PMDS:
Permission-based Malware Detection System

Paolo Rovelli[1] and Ymir Vigfusson[1,2]

[1] School of Computer Science
Reykjavik University
Menntavegi 1, Reykjavik 101, Iceland
[2] Department of Mathematics and Computer Science
Emory University
400 Dowman Drive, Atlanta GA 30322, USA

Abstract. The meteoric growth of the Android mobile platform has made it a main target of cyber-criminals. Mobile malware specifically targeting Android has surged and grown in tandem with the rising popularity of the platform [3, 5, 4, 6]. In response, the honus is on defenders to increase the difficulty of malware development to curb its rampant growth, and to devise effective detection mechanisms specifically targeting Android malware in order to better protect the end-users.
In this paper, we address the following question: do malicious applications on Android request predictably different permissions than legitimate applications? Based on analysis of 2950 samples of benign and malicious Android applications, we propose a novel Android malware detection technique called Permission-based Malware Detection Systems (PMDS). In PMDS, we view requested permissions as behavioral markers and build a machine learning classifier on those markers to automatically identify for unseen applications potentially harmful behavior based on the combination of permissions they require. By design, PMDS has the potential to detect previously unknown, and zero-day or next-generation malware. If attackers adapt and request for fewer permissions, PMDS will have impeded the simple strategies by which malware developers currently abuse their victims.
Experimental results show that PMDS detects more than 92–94% of previously unseen malware with a false positives rate of 1.52–3.93%.

Keywords: Android, Permissions, Malware Detection System, Machine Learning, Data Mining, Heuristics

1 Introduction

Mobile devices are being adopted at an exponential rate: mobile phone subscriptions have increased from 700 million in 2000 to 7 billion by the end of 2014, representing more than 96% of the world's population, with the market penetration in developing world projected to reach 90% by the same time [1]. The number of smartphones in use worldwide have surpassed one billion-unit for

the first time ever in second half of 2012 [7], and already in 2013 did the total number of smartphones shipped exceed that of feature phones [8].

The convenience of interactive mobile devices has enticed their users, who now carry a wealth of sensitive information around with them: personal data, bank information and account details, GPS location, contacts, text messages and emails [9–13]. The value of these data has attracted cyber-criminals who invest time and money in exploiting vulnerable mobile platforms, commonly through malware.
Google's Android platform has become the most targeted mobile operating system, likely for two key reasons [14]. On one hand, Android is a ubiquitous platform, with more than 1.9 billion installed-base devices [2]. On the other hand, Android applications are easy to reverse-engineer and can be readily modified or repackaged. Since attackers focus their energy on targets that have the highest return on investment, popular platforms like Android with accessible inner workings are doomed to attract special attention from cyber-criminals.

In anticipation of malicious behavior, the fundamental Android design includes various security and authentication mechanisms. One of the fundamental mechanisms is application permissions, where newly installed applications will ask the user for approval on what types of access will be required for the program to work. The mechanism enables granular control of restrictions into what specific operations can be performed by the particular application.

Yet the old adage that security is only as strong as the weakest link continues to apply: many families of Android malware prey on unsuspecting users by camouflaging themselves as regular applications that need elevated privileges to the system. Often times, legit applications are "repackaged" with a Trojan payload that preys on unsuspecting users or steals the ad revenue from the host application [18, 19]. Researchers from the App Genome project reported that 11% of the Android applications in two alternative China-based markets were repackaged, and another study has reported an alarming 5-13% repackaging rate among six different third-party markets [20]. The effectiveness of the permissions mechanism is compromised if the user fails to notice unusual requests for permissions.

In this paper, we focus on a basic question: *to what extent can Android malware be detected and thereby thwarted by solely focusing on the permissions they request?* We compare 1450 malware samples to 1500 benign applications from the Google Play Store and analyze differences and patterns between the two groups. We found correlation between the group of permissions required by an application with its behavior, that is whether the application was benign or malicious. Using this link, we propose a novel Android malware detection technique, called Permission-based Malware Detection Systems (PMDS). PMDS uses a machine learning classifier to automatically identify (potentially) dangerous behavior of previously unseen applications based on the combination of permissions they require. Through a machine learning approach, PMDS has the potential to detect previously unknown and zero-day or next-generation malware.

The contributions presented in this paper are the following.

