, Random Forest yields TPR of 0.9209 at a FPR of 0.0135.

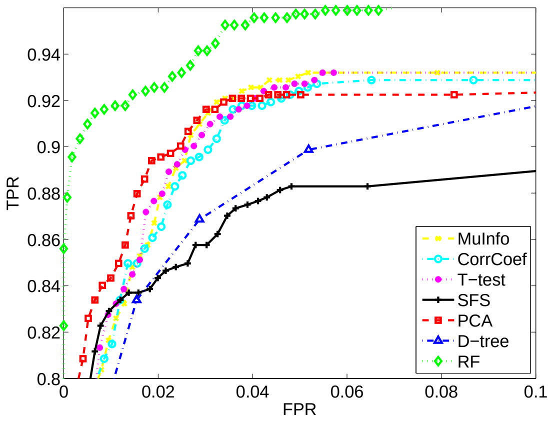


Fig. 7. The ROC of 7 methods on a set of Appchina.

VI. Discussion and Analysis

In this Section, we discuss two major questions for the analysis and detection of malapps with permission requests: (1) is it feasible for the detection of malapp detection with only permissions requested by apps? and (2) what are the main limitations behind the empirical results? what are the False Positives and False Negatives, and why?

Feasibility. Most malapps need to request sensitive permissions that allow them to access sensitive API, code or data in the device, while most benign apps do not. Intuitively, malapps can be distinguished from benign ones with permission requests. This is approved by comparing the permission usage between the malapps and benign apps shown in Fig. 2, where certain sets of permissions are very frequently requested by malapps but requested by benign apps with less probability. READ_SMS, for example, is requested by 53.62% of malapps but only requested by 1.36% of benign apps.

As shown in Fig. 8, malapps and benign apps request different number of permissions, showing different patterns of permission usage. In details, as shown in Fig. 9, different sets of malapps depict different patterns of permission usage, too. The different number of permissions requested by different kinds of apps gives another evidence that permissions can be used as effective features to discriminate malapps from benign apps.

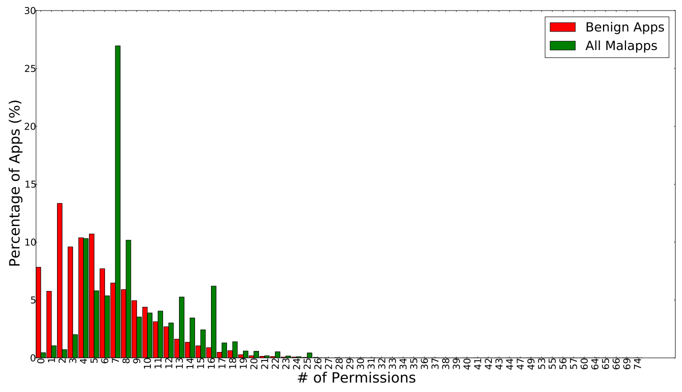


Fig. 8. The number of permissions requested by malapps and by benign apps (percentage).

We employ three methods, i.e., Mutual Information, CorrCoef and T-test to rank individual permission w.r.t. their risk to the system. We conduct extensive experiments on a very large app set and find that with only the 40 permissions, the detection accuracy almost reaches the best or maintains stable based on our 7 methods. We thus identify the 40 risky permissions that can be used for detecting malapps. With the 40 risky permission requests, our methods including SVM, DTree and Random Forest detect more than 92% malapps at a false positive rate of less than 0.59%. The decision rules constructed by DTree detect as high as 74.03% of malapps collected from different organizations.

