PIndroid: A novel Android malware detection system using ensemble learning methods

Fauzia Idrees[a,*], Muttukrishnan Rajarajan[a], Mauro Conti[b], Thomas M. Chen[a], Yogachandran Rahulamathavan[c]

[a]*School of Mathematics & Engineering, City University London, EC1V 0HB, UK*
[b]*Department of Mathematics, University of Padua, 35122 Padova, Italy*
[c]*Institute for Digital Technologies, Loughborough University London, UK*

Abstract

The extensive use of smartphones has been a major driving force behind a drastic increase of malware attacks. Covert techniques used by the malware make them hard to detect with signature based methods. In this paper, we present PIndroid—a novel **P**ermissions and **In**tents based framework for identifying An**droid** malware apps. To the best of our knowledge, PIndroid is the first solution that uses a combination of permissions and intents supplemented with Ensemble methods for accurate malware detection. The proposed approach, when applied to 1,745 real world applications, provides 99.8% accuracy (which is best reported to date). Empirical results suggest that the proposed framework is effective in detection of malware apps.

Keywords: Malware classification, Permissions, Intents, Ensemble methods, Colluding applications

1. Introduction

In past few years, smartphones have transformed from simple mobile phones into mobile computers, making them suitable for personal and business activities. Smartphones have become the major target for mobile malware due to increased reliance on them for daily activities such as storing private data, financial transactions, emailing, socializing and online shopping.

*Corresponding Author
Email address: Fauzia.Idrees.1@city.ac.uk (Fauzia Idrees)

Android being the most widely used platform for smartphones is under constant attacks. Existing anti-virus solutions are not capable of eliminating the exponentially increasing malware threats due to their reliance on signature-based detection. Moreover, resource constrained smartphones are unsuited for continuous malware scanning. There is a need to have an efficient method capable of overcoming the current challenges of outdated signatures, code obfuscation and resource constraints.

Permissions are used to guard against misuse of system resources and user data, however, some of Android's features like *intents* can break this shield. A lot of research has been done on permissions; however, intent is an under investigated area (in malware detection), providing opportunity for the evolving malware threats.

We propose a malware detection approach which classifies apps against certain combinations of permissions and intents which are unique to malware apps. These combinations form an efficient detection pattern to differentiate between malware and benign apps with a granularity to classify malware families. We evaluate the efficacy of proposed approach by applying machine learning algorithms. A comparative study of classifiers is carried out against different performance measures to select the most accurate and efficient classifier. We apply the ensemble methods to optimize the results.

Contributions. The main contributions presented in this paper are:

1. To the best of our knowledge, this is the first work that combines intents and permissions for collaborative malware detection. This work combines *permissions* and *intents* of applications to generate a distinguishing matrix that is used for efficient and accurate detection of malware and its associated families. Our method is capable of achieving 99% detection accuracy by combining permissions and intents.
2. We propose a new approach using ensemble methods to optimize the classification results. Our results show a detection accuracy of 99.8% by connecting multiple classifiers laterally with a meta-classifier.
3. We apply statistical significance test to investigate the correlation between permissions and intents. We found statistical evidence of a strong correlation between permissions and intents which could be exploited to detect malware applications.

Organization. Section 2 discusses the related work; Section 3 provides an overview of Android permissions and intents and Section 4 discusses about

the analysis carried out on permissions and intents. Section 5 presents the proposed framework; Section 6 describes the model evaluation, experimental settings, and results. Section 7 concludes the paper.

2. Related Work

There is a plethora of research work on Android security covering vulnerability assessments, malware analysis and detection. Faruki et al (2015) present an overview on the current malware trends. Malware analysis leverage static, dynamic and hybrid methods. In static malware analysis, properties of apps are extracted by analysing different static features without running the code. In dynamic analysis, the runtime profiles of apps are generated by monitoring and collecting the memory consumption, CPU usage, battery usage and network traffic statistics (Shabtai et al., 2012) and (Crowdroid, 2011). Here, we provide an overview of related works in this area.

2.1. Static malware analysis on Android platform

Different static features such as permissions, API calls, Inter-process communication (IPC), code semantics, intents, hardware, components and developer ID have been used for malware detection. However, permissions, API calls, and IPC have attracted more attention from the researchers. There are a few works in which different features have been combined for malware detection. Here, we discuss the relevant works which use permissions, ICC/intents or hybrid features.

2.1.1. Permission analysis

Permission is the most investigated feature in malware detection. Barrera et al. (2010) examined 1,100 apps for permission usage and found the high frequency of certain permissions. Peng et al. (2012) calculated the risk scores of apps by analysing the requested permissions. Kirin (2009) identified dangerous combinations of permissions and developed the security rules to identify malicious apps. Vidas and Christin (2014) identified the unnecessary permission requests by the apps. VetDroid (2013) examined mapping between API calls and permissions for behaviour profiling. Sarma et al. (2012) calculated risks and benefits of requested permissions to discern the adverse affects of app. Stowaway (2011) is a tool to check the over-privilege of apps by mapping requested permissions with APIs. PScout (2012) is another tool which extracts permissions from the source code and maps them with URIs.

Most of these methods aim to provide help to app developers and security analysts. These methods may be used as add-ons with malware detection solutions.

2.1.2. Inter-Component Communication / Intents analysis

ICC and intents have not been explored the way permissions have been investigated. Most of the existing ICC based studies focus on finding the ICC related vulnerabilities. Enck et al. (2009) investigated the IPC framework and interaction of system components. ComDroid (2011) detects the ICC related vulnerabilities. Kantola et al. (2012) suggested improvement in ComDroid by segregating the communication messages into inter and intra-applications groups so that the risk of inter-application attacks may be reduced. Maji et al. (2012) characterized Android components and their interaction. They investigated risks associated with misconfigured intents. CHEX (2012) examined vulnerable public component interfaces of apps. Avancini et al. (2013) generated test scenarios to demonstrate the ICC vulnerabilities. DroidSafe (2015) performs information flow analysis to investigate the communication exploits. Gallingani et al. (2015) investigated intents related vulnerabilities and demonstrated how they may be exploited to insert the malicious data. Their experiments found 29 out of a total of 64 investigated apps as vulnerable to intent related attacks. All of these works focus on finding communication vulnerabilities, and none of them used ICC and intents for malware detection.

2.1.3. Malware analysis with hybrid features

In this category, different features are combined for effective malware detection. Most relevant works are: Drebin (2014), DroidMat (2012) and Marvin (2015) as they use permissions and intents in addition to other features for malware classification. Drebin (2014) examines the manifest file and code of apps to check the permissions, API calls, hardware resources, app components, filtered intents and network addresses. It uses Support Vector Machines (SVM) for malware classification. DroidMat (2012) analyses features extracted from the manifest and smali files of disassembled codes. These features include permissions, components, intent messages and API calls. It applies K-means algorithm and Singular Value Decomposition (SVD) method for clustering and low-rank approximations respectively. They analysed a total of 1738 apps comprising of 1500 benign and 238 malware samples. Marvin (2015) uses both the off-device static and dynamic analyses meth-

ods for malware detection. It uses around 490,000 features extracted from the manifest files and disassembled codes. Its high-dimensional feature set includes permissions, intents, API calls, network statistics, components, file operations, phone events, app developer IDs, package serial numbers and bundles of other features. It uses linear classifier to detect malware apps and assigns malicious score on a scale of 0 to 10, with 0 being benign and 10 being malicious.