– We propose PMDS: a simple, novel approach to categorize the behavior of an Android application and consequently to detect malware.
– We present a low-overhead cloud-based architecture for PMDS, detailing both the client-side and a server-side applications, which uses the previously presented Android malware detection technique.
– We demonstrate the feasibility of our system through cross-validation on 2950 real-world samples, showing that PMDS was able to detect 92–94% of previously unseen Android malware at 1.52%–3.93% false positive rate.

2 Background

Android is an operating system, primarily designed for touchscreen mobile devices, including smartphones and tablets, the core of which is built on top of a modified version of the Linux kernel. Every application on Android is run as a different user in its own, separate Linux process [38]. Above the Linux kernel layer, Android provides native user-space libraries, such as *OpenGL* and *WebKit*, and the Dalvik Virtual Machine (VM): an open source Virtual Machine optimized to run Java applications on mobile devices. On top of the Android system architecture there are the applications, which run on an application framework composed of Java-compatible libraries based on Apache Harmony.

Android applications are distributed and installed using the Android PacKage (APK) file format: an archive file built on the ZIP file format and carries the *.apk* file extension. Since each Android application runs in its own process with its own instance of the Dalvik VM, all application code runs in isolation from other applications.

Furthermore, each application in the Android architecture has access only to the components it requires to complete its work, and none other. Consequently, an application that wishes to access particular system components must request specific permission. The design creates a secure operating environment in which applications cannot access parts of the system unless they are explicitly granted privileges [38, 39].

An application that requires special privileges must explicitly declare those in an *AndroidManifest.xml* file, an entry file bundled within APK packages that provides semantic-rich information about the application itself and its components.

Furthermore, Android applications (as well as libraries) can enforce their own, custom permissions. These custom permissions are also declared in the *AndroidManifest.xml* file together with the system ones. [40, 39]

Because each permission is related to an action, the permissions required by an application can be seen as an indicator of its *possible*[3] future behavior and thus the risk manifested by granting the privileges to the program at hand.

We note that not every piece of malware asks for a dangerous combination of permissions. Some rely on exploiting vulnerabilities in other services to escalate

[3] It is important to note that the declaration of certain permissions in the *AndroidManifest.xml* file does not necessarily imply their use at runtime.

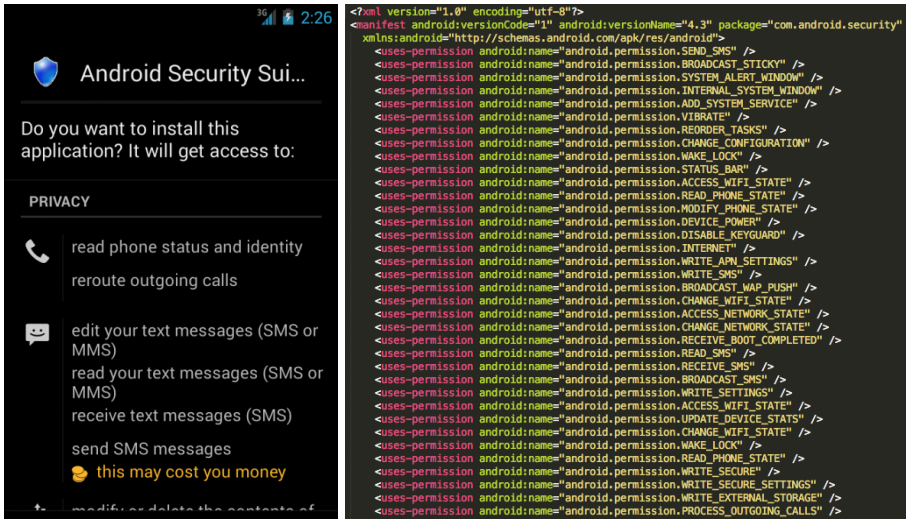

Fig. 1. Example of a malicious application (*Trojan-Banker://Android/ZitMo.B*) which requires for a specific group of permissions. On the right the permissions are required during the installation process, while on the left the *AndroidManifest.xml* file in which the required permissions are declared.

privileges, and may not require for any special permissions at all [14]. Zhou *et al.* showed that bundling exploits was the exception rather than the norm in modern Android malware. Our focus will be on malware that specifically requests permissions to undertake undesirable behavior.