Different apps may have the same permission requests. As shown in Table I, the number of distinct permission vectors only account for 9.25% of all the permission vectors. We thus examine the results to check whether a single malicious

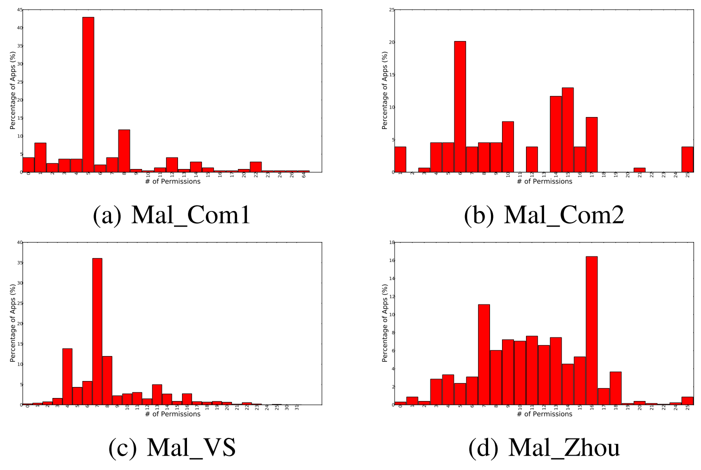


Fig. 9. The number of permissions requested by apps in Mal_Com1, Mal_Com2, Mal_VS and Mal_Zhou.

permission vector that we correctly detected represents a large number of malapps, leading to undesired biased detection results. Through the careful examination, we have two observations supporting our unbiased results. First, most families of malapps request different combinations of permissions, i.e., have different distinct permission vectors. For example, *Mal_Zhou* consists of 1,260 malapp samples that are categorized into 49 malapp families. There are in total 332 distinct permission vectors in *Mal_Zhou*, in which 324 distinct permission vectors are distributed in different malapp families with almost the same probability proportional to the number of malapps in their associated families. This also indicates that to a certain extent permission requests characterize the behaviors of Android apps. Second, the malicious permission vectors that have been correctly detected do not represent a large number of malapps. We describe the detailed information on what apps in the set of *Mal_Zhou* are correctly detected with Random Forest in Table V. It is observed that the malapps correctly detected by Random Forest are almost uniformly distributed in the malicious families of *Mal_Zhou*. The empirical results as well as the analysis further approve that permission requests can be effective to detect Android malapps.

Limitations. As Random Forest shows best performance for the detection of malapps, we carefully investigate in detail the FNs and FPs it produces. When Random Forest successfully detects 92.79% malapps (351 malapps undetected), simultaneously it produces 0.19% False Positives (605 benign apps falsely reported). By thorough investigation, we are aware that only considering permission requests as features may have difficulties to improve the current detection accuracy. Most apps only request a small number of permissions. As shown in Fig. 8, 92.8% benign apps and 97.3% malapps request no more than 12 and 18 permissions, respectively. The averaged number of permissions requested by all the apps used in our experiments is 5.7. As a result, the permission vectors are very sparse, which brings difficulties for the detection, not to mention that the permission vectors are Boolean type. We summarize the following kinds of difficulties.

First, a number of malapps request the exactly same permissions that are requested by benign apps. This gives negative impact on the detection accuracy. As shown in Table VI, 841 malapps request the same permission lists that are also requested by 67,836 benign apps. Among the undetected 351 malapps and 605 false positives, we find that 263 malapps share the same permission vectors with benign apps, while 392 false positives share the same permission vectors with malapps. In this case the detector cannot produce stable results that are basically determined by the ratio between the number of benign apps and of malapps. This may explain why the TPR increases slowly after it is over 0.97 for Random Forest or over 0.92 for SFS with the increase of TPR (see Fig. 5).

Second, some root-exploit type malapps do not need to request any permissions. In this case, relying on only permissions is not feasible for the detection of malapps (a large number of benign apps request no permission too). In addition, there is a number of malapps (14 samples in our data) that embed their malicious apks inside the up-level benign apk without requesting a permission. Normally it returns no permission if we directly extract the permissions from the benign up-level apk. This leads to false negatives. As mentioned, the Android app developers may request overprivileged permission requests that are never actually used in the app. This leads to false positives if only using the permission information for the detection.

Third, the 88-dimensional sparse vectors are not informative enough to comprehensively describe behaviors of an app. In our experiments, among the 605 false positives, 565 benign apps request risky permissions like SMS-related permissions or “READ_PHONE_STATE”, and are thus falsely reported as

TABLE V
DETAILED DETECTION RESULTS ON *Mal_Zhou* WITH RANDOM FOREST.