3. Background on Permissions and Intents

Android uses permissions and intents to protect user data and device resources. Android has 117 permissions and 227 intents in version 4.4, API level 19 - an API level is an integer value which identifies the application's compatibility with the Android versions. The earliest Android version: API level 1, contains only 76 permissions and 124 intents. Google adds new permissions and intents into every upcoming versions. This trend is depicted in Table 1, where monotonic increment in permission and intents against the API levels is obvious. The increased number of permissions and intents has not only added new features but also opened the doors for malware. In this section, we present a high-level overview of Android permissions and intents.

3.1. Permissions

Permissions play a pivotal role in Android security. It controls access to the sensitive system resources, user data, and device functionalities. Permissions invoke API calls related to different functionalities. A complete set of permissions is declared in app's manifest file and at the onset of installation. User is prompted to approve the complete set of requested permissions as a pre-requisite of app installation. There is no option to choose among the requested set of permissions. Once the access is granted, the permissions remain valid till the time the app is either un-installed or updated. User can only check the permissions of apps but cannot delete or change them. A feature to change the permissions was added in Android 4.2 but was later removed through an update to avoid app crash if user disables the required permission(s).

Android permissions are categorized into four protection levels: *Normal, Dangerous, Signature* and *Signature or System*. Android has an access mechanism to check the permissions of apps and determine if they are authorized to access the protected resources. *Normal permissions* are automatically

granted to apps without user's intervention as these are not considered harmful. *Dangerous permissions* need user's approval due to associated risk of privacy leaks and access to sensitive API calls. *Signature permissions* are granted only to the apps, signed with the same certificate which defines the permission. *Signature or system permissions* are granted to either the pre-installed apps or those signed with the device manufacturer certificate. These permissions are unobtainable by third party apps.

3.2. Intents

Intent is the basic communication mechanism used to exchange the inter- and intra-app messages. An intent conveys the intention of the app to perform some action. It specifies the label for a component, its category and action to be performed.

Intents are of two types: *Explicit* and *Implicit*. *Explicit intent* specifies the component exclusively by the class name. These are generally used by apps to start their own components. *Implicit intent* does not specify the component by name. It states the required action only; system selects the app that has the component to handle the stated action. With explicit intent, the system launches the specified component immediately while with implicit intents, system looks for the appropriate component by comparing the intent filters. If there is any match between the intent and intent filter, the component of that app is launched. In case of multiple matching intent filters, users are sent with a dialogue box to select the app Yang et al. (2014).

Intents facilitate apps with same user ID to use each other's functionalities without separately declaring the permissions for them. This helps apps to gain extra privileges by augmenting the permissions.

4. Analysis of Permissions and Intents

A total of 1300 malware and 445 benign apps are analysed which are collected from well known sources such as Google Playstore[1], Contagiodump[2],

[1]Google Play, Web: https://play.google.com/store?hl=en

[2]Contagio Mobile: mobile malware mini dump, Web: http://contagiominidump.blogspot.co.uk/

Genome[3], Virus Total[4], theZoo[5], MalShare[6], and VirusShare[7]. Table 2 depicts the details of malware samples collected from each source. These sources contain the datasets of already known malware samples. Maliciousness of these samples is also confirmed with VirusTotal service integrated with ten detection engines. We labelled the app as malware, if it was detected as malicious at least by two of the engines. Cryptographic hashes (SHA-1) of files were also checked with a tool: HashTab[8] to ascertain the uniqueness of samples. Details of known malware families, their malicious activities and number of analysed apps from each families are shown in Table 3.

To validate our method, we also downloaded 445 benign apps from known app stores such as Google Play, AppBrain[9], F-Droid[10], Getjar[11], Aptoid[12], and Mobango[13]. The benign apps are selected from different categories such as social, news, entertainment, finance, education, games, sports, music, and audio, telephony, messaging, shopping, banking, and weather to learn the normal behaviour of benign apps. Table 4 depicts the details of categories of benign apps, number of analysed apps from each category and the corresponding app stores.

Our investigation of Android security framework and analysis of benign and malware samples resulted in interesting finding: identification of key permissions and intents used for malware attacks and propagation. We also establish that certain permissions and intents which are frequently used by malware apps are seldom used by benign apps. Malware families use a particular set of permissions and intents targeting specific capabilities and resources. Almost all the malware samples belonging to that particular family use a unique set of permissions and intents.

[3]Android Malware Genome Project, Web: http://www.malgenomeproject.org/

[4]VirusTotal for Android, Web: https://www.virustotal.com/en/documentation/mobile-applications/

[5]theZoo aka Malware DB, Web: http://ytisf.github.io/theZoo/

[6]MalShare project, Web: http://malshare.com/about.php

[7]Web: https://virusshare.com/

[8]HashTab, Web: http://implbits.com/products/hashtab/

[9]Web: http://www.appbrain.com/

[10]Web: https://f-droid.org/

[11]Web: http://www.getjar.com/

[12]Web: https://www.aptoide.com/

[13]Web: http://www.mobango.com/

4.1. Permission usage by Applications

In this section, we present our findings on how the malware applications use permissions differently from benign apps. We also discuss how this distinct usage pattern may be exploited to detect malware apps. There are 58 permissions out of 145 were frequently used by the malware and benign apps, whereas remaining 87 are rarely used. In order to visualize the usage pattern of permissions in malware and benign apps, we chose the top 24 permissions and plotted there usage percentages among the malware and benign applications in Fig. 1. Although, some of the permissions are used by both the malware and benign apps, there remain a noticeable distinguishing usage pattern as shown in Fig. 1. Based on the usage pattern of permissions, we split the permissions into two groups: *Normal permissions* and *Dangerous permissions*.

The dangerous permissions are those permissions that are frequently used by malware apps and have more risk to access and exploit different sensitive resources and private data. Examples of frequently used permissions by benign apps are: Full Network access, Create/Add/remove/user accounts, Delete/Modify USB contents and Read/write/modify contacts. Malware apps usually use permissions: Read phone status/ID, Access Network state, Send SMS/MMS, Receive boot complete, Receive SMS, Delete/Modify USB contents, and your location. There are a few malware-friendly permissions, which are seldom used by the benign ones, e.g., Access Network state, Receive boot complete, Restart packages, Mount/Unmount File system, Set wallpapers, Read/write history bookmarks of browser and Write APN settings.