3 Design and Implementation

The key goal of our study is to understand whether the group of permissions required by an application correlates with its ultimate behavior as either benign or malicious. If there is a correlation, how can it be leveraged to automatically identify potentially dangerous behavior of previously unseen applications? In this section we describe design decisions and implementation of a detection framework architecture that exploits a classifier based on application permissions.

3.1 Design

A malware detection framework for Android can be architected in a variety of ways. A proactive approach would be to embed detection for all users into an Android marketplace, such as Google Play Store or an alternative third-party market, and perform a scan when developers upload their code. However, such changes are dramatic and should only be issued after the methods have been thoroughly validated. Here, we instead describe a proof-of-concept architecture focused on protecting individual clients.

By taking a client-centric view, the next issue is to address the trade-off between overhead and accuracy. On one end of the spectrum, the detection logic could reside entirely on the mobile device, with updates periodically issued from an Internet service. This option requires more processing and in turn consumes more battery from the device. It also leaves open a window of vulnerability between updates during which malicious applications could be installed without warning. On the opposite end, one could outsource detection entirely to a cloud service, leaving a bare bones client-side framework that focuses solely on mitigation when malicious applications are detected. This strategy allows more elaborate and powerful detection programs to be run in the cloud where resources are less constrained. However, communication with a remote service incurs higher latency and may degrade user experience if the application cannot start until it has been scanned, or security if the application starts before the response from the scan has been received from the cloud server.

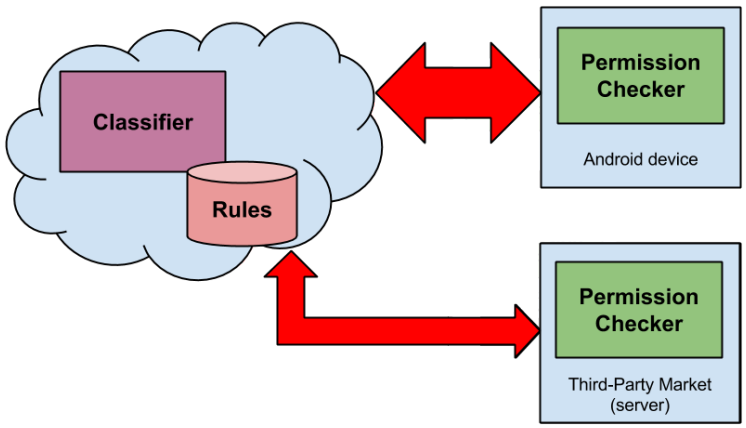

Fig. 2. The Permission-based Malware Detection System (PMDS) architecture. The clients, whether they are Android devices or Android marketplaces, have a "Permission Checker" that extracts permissions by an application and sends them to a server-side application where the application is evaluated as benign or malicious.

Taking these factors into consideration, PMDS is implemented as a cloud system. Such system has several additional advantages. An important one is that the cloud service is no longer subjected to the Android architecture design and its limitations, allowing more accurate analysis that are poised to improve detection rates. As mentioned above, the second main advantage is that fewer resources are spent on the device itself, which in turn will improve the battery lifetime compared to an on-device scan.
As shown in Figure 2, the Permission-based Malware Detection System (PMDS) is composed of a client-side application, whose task is to extract the permissions of an application and send them to the server-side component. The server is the

core of our technology, as it is tasked with classifying the behavior of the given application instance as either benign or malicious and, then, signaling the results to the client-side application as quickly as possible. In order to do so, we will rely on machine learning classifiers that we discuss below.

As with other malware detection systems, the client-side component can implement two strategies: it can provide an interface that enables the user to perform a manual scan of installed programs (on-demand scan), or it can be used to automatically scan recently obtained applications (on-install scan). Anecdotal evidence suggests that people are not vigilant about virus and malware scans, making time between scans larger than needed, so we focus on on-install scans.

However, the standard Android platform is designed to disallow applications from interacting with the installation process. Specifically, when an application is installed from Google Play Store or another third-party market, our malware scanner is limited to registering and then handling an event telling us that a new application has been installed, rather than interrupting the process. In light of this limitation, our on-install scanner will be able to detect a malicious application only after it has been installed, creating a race condition between execution of the new program and the scan. If the scan is fast enough, PMDS will detect and remove a malicious application before it is launched and thus before it can do harm. Our implementation is guided by the concern that lookups must produce a prompt response, meaning that we strive to minimize server-side computation time and network latency. On a positive note, the asynchronous notifications about newly installed applications implies that no particular delay is added to the installation process, thereby minimizing user interruption.