Family	# of samples	# of distinct samples	# of detected samples (TP)	# of distinct detected samples
ADRD	22	10	22	10
AnserverBot	187	11	187	11
Asroot	8	5	4	3
BaseBridge	122	33	102	20
BeanBot	8	5	7	4
Bgserv	9	2	9	2
CoinPirate	1	1	0	0
CruseWin	2	1	2	1
DogWars	1	1	0	0
DroidCoupon	1	1	1	1
DroidDeluxe	1	1	0	0
DroidDream	16	13	14	11
DroidDreamLight	46	25	43	22
DroidKungFu1	34	14	34	14
DroidKungFu2	30	8	30	8
DroidKungFu3	309	60	301	55
DroidKungFu4	96	23	94	21
DroidKungFuSapp	3	1	3	1
DroidKungFuUpdate	1	1	1	1
Endofday	1	1	1	1
FakeNetflix	1	1	1	1
FakePlayer	6	1	6	1
GGTracker	1	1	1	1
GPSSMSSpy	6	2	6	2
GamblerSMS	1	1	1	1
Geinimi	69	24	69	24
GingerMaster	4	1	4	1
GoldDream	47	21	47	21
Gone60	9	2	9	2
HippoSMS	4	2	4	2
Jifake	1	1	0	0
KMin	52	4	52	4
LoveTrap	1	1	1	1
NickyBot	1	1	1	1
NickySpy	2	2	2	2
Pjapps	58	24	58	24
Plankton	11	8	7	5
RogueLemon	2	2	2	2
RogueSPPush	9	2	9	2
SMSReplicator	1	1	0	0
SndApps	10	3	8	2
Spitmo	1	1	1	1
Tapsnake	2	1	1	1
Walkinwat	1	1	0	0
YZHC	22	4	22	4
Zitmo	1	1	1	1
Zsone	12	3	11	2
jSMSHider	16	4	6	2
zHash	11	4	11	4
SUM	1260	341	1196	300

TABLE VI
THE NUMBER OF SHARED PERMISSION REQUESTS BETWEEN THE BENIGN APPS AND MALAPPS.

App Set	# of overall samples	# of overlapped samples
MAL_ZHOU	1260	109 (8.65%)
MAL_COM1	247	73 (29.55%)
MAL_COM2	154	20 (12.99%)
MAL_VS	3207	639 (19.93%)
All Malapps	4868	841 (17.28 %)
Benign Apps	310926	67836 (21.82 %)

malapps. Indeed, the functionality of these apps is truly relevant to the requested risky permissions. However, additional information is required to justify their permission requests.

Going beyond our work on malapp classification, when conducting **anomaly detection**, a set of benign apps with informative features is required to build normal detection models that are effective to discover novel malapps. A sample is reported as an anomaly if its behavior significantly deviates the normal models. However, it would be a barrier for anomaly detection because of the sparseness of the permission vectors. As malicious apps have become sophisticated and polymorphic, more features need to be explored in order to improve the capacity of detectors for the detection of novel malapps. Our in-depth analysis and extensive results indicate that the use of permissions can be used to detect malapps for the first step, since it can be effective and can process a very large set of samples efficiently.

VII. Related Work

The analysis and detection of Android malicious applications is an emerging issue. The related work of this paper mainly falls into two aspects: permission analysis and permission-based malapp detection.

Permission analysis. Felt et al. [22] performed two usability studies to examine whether the Android permission system is effective at warning users. In detail, they evaluated whether Android users pay attention to, understand, and act on permission information during installation. Their work indicates that current Android permission warnings do not help most users make correct security decisions. Pandita et al. [6] proposed WHYPER, in an attempt to help the users to understand what permissions an app requests before its installation, by identifying the sentences that describe the need for a permission with Natural Language Processing (NLP) techniques. Barrera et al. [35] examined 1,100 Android apps' permission requests and used Self-Organizing Maps (SOM) to perform 2-dimensional visualization that depicts which permissions are used in apps with similar patterns.

