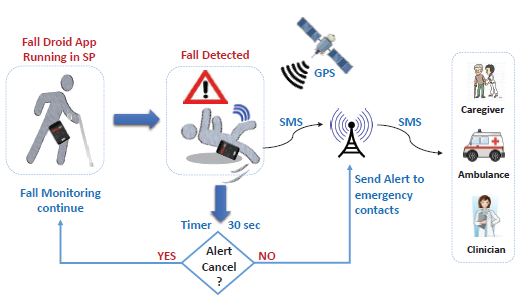
**CHAPTER 4**

**METHODOLOGIES**

**4.1 Falldroid**

FallDroid is designed as a standalone and user independent fall detection system that actively runs in the background and uses a two-step algorithm to analyze subject movement. Upon fall detection, the application triggers the SP to vibrate and an alert cancellation page appears on the screen. Unless canceled within a specified time period (default setting of 30 s) by the user, a sound alarm is activated followed by a help text message containing location information being sent to specified emergency contacts. This process is illustrated in Fig. 1.



**Fig. 4.1: Overview of fall detection and emergency alert system.**

**4.1.1. FallDroid Implementation**

FallDroid has been developed using Android Studio IDE with min API 17 and target API level 23. The GUI of application consists of four screens: the main page, settings, fall alert cancellation, and feedback. The layout is designed to facilitate usability by the elderly with an overall focus on reducing battery power consumed particularly for unnecessary computations. In what follows, we highlight the main services provided by the application.

**1) Configuration and Control**: The main page of the application serves three functions:

(i) start/stop the fall detection process,

(ii) application configuration settings, and

(iii) ummarized display of critical/crucial settings.

Once the user starts the sensing process, a notification icon will appear on the top left corner of the screen in the notification bar. The icon remains visible as long as the application is running in the background.The settings page can be used to customize personal details, location of the SP being carried, fall detection service priority, and settings related to fall alert notification. Settings for fall alert notification comprise of alarm sound and duration, countdown timer for alert cancellation, and entries for emergency contacts. Changes made to any user-specified setting can be seen immediately under the settings summary section on the main page. The settings are saved using the Shared Preferences class which permits previous

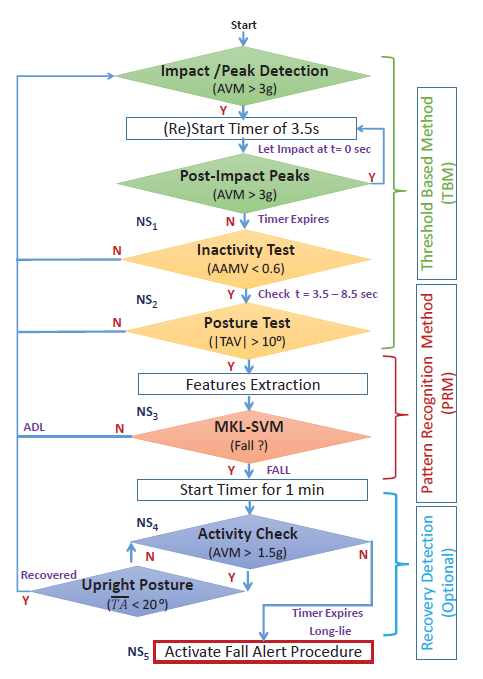
settings to persist over multiple sessions as well as after SP reboot.

**2) Fall Detection Service**: The algorithm proposed for fall detection is implemented as an Android service, using the Intent Service class, that can run continuously in the background irrespective of the application. The intent service runs the algorithm in its own separate worker thread without blocking the main UI thread which otherwise can make the application non-responsive. Once the service is activated, it instantly acquires the PARTIAL WAKE LOCK, which prevents the CPU from going into sleep mode when the phone is idle. Subsequently, the algorithm checks the device accelerometer and its sensing capabilities. Once found, it registers the Sensor Event Listener to receive motion data from the accelerometer at a desired sampling rate. The incoming sensor data values are stored in a linked-list queue following the First-In-First-Out (FIFO) discipline, and maintains data history of up to 6 s. Initial steps of the TBM algorithm use this data to detect fall like events. Upon detection, the rest of the algorithm is then executed. The parameters associated with pre-trained MKLSVM are not hard-coded but instead provided via a file. In this manner, application personalization or update is facilitated by replacing the parameters file with a new one.