The most popular benign apps such as YouTube, Skype and Viber tend to use on average 8 to 16 permissions while this number goes down to 3 to 6 for the least popular apps. The same trend can be observed in malware apps. We categorize malware apps into the most harmful and the least harmful apps depending on the ease of access to sensitive resources and data regarding used permissions and intents. The most harmful malicious apps are those who are accessing more sensitive resources and data and may provide monetary damages to the users like sending premium rate SMSs, making calls, and accessing bank accounts details. The least dangerous malicious apps are those who can access some useful data and resources, but they may not cause financial or serious damage to the user or device. The most harmful malware apps use more than 16 permissions and least harmful use 3

to 6 permissions. Some permissions used by the most and least popular apps as well as the most and least harmful apps are shown in Fig. 2.

4.2. Intent usage by Applications

There are 35 intents out of 227 in Android version 4.4, which are frequently used by apps. The most popular benign apps usually use on average 1 to 4 intents and the least popular use 1 to 2. Similarly, the most harmful apps use a minimum of 5 and maximum of 8. Least harmful malware apps use at least 2 or 3 intents. Fig. 3 shows the overall trend of intents usage popular and harmful apps. Benign apps use only `ACTION_MAIN, CATEGORY_LAUNCHER and CATEGORY_DEFAULT` intents whereas malware apps usually use more intents to gain extra capabilities. Mostly malware apps use `BOOT_COMPLETED`, `ACTION_CALL`, `ACTION_BATTERY_LOW`, `SMS_RECEIVE` and `NEW_OUTGOING_CALL`.

Malware apps are seen to use a few of the normal permissions and intents while they use a significant number of dangerous permissions and intents. Benign apps show a similar trend of using only normal permissions and intents. These findings suggest that permissions and intents play a central role in accessing, controlling and sharing of sensitive data and resources. These features may be exploited to detect and mitigate the malicious attacks.

4.3. Correlation between permissions and intents

Correlation is a technique to measure the strength of association between two variables. Different correlation coefficient methods are used to measure degree of correlation. The most common is the Pearson correlation coefficient (r). It is calculated by dividing the covariance of two variables with product of their standard deviations. Pearson's correlation coefficient has a value between -1 (perfect negative correlation) and 1 (perfect positive correlation).

Suppose we have n malware applications, each application is using X dangerous permissions written as $x_i = \{x_1, x_2, \ldots, x_n\}$ and Y dangerous intents such that $y_i = \{y_1, y_2, \ldots, y_n\}$, then the Pearson correlation coefficient (r) can be calculated using equation (1).

$$r_{xy} = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n \sum {x_i}^2 - (\sum x_i)^2}\ \sqrt{n \sum {y_i}^2 - (\sum y_i)^2}}. \quad (1)$$

Two sets of malware apps are used to measure the strength of correlation between dangerous permissions and dangerous intents. One set consists of 200 malware apps which are randomly selected from different malware families and the other consists of 20 malware apps from same malware family.

For the first set, the correlation coefficient (r) equals 0.74, indicating a strong relationship between dangerous permissions and dangerous intents for the significance level: $p < 0.001$. For the other set, the correlation coefficient (r) equals to 0.94, indicating a very strong correlation between dangerous permissions and intents in the case of samples belonging to the same malware family. The strong correlation between the dangerous permissions and intents supports our conjecture about the association between permissions and intents to carry out the malicious activity.

The Pearson correlation coefficients of 0.74 for different malware families and 0.94 for same malware family confirm the positive correlation between permissions and intents. However, we need to perform a significance test to decide whether or not there is any evidence which supports or contradicts the presence of a linear correlation in the whole population of malware apps. We use the hypothesis testing, for which we test the null hypothesis, H_0, and alternate hypothesis, H_1 as

$$\begin{aligned} H_0 \ &: \ \textit{malware and benign applications use} \\ &\quad\ \textit{the same set of permissions and intents}, \\ H_1 \ &: \ \textit{malware and benign applications don}'\textit{t use} \\ &\quad\ \textit{the same set of permissions and intents}. \end{aligned}$$

For hypothesis testing, we use the Mann-Whitney $U-$test with the p-value of 0.05. We calculate U_1 and U_2 values for both the permissions and intents respectively using equations 2 and 3, respectively. In following equations, R_1 and R_2 are the sums of ranks for permissions and intents, respectively, and n_1 and n_2 are the sample sizes for both the variables.

$$U_1 = R_1 - \frac{n_1(n_1+1)}{2}; \tag{2}$$

$$U_2 = R_2 - \frac{n_2(n_2+1)}{2}. \tag{3}$$

We take the smallest of U and compare it with the critical value obtained from the Mann-Whitney critical values table Mann et al. (1947). We use Mann-Whitney critical values table for a small number of malware samples and Z-test for large samples of malware apps due to limitations of the number of entries in the Mann-Whitney critical value table. With samples from same

malware family *(n_1= 20, n_2 = 19, p=0.05, critical value = 119)*, the smallest U value obtained is 87 which is less than the critical value of 119, we would reject the null hypothesis for the malware apps belonging to same family. For a large sample of apps belonging to different malware families *($n_1 = n_2 = 200$, p=0.05, Z-critical value = 1.64)*, we calculate z-score with Z test. We obtain z-score of 13.0594 which is greater than Z-critical value hence suggesting the rejection of null hypothesis H_0. We have very strong statistical evidence to accept the alternate hypothesis H_1, which suggests that the malware and benign apps use a different set of permissions and intents. This conjecture is further verified with normal distribution testing and classification analysis using different machine learning algorithms.

The normal distribution is important for statistical inference point of view Cohen et al. (1992). We use box plots to test whether the sample distribution is normal. The box plots of permissions and intents related to benign and malware apps are shown in Figures 4 and 5, respectively. The distribution appears to be approximately normal, with the upper whiskers longer than the Q1 to median distance and the box containing middle 50% of the data almost tightly grouped in the center of distribution.

5. Malware Classification

We describe how the data is represented and then present a detailed description of our proposed system.

5.1. Data Representation

Our dataset consists of n applications from K classes with m features. Let $C = \{1, 2,..., K\}$ are the set of indices of the classes, $A = \{1, 2,..., n\}$ the set of indices of the applications and $F = \{1, 2,.., m\}$ the set of indices of the features. Also, let a_k, $k \in C$ and $a_k \subseteq A$ be the set of indices of applications belonging to class k. Additionally, let f_j, $j \in F$ be the domain of the jth feature. Let ith application, such that $i \in A$ is represented as $(c_i, f_i) = (c_i, f_{i,1}, f_{i,2},..., f_{i,m}) \in C\ x\ F_1\ x...x\ F_m$, where c_i is the class of application i such that $C \in \{malware\ ,\ normal\}$ and $(f_{i,1}, f_{i,2},..., f_{i,m})$ is the number of permissions and intents used by ith applications, and $f_{i,m} \in \{0,1\}$ which indicates if the ith application uses mth feature. We compute the Information Gain (IG) of each feature x_m against the class variable as

follows:

$$IG(F_i, C) = \sum_{f \in (0,1)} \sum_{c \in (mal,nor)} P(F_i = f; C = c).log_2 \left(\frac{P(F_i=f;C=c}{P(F_i=f)P(C=c)} \right); \quad (4)$$

Given that

$$P(F_i = f; C = c) = P(F_i = f).P(C = c|F_i = f), \quad (5)$$

Equation (4) can be simplified as

$$IG(F_i, C) = \sum_{f \in (0,1)} \sum_{c \in (mal,nor)} P(F_i = f).P(C = c|F_i = f).log_2 \left(\frac{P(C=c|F_i=f)}{P(C=c)} \right). \quad (6)$$

Using equation (6), the features with highest IG are selected to train the model.