Felt et al. [19] developed Stowaway to detect overprivileged permissions in Android apps. Au et al. [36] extended Stowaway and developed PScout, a tool that extracts the permission specification from Android with static analysis. With PScout they analyzed 4 versions of Android spanning from version 2.2 up to the Android 4.0. Their work indicates that while there is little redundancy in the permission specification, 22% permission are unnecessary if apps can be constrained to only use documented API. AppProfiler presented by Rosen et al. [37] produced apps' behavioral profiles by creating a knowledge base of API calls with privacy-related behavior. The profiles were then used for detecting privacy-related app behaviors. Zhang et al. [38] proposed VetDroid that is a dynamic analysis framework, in an attempt to capture app-system interactions and sensitive behaviors inside an app, by using the permission usage behavior. While our methods also provide a better understanding of permission use, our work is mainly based on the comprehensive analysis of permission requests for the detection of malapps on a very large scale.

Permission-based malapp detection. There exists related work that mainly relies on Android permissions to detect or analyze malicious apps. Chen et al. proposed [39] Pegasus, in an attempt to detect malapps that are characterized by the temporal order in which an app uses APIs and permissions. They constructed Permission Event Graph (PEG) with static analysis and implemented models of the Android event-handling mechanism and APIs. Over 200 apps were used in their experiments to evaluate Pegasus. Peiravian and Zhu [40] used permissions and API calls of Android applications to detect malapps. Chakradeo et al. [13] used Multiple Correspondence Analysis (MCA) method to detect malapps with attributes like permission requests, intent filters, native code and presence information of zipped files that were extracted from manifest files of apks. Their methods detect 90% of malapps while suffering a false positive rate of 6.5%. Arp et al. [12] used permission requests as well as other static features (e.g., API calls, intent filters) to detect malapps. When using SVM as classifier, they produce a TPR of 94% at a FPR of 1%. Our work mainly focuses on Android permission. We used 7 methods for the classification and the results given by Random Forest shows slight superior results (i.e., TPR of 94.62% at a FPR of 0.6%). Zhou et al. [41] proposed Droidranger in order to detection malapps on popular Android markets. They used several schemes to aid the detection. Permission request information is mainly used as a criteria to filter out unrelated apps. Our work focuses on the analysis of malapps with permission usage. We systematically evaluate and discuss the malapp detection results with permission ranking and different effective claudication algorithms on a very large data.

The closest research to our work was reported by Sarma et al. [5], Peng et al. [4] and Enck et al. [23]. Sarma et al. [5] used permission information as data source to evaluate the risk of an app by examining how rare permissions it requests comparing with the permissions requested by other apps in the same categories. 158,062 benign apps as well as 121 malapps were used for the evaluation. Our work provides systematic analysis on permissions. We classify the malapps based on several algorithms with permission usage without considering their categories. In addition, while we use a much larger app set for the evaluation, our method shows superior results in Table III. Peng et al. [4] used three probabilistic generative models, i.e., Naive Bayesian models, mixture of Naive Bayes models and novel hierarchical Bayesian models for risks scoring and risk ranking for apps. Our work is different with their research in that we rank the risk of permissions (while they ranked the apps with a score) and identify a subset of risky permissions for the detection of malapps. In addition, we systematically discuss the feasibility as well as the limitations by using the permissions for the analysis of malapps.

Enck et al. [23] constructed 9 permission rules called Kirin that classifies an app as potentially malicious if the app requests certain combinations of permissions that match the rules. The rules are defined by security requirement engineering. In contrast, we generated over 200 detection rules based on empirical analysis of a very large app sets with decision tree. Our rule set outperforms Kirin in the detection of malapps.

Our work provides in-depth analysis and discussion on the effectiveness as well as the limitations of malapp detection with only permission requests based on empirically extensive experiments on a very large scale. Permission usage is intuitive and important feature for Android apps. Our analysis thus provides a vision regarding the use of permission requests for the detection of malapps.