**3) Notification System:** When a fall event is detected by MKL-SVM, the SP starts vibrating followed by an automatic launch of the fall alert cancellation page. The user has a default period of 30 s to cancel the false alert in case an ADL event is mis-classified as fall. If the user cancels the alert, the application then requests for feedback through the feedback page. The entered data is stored in a file which can later be used for further analysis to improve the algorithm performance. However, in case of a real fall event, the soundalarm is triggered once the alert countdown timer expires. The system then retrieves the last known geographical position of the user based on the available location providers (like GPS, Network) and sends the most accurate result in the form of an alert text message to the pre-specified emergency contact number. If the last known location was not too accurate (> 100

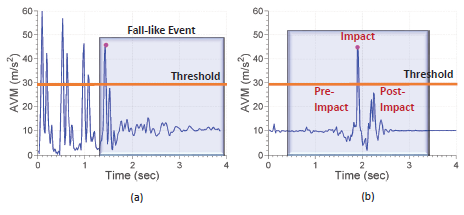
meters) or not recent (> 10 min), the application then tries to acquire the latest/current location prior to sending the alert message.

**4.1.2. Fall Detection Algorithm**



**Fig 4. 2: Fall detection algorithm**

As shown in Fig. 2, FallDroid uses a two-step fall detection approach composed of TBM and MKL-SVM. The first step of the algorithm relies on TBM for determining fall-like events and efficiently discarding most fall-like ADLs using inactivity and posture tests. As falling down is a single incident that occurs instantaneously, a fall-like event does not exhibit the traits of repetitiveness and is basically characterized as an acceleration peak of magnitude greater than the 3 g threshold, followed by a period of 3:5 seconds without further peaks exceeding the threshold (Fig. 3).



**Fig 4. 3: (a) Fall-like ADL event generated by running and sudden stop activity.**

**(b)Forward simulated fall pattern with a 3 sec window centered at impact point.**

The threshold value 3g is selected as it is widely reported in the literature and is small enough to avoid false negatives, as even low impact falls have peak acceleration greater than 3g value. When the user is static (i.e. no SP or sensor movement), the acceleration vector magnitude (AVM) shows a value of 1 g representing

gravitational acceleration. However, during free fall, this value falls below 1 g and upon having impact with the ground, a high value (> 3 g) appears. The AVM signal used to detect

fall-like events is:

(1)

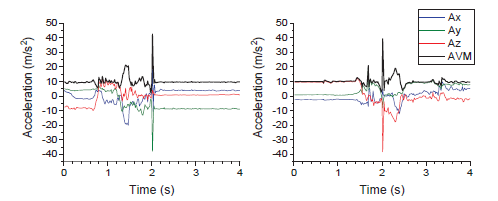


where Ax, Ay and Az represent respectively, the acceleration signals along x, y, and z axes of the sensor. When a falllike event is detected, the user is sensed for any activity over a default period of 5 s. The average absolute acceleration magnitude variation (AAMV) measure defined below is then compared with pre-defined threshold to analyse the inactivity event:

(2)



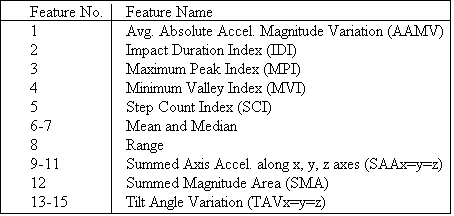
In case of little or no movement after impact, (2) yields values close to zero (AAMV < 0:6). If the user was inactive, then his/her posture variation is evaluted using pre- and postimpact (peak) signals. If the user shows postural change, as tested by tilt angle variation (TAV > 10\_), then it is most likely a fall. Hence, decisions on such fall-like events that are not discarded by TBM are made in the second step of the algorithm which exploits the powerful MKL-SVM pattern recognition technique, explained later on in this sub-section. Some examples of acceleration patterns generated by thigh located SP that cause false alarms are shown in Figure 4.

****

**Fig. 4.4: Examples of thigh acceleration patterns during ADL that triggered false alarms**

An optional long-lie or recovery detection step can also be applied at the end of the algorithm to further alleviate the false alarms. If the user shows significant movement after the fall event and regains an upright posture within one minute, then it will be treated as recovery from fall and no fall alert will be generated. The extended test serves more efficient in discarding fall-like ADL events or to detect recovery from (non-injurious) fall as the user is more likely to show movement. On the other hand however, such prolonged tests would delay decision-making and medical assistance in case of an actual fall. The thresholds are obtained from heuristic analysis using the training data set samples. Furthermore, the thresholds are chosen such that no actual fall event is mistakenly classified and discarded as ADL in the TBM stage. This ensures that the algorithm can achieve 100% sensitivity depending on the performance of MKL-SVM. In other words, by choosing higher threshold values for AAMV and lower values for TAV, more number of fall-like ADL events are likely to be passed to the computationally expensive MKL-SVM. However, owing to the fact that the number of fall-like events occurring daily are usually very few, the impact on power consumption will be insignificant.