5.2. Probability Estimation

The probability of an application belonging to a particular class is calculated using Bayesian theorem:

$$P(\overline{C} = c|\overline{F} = f) = \frac{P(C = c) \prod_{i=1}^{m} P(F_i = f_i|C = c)}{\sum_{j \in (0,1)} P(C = c_j) \prod_{i=1}^{m} P(F_i = f_i|C = c_j)}; \quad (7)$$

An app is classified as malware if

$$P(\overline{C} = malware|\overline{F} = f) > P(\overline{C} = normal|\overline{F} = f). \quad (8)$$

5.3. System Description

The proposed system is shown in Fig. 6. It consists of three main stages: Feature extraction, Pre-processing, and Classification. The feature extraction stage analyses the manifest file and extracts the permissions and intents. This stage comprises of two monitors which are used to measure: (i) type of permissions (normal or dangerous) and their numbers and, (ii) type of intents (normal or dangerous) and their number. Permissions and intents are labelled into four groups: *normal permissions, normal intents, dangerous permissions* and *dangerous intents*. Dangerous permissions and intents are frequently used by malware apps whilst normal permissions and intents are frequently used by benign apps.

The pre-processor stage processes the extracted data to generate the vector dataset in an ARFF file format. The generated dataset is randomized using unsupervised instance randomization filter for better accuracy and sent to the classifier stage. The classifier stage takes each monitored vector as input and classifies the data set using trained classifier. Six machine learning classifiers: *Nave Bayesian, Decision Tree, Decision Table, Random Forest, Sequential Minimal Optimization* and *Multi Lateral Perceptron (MLP)* are used for classification. Their performances are also compared in terms of different performance metrics. Finally, the reporter stage generates notifications for the user based on the classifier results.

6. Evaluation

6.1. Experimental Setting

The experiments are carried out on an Intel Core i7-3520 M CPU @ 2.90 GHz, 2901 MHz machine with 8GB RAM. Each of the classifiers are evaluated with two methods: *10-fold cross-validation* and *80% split*. In 10-fold cross-validation, the data set is divided into ten subsets, and the holdout method is repeated ten times. In each round, one subset is taken as test set and the remaining nine subsets are combined to form the training set. Errors of all the ten rounds are averaged out to obtain a final output. This method ensures that each instance is included at least once in the test set and nine times in the training set. The final model is the average of all ten iterations. The second method we use is 80% split, which uses 80:20 ratio (80% of a dataset for training and 20% for testing). This method is efficient but less accurate than the 10-fold method. In this section, we only report the results from 10-fold method.

6.2. Performance Comparison of different Classifiers

Performance of six classifiers is compared in terms of True Positive Rate (TPR), False Positive Rate (FPR), accuracy, F1-score and Area Under Curve (AUC). These metrics are calculated using the confusion matrix as shown in Table 5. Table 5 is generated from the four measures: True Positive (TP) —the number of correctly classified instances that belong to the class, True Negative (TN) —the number of correctly classified class instances that do not belong to the class, False Positive (FP) —instances which were incorrectly classified as belonging to the class and False Negative (FN) —instances which were not classified as class instances.

$$TPR = \frac{TP}{TP+FN}; \tag{9}$$

$$FPR = \frac{TP+TN}{TP+FN+FP+TN}; \tag{10}$$

$$Accuracy = \frac{TP+TN}{TP+FN+FP+TN}; \tag{11}$$

$$F1-Score = 2.\frac{Precision\ .\ Recall}{Precision+Recall}; \tag{12}$$

$$AUC = \frac{1}{2}\left(\frac{TP}{TP+FN}+\frac{TN}{TN+FP}\right). \tag{13}$$

Table 6 lists the TPR, FPR, Precision, F1-score, recall, AUC and model build-up time. All the analysed classifiers perform well with an accuracy of 0.90 or more. However, MLP and Decision table dominate with an accuracy of 0.99. In terms of time, Nave Bayesian, Decision Tree and Decision Table are more efficient than MLP and Random forest. Overall, Decision Table gives the best results.

6.3. Optimization with Ensemble methods

Ensemble methods combine results from multiple machine learning algorithms to improve the predictive performance Dietterich et al. (2000). It is not necessary that the performance of ensemble learning be better than the individual classifiers. The stacked performance depends on the selection of classifiers and methods used to combine the output predictions Saso et al. (2004).

We apply three ensemble methods: Boosting, bagging, and stacking to further improve the detection accuracy. Stacking gives the better results as compared to boosting and bagging. In stacking, multiple algorithms are trained individually with the training dataset and the outputs from the classifiers are sent to a meta-classifier which is trained to combine the results to makes a final prediction. Decision Table, MLP, and Decision Tree classifiers are applied in first stage and their results are combined using three schemes: an average of probabilities, a product of probabilities and majority voting.

Average of probabilities. It takes an average of the probabilities of each class from the individual classifiers (k=*3* for three classifiers) and compares which

class has greater probability such that,

$$Malware, if P_{avg} \sum_{k=1}^{3} Class_{malware} < P_{avg} \sum_{k=1}^{3} Class_{benign}; \tag{14}$$

$$Benign, if P_{avg} \sum_{k=1}^{3} Class_{malware} > P_{avg} \sum_{k=1}^{3} Class_{benign}. \tag{15}$$

Product of probabilities. Product of probabilities is taken from each of the classifiers and highest probability of class is assigned as:

$$Malware, if P_{avg} \prod_{k=1}^{3} Class_{malware} < P_{avg} \prod_{k=1}^{3} Class_{benign}; \tag{16}$$

$$Benign, if P_{avg} \prod_{k=1}^{3} Class_{malware} > P_{avg} \prod_{k=1}^{3} Class_{benign}. \tag{17}$$

Majority vote. The final result is decided based on the results obtained from the majority of the results.

Results of ensemble classification are depicted in Table 7. The product of probabilities method yields the best results.