VIII. Conclusion

In this work, we provide a systematic study on the exploration of permission-induced risk in Android apps on a large-scale in three levels. First, we rank all the individual permissions w.r.t. their potential risk with three methods. Second, we identify subsets of risk permissions with sequential forward selection as well as with PCA. We then employ several algorithms, namely, SVM, Decision Tree and Random Forest, to detect malapps based on the identified subsets of risky permissions. We also construct rule sets with Decision Tree to detect malapps with different characteristics. A large official app data set consisting of 310,926 benign apps and 4,868 malapps, as well as a third-party app set are used for the evaluation. The empirical results show that with the 40 risky permissions, the best detection rate with Random Forest reaches 0.9462 at a false positive rate of 0.006. We share our data on the public for the research community. Our study indicates that risky permissions can be effective for the detection of malapps at least for the first scan of a large amount of Android apps. Our study also provides insightful understanding regarding the risk of individual permissions and of their combinations for Android users as well as for the developers.

We discuss and analyze in depth the effectiveness as well as the limitations on the detection of malapps with only permission requests. While the permission requests characterize the behaviors of apps to certain extent and the detection can be effective, only considering the permissions would have difficulties to improve the current detection accuracy, as the permission vectors are very sparse and Boolean type. In the future work, we are exploring more relevant features that inherit in apps in order to improve the detection accuracy of Android malapps.

References

[1] D. Research, "Global smartphone shipments to reach 1.24 billion in 2014," http://www.digitimes.com/news/a20131125PD218.html, published in 2013.

[2] I. D. Corporation, "Android pushes past 80% market share while windows phone shipments leap 156.0% year over year in the third quarter, according to idc," http://www.idc.com/getdoc.jsp?containerId =prUS24442013, published in 2013.

[3] B. Uscilowski, "Symantac white paper: Mobile adware and malware analysis," http://www.symantec.com/content/en/us/enterprise /media/security_response/whitepapers/madware_and _malware_analysis.pdf, published in 2013.

[4] H. Peng, C. S. Gates, B. P. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *CCS*, 2012.

[5] B. P. Sarma, N. Li, C. S. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: a perspective combining risks and benefits," in *SACMAT*, 2012.

[6] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "Whyper: Towards automating risk assessment of mobile applications," in *USENIX Security Symposium*, 2013.

[7] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *USENIX Security Symposium*, 2011.

[8] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *MobiSys*, 2012.

[9] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *CCS*, 2012.

[10] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy*, 2012.

[11] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *SecureComm*, 2013, pp. 86–103.

[12] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieckand, "Drebin: Effective and explainable detection of android malware in your pocket," in *NDSS*, 2014.

[13] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "Mast: triage for market-scale mobile malware analysis," in *WISEC*, 2013, pp. 13–24.

[14] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI*, 2010.

[15] P. Hornyack, S. Han, J. Jung, S. E. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *CCS*, 2011.

[16] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in *CODASPY*, 2013.

[17] L.-K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis," in *USENIX Security Symposium*, 2012.

[18] Android, "Andriod developers guides," http://developer.android.com/guide/topics/security/ permissions.html, retrieved in June 2014.

[19] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *CCS*, 2011.

[20] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *NDSS*, 2012.

[21] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *USENIX Security Symposium*, 2011.

[22] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: user attention, comprehension, and behavior," in *SOUPS*, 2012.

[23] W. Enck, M. Ongtang, and P. D. McDaniel, "On lightweight mobile phone application certification," in *CCS*, 2009.

[24] Android, "Android security overview," http://source.android.com/devices/tech/security /index.html, retrieved in June 2014.

[25] ——, "Android manifest," http://developer.android.com/guide/topics/ manifest/permission-element.html, retrieved in June 2014.

[26] W. of VirusTotal, "Virustotal," https://www.virustotal.com, retrieved in June 2014.

[27] W. of virusshare, "Virusshare," http://virusshare.com/, retrieved in June 2014.

[28] A. Developers, "Manifest permission," http://developer.android.com/reference/android/Manifest.permission.html, retrieved in June 2014.

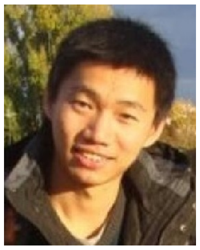
[29] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of Machine Learning Research (JMLR)*, vol. 3, pp. 1157–1182, 2003.