The receiver operating characteristics (ROC) curves of AAMV and TAV measures. It can be seen that each of them can achieve 100% sensitivity while discarding almost 60% of the ADL events. Feature Extraction: A set of 15 features shown in Fig 5. were extracted from each simulated fall and fall-like ADL



**Fig 4. 5: Summary of the 15 candidate features derived from the accelerometer signals.**

**4.2 Permission Induced Risk In Android**

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 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.

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 consider 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.

*•* Analysis is based on 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 .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

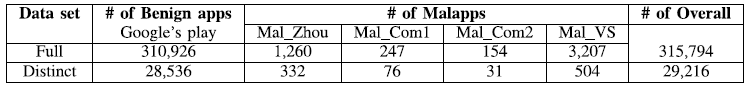
most prominent includes app sandbox and a permission framework that enforces access control to core functionalities. App sandbox is set up in a kernel lever. It enforces security between apps and the system through identifying and isolating app resources. Each Android app is assigned a unique UserID (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. 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 analyse 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.

**4.2.1 Data Sets**

In order to conduct extensive analysis on permission usage, we need to establish a large well-labeled app set. For benign apps, we consider a total number of 310,926 free apps from Google’s play.



**Fig 4.6 :Datasets of Malapps and Benign apps**

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. 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*  that is a repository of malware samples. All the malapps in the *Mal\_VS* were approved by *VirusTotal* . 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

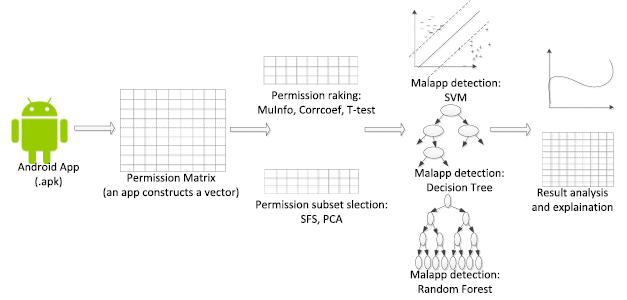
3,207 malapps in *Mal\_VS*. In this work, we only consider the permissions provided by Android system, although an app can also define its own permissions. To analyze the permission usage of apps, we mainly extract the Android permission list from the Manifest

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. 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. Therefore, each app can be represented by a 88-dimensional Boolean vector, where 1 denotes that the app requests the permission and 0 otherwise.

**4.2.2 Methods**

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.



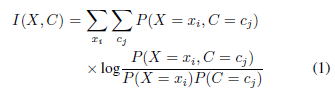
**Fig 4.7. The methods for exploring permission-reduced risks and the detection of malicious app**

**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 . 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) 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



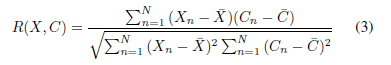
where *P*(*C* = *cj*) is the frequency count of class *C* with value *cj* , *P*(*X* = *xi*) is the frequency count of permission *X* with value *xi*, and *P*(*X* = *xi, C* = *cj*) is the frequency count of *X* with value *xi* in class *cj* . In this paper, the class *C* has binary values, *c*0 for *benign* 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*.

**2) Pearson CorrCoef:**Pearson CorrCoef measures the relevance of *X* and *C* by

(2)



which in our case of binary class and boolean variable becomes

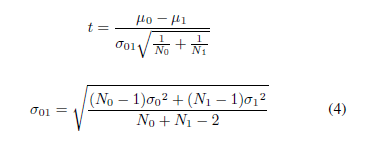


where *X* (resp. *C*) is the average of all sample values of *X* (resp. *C*), *Xn* (resp. *Cn*), *n* = 1*...N* . *R*(*X,C*) has a value in [*−*1*,* 1], where *R*(*X,C*) = 0 indicates the independency of *X* and *C*, *R*(*X,C*) = 1 indicates the strongest positive correlation of them and *R*(*X,C*) = *−*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.

***3*) 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., *μ*0 = *μ*1. T-test is performed by



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 *Xi*, 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*(*Xi, C*), *|R*(*Xi, C*)*|* or *|ti|* 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 (\_*M k*=1 \_*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.

**1) 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.

**2) 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 **X** whose columns are permissions *Xi*, *i* = 1*. . .M*, PCs are actually eigenvectors of the gamma matrix of data **X**, defined as



where **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 **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.

**1) 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 .

**2) Decision Tree (DTree***):* DTree [32] learns a classification model with a tree structure, where nodes arewhere **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 **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.

**3) Random Forest:**Random Forest 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.