6.4. Comparison with related approaches

We compare the performance of PIndroid against related approaches which use some of the similar features and analyzing the samples acquired from same sources: Google Playstore, Genome and Contagiodump. These are known repositories of malware and benign apps and the performance of most of the state of the art malware detection approaches are tested on these samples with a difference of number of samples tested. The most relevant approaches are Drebin (2014), DroidMat (2012) and Marvin (2015). Drebin (2014) examines the manifest file and decomposed code of app to check the permissions, API calls, hardware resources, app components, filtered intents and network addresses. It uses support vector machines (SVM) for malware

classification. Although, they used the largest dataset of 129013 apps, it consists only 4.5% of malware samples thereby may not be able to learn malware patterns. It used many features opposed to our work which uses only two most effective features. It achieved 94% malware detection rate with 1% false positive rate whereas our approach achieved 99.8% detection accuracy with 0.06 false positive rate. Drebin requires extensive processing for extraction and execution of a large number of features from the manifest file and app code, it takes more time to analyse the app and therefore is less efficient than our method. It takes on average 10 seconds to analyse an app, whereas our approach takes less than 1 second. Its use of a large number of features may also result in more false alarms as the efficiency and accuracy of feature based detection approaches highly depend on the selection of more relevant and less number of features.

DroidMat (2012) analyses some features from the manifest file and smali files of disassembled codes. The extracted features include permissions, components deployments, intent messages and API calls. It applies K-means algorithm for clustering and Singular Value Decomposition (SVD) method for low-rank approximation. The minimized clusters are processed with a kNN algorithm for classification into malware or benign apps. It achieves an accuracy of 97.6% with no reported false positive rate. They analysed 1738 apps consisting of 1500 benign and only 238 malware samples. Malware samples are only 13% of total dataset, which is a non-representative data set for capturing the malware usage patterns. The accuracy is less than our method and the processing time is higher as it needs to perform the execution of smali files and manifest files. Since Smali files are much larger than manifest files, the overall cost of methods which analyse smali files forgoes higher. This holds true for both of above solutions: Drebin and DroidMAT.

Marvin (2015) uses off-device static and dynamic analysis for malware detection. It uses around 490,000 features extracted from the manifest files and disassembled codes. Its high-dimensional feature set includes permissions, intents, API calls, network statistics, components, file operations, phone events, app developer IDs, package serial numbers and bundles of other features. It uses a linear classifier to detect malware app and assign a malicious score to the app on a scale from 0 to 10, with 0 being benign and 10 being malicious. They used the largest dataset of 150,000 apps in which only 10% are malware samples. It classifies with an accuracy of 98.24% and false positive rate of 0.04%. Although this approach classifies with the malicious score, this is not an efficient approach considering the high dimensionality

of features and regular updating requirement of the database to maintain the detection performance. Since, both the analyses are done off-device; the mobile app is just to provide an interface to upload the apk to the analysis server. The static and dynamic analyses of an app take several minutes depending on the size of smali files. This approach is less efficient and less accurate than our approach.

We further compare the detection rate of PIndroid on the unlabelled set of 100 apps against these approaches. The results are shown in Table 8. PInDroid significantly outperforms the other approaches with TPR of 0.98 and FPR of 0.1. The other approaches provide a detection rate between 0.90 to 0.93 with FPR between 0.7 to 1. Detection performance of compared approaches as Roc curve is shown in Fig. 7. These approaches are less efficient than our approach in analysing the apps due to their dual processing time. PIndroid gives good results due to the use of most relevant feature set to model the malicious behaviour.

7. Conclusion

Android security framework relies on permissions and intents to control the access to vital hardware and software resources. These two features have never been used in tandem for malware detection. This work proposes a novel malware detection method based on these two vital security features. We use statistical and machine learning methods to validate the conjecture. Our results demonstrate the potential of this approach, where PinDroid outperforms related approaches and detects malware apps accurately with very low false positive rate.

The work also compares the performance of different classifiers for their effectiveness in malware detection. Different ensemble methods are also investigated and applied on the proposed model to improve the detection accuracy. Some malware apps are also developed for Proof of Concept (PoC) and to get an insight into the modalities and complexities of malware apps development. We also investigated Android methods and found that permissions and intents are the basic features used for app collusion. Hence, our proposed malware detection model is particularly suitable for detection of colluding apps in addition to the other types of malware apps.

References

Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song and David Wagner. Android permissions demystified. *Computer and communications security, International Conference*, pp 627–638, ACM, 2011.

Amiya K Maji, Fahad Arshad, Saurabh Bagchi, Jan S Rellermeyer. An empirical study of the robustness of inter-component communication in Android. In *Dependable Systems and Networks (DSN), International Conference*, pp 1–12, IEEE, 2012.

Andrea Avancini and Mariano Ceccato. Security testing of the communication among Android applications. In *Proceedings of the 8th International Workshop on Automation of Software Test*, pp 57–63, IEEE, 2013.

Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. andromaly: a behavioral malware detection framework for Android devices. *Journal of Intelligent Information Systems*, 38(1): pp 161–190, 2012.

Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: a perspective combining risks and benefits. In *17th ACM symposium on Access Control Models and Technologies*, pp 13–22, ACM, 2012.

Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *Computer Security-ESORICS*, pp 163–182, Springer, 2014.

Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon and Konrad Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. NDSS, 2014.

Daniele Gallingani, Rigel Gjomemo, VN Venkatakrishnan and Stefano Zanero. Practical Exploit Generation for Intent Message Vulnerabilities in Android. *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pp 155–157, ACM, 2015.

David Barrera, H Güneş Kayacik, Paul C van Oorschot and Anil Somayaji. A methodology for empirical analysis of permission-based security models

and its application to Android. *Computer and communications security, 17th ACM Conference on*, pp 73–84, ACM, 2010.

David Kantola, Erika Chin, Warren He, and David Wagner. Reducing attack surfaces for intra-application communication in Android. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pp 69–80, ACM, 2012.

Dimitris Geneiatakis, Igor Nai Fovino, Ioannis Kounelis, and Paquale Stirparo. A permission verification approach for Android mobile applications. *Computers & Security*, 49: pp 192–205, 2015.

Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security, Seventh Asia Joint Conference on*, pp 62–69, IEEE, 2012.

Dragos Sbîrlea, Michael G Burke, Salvatore Guarnieri, Marco Pistoia, and Vivek Sarkar. Automatic detection of inter-application permission leaks in Android applications. *IBM Journal of Research and Development*, 57(6): pp 10–1, 2013.

Dzeroski Saso and Zenko Bernard. Is combining Classifiers with Stacking better than selecting the best one. *Machine Learning*, 9:(54), pp 255–273, 2004.

Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pp 239–252, ACM, 2011.

Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru and Ian Molloy. Using probabilistic generative models for ranking risks of Android apps. *Computer and communications security, Conference proceedings on*, pp 241–252, ACM, 2012.

Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pp 50–60, 1947.

Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for Android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pp 15–26. ACM, 2011.