[30] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artificial Intelligence*, vol. 97, no. 1, pp. 273–324, 1997.

[31] C. J. C. Burges, “A tutorial on support vector machines for pattern recognition,” *Data Mining and Knowledge Discovery*, vol. 2, no. 2, pp. 121–167, 1998.
[32] J. R. Quinlan, “Induction of decision trees,” *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986.
[33] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.
[34] Trend, “Trendlabs 2q 2013 security roundup,” http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-2q-2013-trendlabs-security-roundup.pdf, published in 2013.
[35] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, “A methodology for empirical analysis of permission-based security models and its application to android,” in *CCS*, 2010.
[36] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, “Pscout: analyzing the android permission specification,” in *CCS*, 2012, pp. 217–228.
[37] S. Rosen, Z. Qian, and Z. M. Mao, “Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users,” in *CODASPY*, 2013.
[38] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, “Vetting undesirable behaviors in android apps with permission use analysis,” in *CCS*, 2013.
[39] K. Z. Chen, N. M. Johnson, V. D’Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. X. Song, “Contextual policy enforcement in android applications with permission event graphs,” in *NDSS*, 2013.
[40] N. Peiravian and X. Zhu, “Machine learning for android malware detection using permission and api calls,” in *ICTAI*, 2013, pp. 300–305.
[41] 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 *NDSS*, 2012.

Wei Wang is currently associate professor in the School of Computer and Information Technology, Beijing Jiaotong University, China. He earned his Ph.D. degree in control science and engineering from Xi’an Jiaotong University, in 2006. He was a postdoctoral researcher in University of Trento, Italy, during 2005-2006. He was a postdoctoral researcher in TELECOM Bretagne and in INRIA, France, during 2007-2008. He was a European ERCIM Fellow in Norwegian University of Science and Technology (NTNU), Norway, and in Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, during 2009-2011. He visited INRIA, ETH, NTNU, CNR, and New York University Polytechnic. He is young AE of Frontiers of Computer Science Journal. He has authored or co-authored over 35 peer-reviewed papers in various journals and international conferences. His main research interests include mobile, computer and network security.

Xing Wang is currently a Ph.D. student in the School of Computer and Information Technology, Beijing Jiaotong University, China. He received his B.S. degree from Beijing Jiaotong University in 2009. He visited King Abudullah University of Science and Technology (KAUST) during January-April 2014. His main research interests lie in mobile security.



Dawei Feng is currently a Ph.D. student in Laboratoire de Recherche en Informatique, CNRS, Université Paris-Sud, France. He received his B.S. degree and M.S. degree from National University of Defense Technology in 2008 and in 2011, respectively. His research interests lie in data mining and machine learning. He visited King Abudullah University of Science and Technology (KAUST) in August 2013. His main research interests include machine learning, data mining as well as cloud computing.

Jiqiang Liu received his B.S. (1994) and Ph.D. (1999) degree from Beijing Normal University. He is currently a Professor at the School of Computer and Information Technology, Beijing Jiaotong University. He has published 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 his Ph.D. degree from China Academy of Engineering Physics, in 1991. He is currently a Professor at the School of Computer and Information Technology of Beijing Jiaotong University. He has authored or co-authored over 100 papers in various journals and international conference. 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 (MINE) Laboratory (http://mine.kaust.edu.sa) in the Division of Computer, Electrical and Mathematical Sciences & Engineering, King Abdullah University of Science and Technology (KAUST). She was an European ERCIM research fellow in the Department of Computer and Information Science, NTNU, Norway, during April-August 2010. She earned her Ph.D. degree in computer science with great honors from INRIA-University Paris-Sud 11, France, in July 2010. She visited IBM T. J. Watson Research Center, Texas A&M University, University Paris-Sud 11, Concordia University, Microsoft Research Asia, and the University of Luxembourg. She serves as a PC member for premier conferences like KDD 2014, ICDE 2014 and ICDM 2014. She has authored or co-authored over 50 refereed papers in various journals and conferences. 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.