Hyunjae Kang, Jae-wook Jang, Aziz Mohaisen, and Huy Kang Kim. Detecting and classifying Android malware using static analysis along with creator information. *International Journal of Distributed Sensor Networks*, pp 1–9, 2015.

Jacob Cohen. A power primer. *Psychological bulletin*, 112(1): pp 155, 1992.

Kabakus Abdullah Talha, Dogru Ibrahim Alper and Cetin Aydin. APK Auditor: Permission-based Android malware detection system. *Digital Investigations, Elsevier Journal on*, PP(13): pp 1–14, Elsevier, 2015.

Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang and David Lie. Pscout: analyzing the Android permission specification. *Proceedings of conference on Computer and communications security*, pp 217–228, ACM, 2012.

Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee and Guofei Jiang. Chex: statically vetting Android apps for component hijacking vulnerabilities. *Proceedings of conference on Computer and communications security*, pp 229–240, ACM, 2012.

Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *Information Security*, pp 346–360, Springer, 2011.

Martina Lindorfer, Matthias Neugschwandtner and Christian Platzer. MARVIN: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis. *Computer Software and Applications, 39th Annual Conference*, pp 422–433, IEEE, 2015.

Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen and Martin C Rinard. Information Flow Analysis of Android Applications in DroidSafe. pp 1–16, NDSS, 2015.

Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Gaur, Mauro Conti and Raj Muttukrishnan. Android security: A survey of issues,

malware penetration and defenses. *Communications Surveys & Tutorials*, 17(2): pp 998–1022, IEEE, 2014.

Thomas G Dietterich. Ensemble methods in machine learning. In *Multiple classifier systems*, pp 1–15, Springer, 2000.

Timothy Vidas and Nicolas Christin. Evading Android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp 447–458, ACM, 2014.

William Enck, Machigar Ongtang and Patrick McDaniel. On lightweight mobile phone application certification. *Computer and communications security*, pp 235–245, ACM, 2009.

William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android security. *IEEE security & privacy*, (1): pp 50–57, IEEE, 2009.

Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang and Binyu Zang. Vetting undesirable behaviors in Android apps with permission use analysis. *Computer & communications security*, pp 611–622, ACM, 2013.

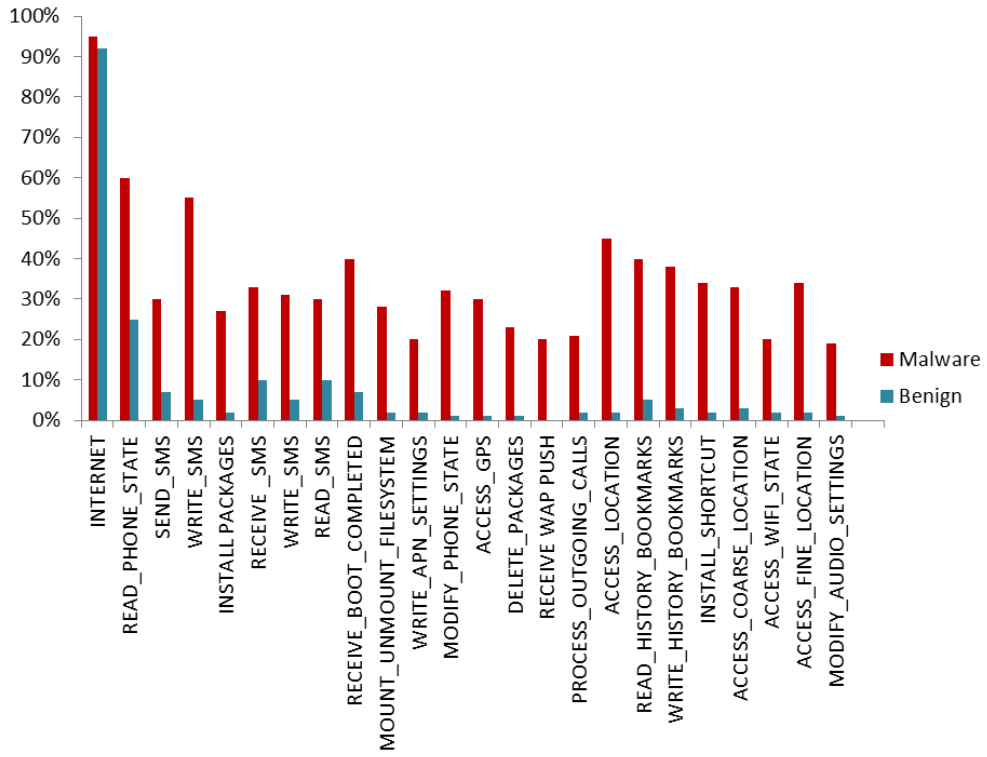

Figure 1: Frequently used permissions by malware applications

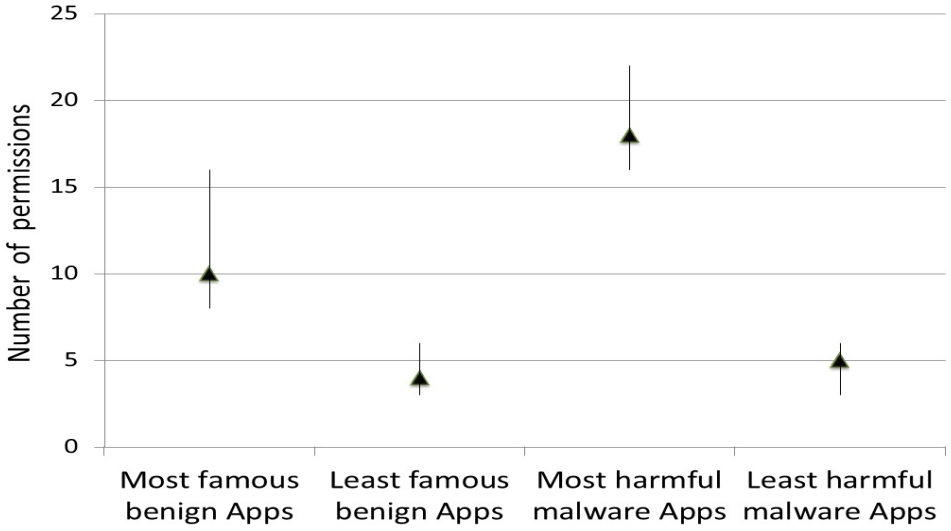

Figure 2: Number of permissions used by malware and benign applications

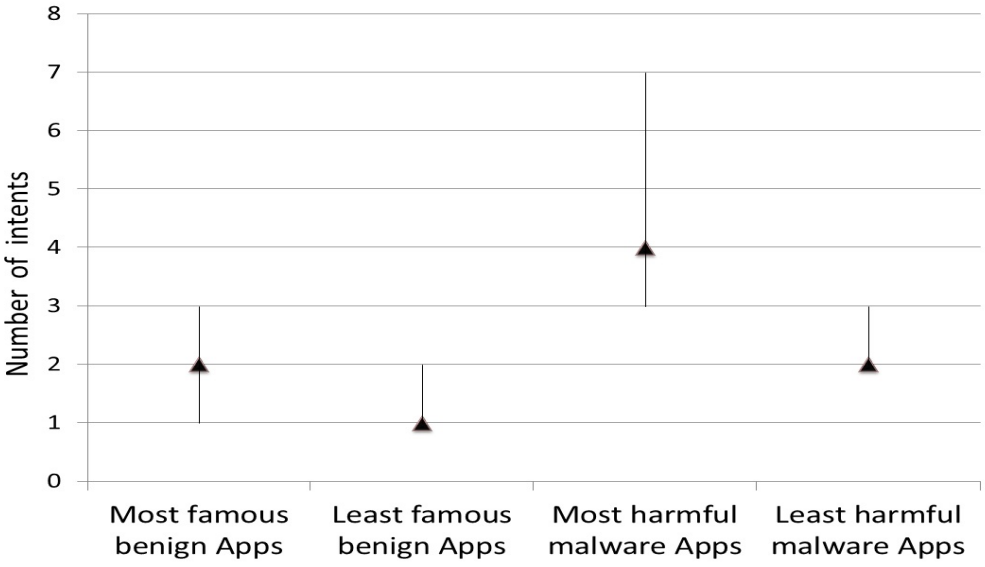

Figure 3: Number of intents used by malware and benign applications

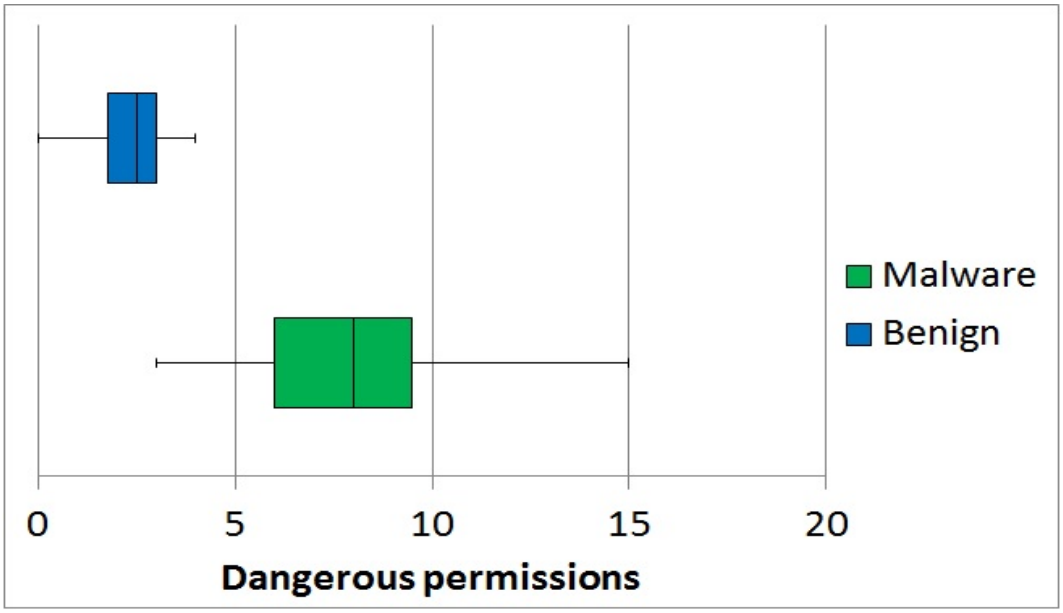

Figure 4: Box-plots for dangerous permissions

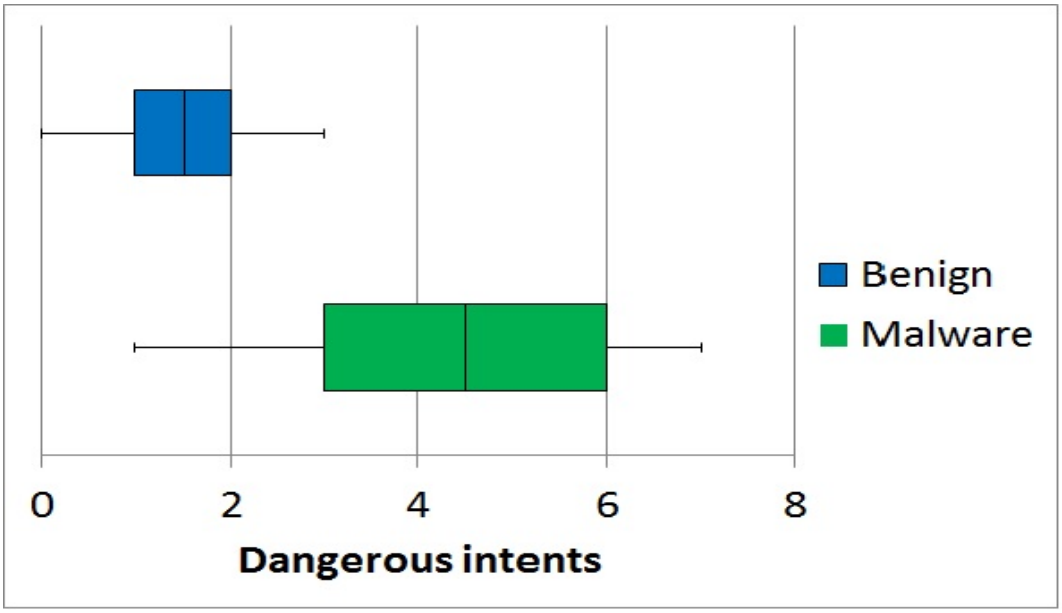

Figure 5: Box-plots for dangerous intents

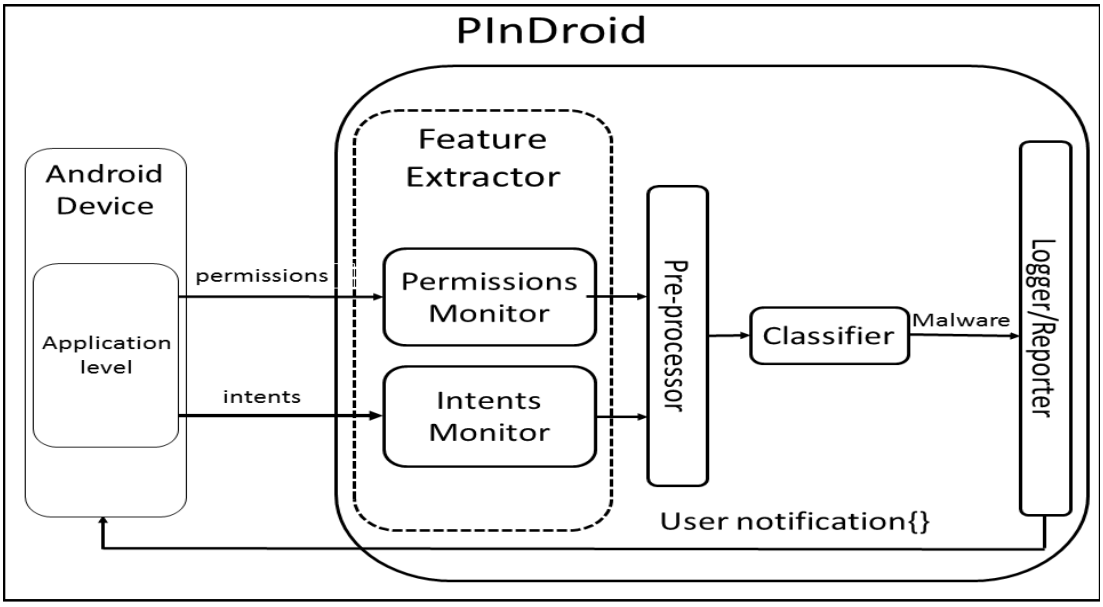

Figure 6: Diagram of proposed system

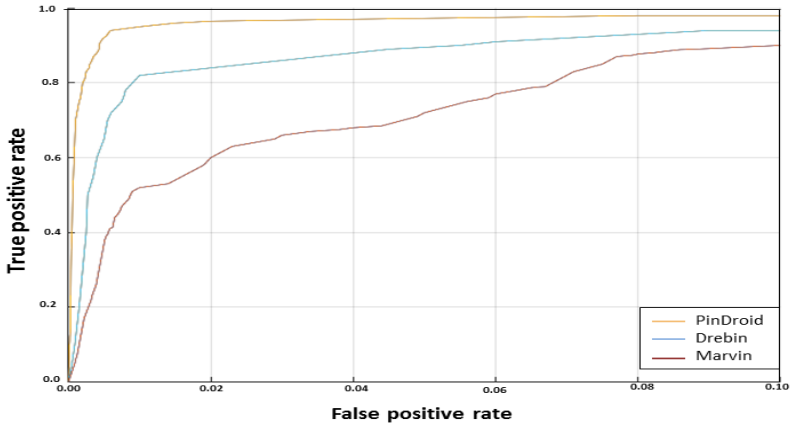

Figure 7: Detection performance of related approaches as ROC curve

Table 1: Number of permissions and intents in API levels

API Level	No of Permissions	No of Intents
23	135	252
22	124	243
21	123	238
20	118	227
19	117	227
18	106	221
17	103	214
16	103	203
15	99	201
14	99	191
13	97	180
12	96	180
11	96	176
10	96	167
9	95	167
8	92	167
7	88	161
6	88	158
5	88	158
4	87	146
3	83	136
2	78	124
1	76	124

Table 2: Sources of Malware samples

Source	**No of Malware samples used**
Contagio	60
Drebin	100
Genome	1000
VirusTotal	70
theZoo	20
MalShare	25
VirusShare	25

Table 3: Number of malware samples collected from different sources

Malware family	**No of samples analyzed**	**Malware type**
Basebridge	11	Botnet, Information stealing
DroidKungFu	11	Botnet, Information stealing
DroidKungFu	10	Botnet, Information stealing, Backdoor
FakeDolphin	4	Adware
Locker	2	Ransomware
VDLoader	3	Backdoor, Information stealing
FakeBank	5	Trojan Banker, Money stealing, Information stealing
GinMaster	7	Information stealing, Backdoor
Boxer	2	Sends SMS
JIFake	3	Sends SMS
SNDApps	1	Information stealing
OpFake	4	Sends SMS
FakeInst	3	Installer
FakePlayer	3	Sends SMS
BgServ	7	Botnet, Information stealing, Botnet, Trojan Installer, backdoor
Plankton	7	Money stealing, Botnet, Information stealing, Backdoor, Trojan installer
Geinimi	9	Botnet, Information stealing, Root access
AnserverBot	13	Information stealing
PjApps	9	Botnet, backdoor
GoldDream	10	Trojan, Information stealing
DroidSheep	7	Session hijacker
CopyCat	4	Adware
DroidDream	10	Information stealing, Adware
DroidKungFu	11	Botnet, Information stealing, Root access
Keji	4	Information stealing, Trojan Installer
HolyBible	5	Adware, Backdoor
Obad	2	Botnet, Information stealing, Botnet, Trojan Installer, backdoor, SMS sending, Location
Nickispbby	5	Spying, Information stealing
RuFraud	3	SMS sending
Jsmshider	3	Information stealing
Zitmo	3	Money, Information stealing, Backdoor
AngryBird	13	Botnet, Information stealing
KMin	10	Exploit, Information stealing

Table 4: Categories of benign apps

Category	**No of samples used**	**Apps Market**
Social Media	11	Google Play store
Mail	4	Google Play store
Education	10	Google Play store
Banking	4	Google Play store
Entertainment	15	Google Play store
Sports	8	Google Play store
Shopping	6	Google Play store
Finance	6	Google Play store
News	8	Google Play store
Weather	8	Google Play store
Games	15	Google Play store
Medical	10	Google Play store
Fitness	11	Google Play store
Media	11	Google Play store
Casual	15	Google Play store
Music	15	Google Play store
Books	5	Google Play store
Travel	5	Google Play store
Lifestyle	15	Google Play store
Simulation	7	Google Play store
Transportation	4	Google Play store
Misc	15	AppBrain
Misc	10	F-Droid
Misc	10	Getjar
Misc	15	Aptoid
Misc	15	Mobango

Table 5: Confusion Matrix

Actual Class	**Classified as Malware**	**Classified as Benign**
Malware	TP	FN
Benign	FP	TN

Table 6: Comparison of Classification Algorithms

Algorithm	**TPR**	**FPR**	**Precision**	**F1Score**	**Recall**	**AUC**	**Time**
MLP	0.993	0.006	0.995	0.995	0.995	0.996	1.18
Decision Table	0.993	0.006	0.995	0.996	0.996	0.996	0.23
Decision Tree	0.992	0.011	0.993	0.992	0.993	0.992	0.01
Nave Bayesian	0.982	0.012	0.989	0.988	0.989	0.997	0.01
Random Forest	0.982	0.007	0.985	0.985	0.985	0.989	0.43
SMO	0.952	0.033	0.956	0.956	0.956	0.978	0.24

Table 7: Comparison of Ensemble results

Method	**TPR**	**FPR**	**Precision**	**F1Score**	**Recall**	**AUC**
Average Probability	0.972	0.012	0.975	0.975	0.975	0.982
Product Probability	0.998	0.011	0.998	0.997	0.997	0.998
Majority Vote	0.982	0.021	0.986	0.986	0.986	0.989

Table 8: Comparison with related approaches

Method	**TPR**	**FPR**	**Average Time (sec)**
PIndroid	0.984	0.1	1
Drebin	0.932	0.9	10
Marvin	0.918	0.7	9
Droidmat	0.903	1.0	9