Finally, we must be mindful of two additional concerns when designing a cloud system for mobile devices. First, one cannot trust that mobile devices is connected to the global Internet at all possible times. We will make the assumption that applications are generally installed from markets on the Internet, and that the connection will remain up for a short amount of time following the installation. In fact, our analysis below confirms that most applications require Internet access to function properly. The second concern is that applications should strive to minimize network traffic. The communication protocol we implemented is designed to communicate the required information efficiently, as described below.

3.2 Implementation

Server-side Application The server-side application is the crux of our Permission-based Malware Detection System (PMDS) as it is the component responsible for classifying an application's behavior as either benign or malicious.

In our prototype of the server-side application, we use Python to automatically extract the permissions declared by an application in its *AndroidManifest.xml* file of the APK package (see Figure 3). We piggyback on the Android Asset Packaging Tool (*aapt*) which is a part of the Android SDK. The *aapt* program is versatile, in particular the *"aapt dump permissions"* command lists the

permissions declared by an application:

```
$ aapt dump permissions MyPkg.apk | sed 1d | awk '{print \$NF}'
```

After extracting the permissions, the program automatically save the information in the Weka's Attribute-Relation File Format (ARFF) [43, 44].
Our prototype then uses Weka [33] to train multiple classifiers in order to detect new and unseen malware.

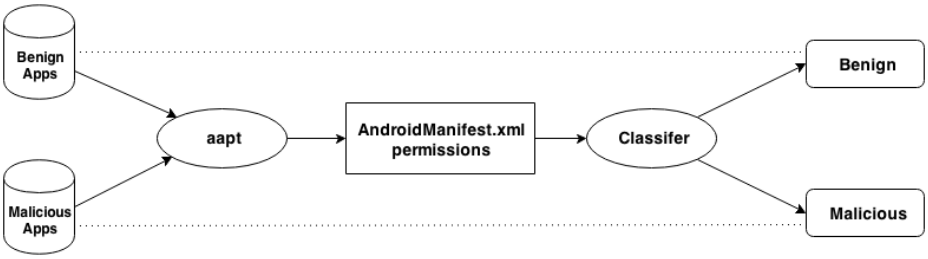

Fig. 3. To create the classifier's dataset, the permissions declared in the *AndroidManifest.xml* file of an application are automatically extracted using the Android Asset Packaging Tool (*aapt*). Then, the classifier automatically labels the application behavior, as either benign or (potentially) malicious, according to the combination of permissions the application requires.

Android applications can enforce their own custom permissions, as mentioned earlier, which appears as noise in the classifier. In PMDS, we omit custom permissions and collect only the system permissions available in the Android API documentation, for a total of 130 permissions [41]. The permissions requested by an application is represented as a binary vector of Boolean values, one for each permission, where *TRUE* stays for the presence of that particular permission while *FALSE* stays for its absence. In our database, we also store the application behavior, saved as either *benign* or *malicious*.

```
FALSE,FALSE,FALSE,FALSE,FALSE,TRUE,FALSE,TRUE,FALSE,TRUE,TRUE,TRUE,
TRUE,...,FALSE,FALSE,FALSE,malicious
```

As an optimization, we store permissions in a fixed-length array rather than an extensible map. Modifying the permission types, such as by adding new permissions, or order can compromise the integrity or prediction quality of the classifiers. The optimization allows us to compress the data for each application into no more than a bit for each permission.

In order to automatically and properly label the behavior of previously unseen applications (as either benign or malicious), we train a representative classifier from each of four modern approaches for classifiers. We chose the following

machine learning algorithms: C4.5 Decision Tree, K*, RIPPER and Naïve Bayes, and evaluate their performance in Section 4.

C4.5 Decision Tree-based Learning. We used the J48 open source Java implementation of the Ross Quinlan's C4.5 *Decision Tree-based learning algorithm* which is available in Weka. The C4.5 algorithm builds decision trees from a set of training data using information entropy as a distance measure. The decision trees are then used as predictive models for mapping observations about an item – its features – and conclusions about the item's target value – its class label. [34, 32, 33]

K* Lazy Learning. On the other hand, K* is a *Lazy (Instance-based) learning algorithm* developed by Cleary and Trigg. The K* algorithm classifies an instance by comparing it to a database of pre-classified examples and using entropy as a distance measure. The biggest drawback is that evaluation is comparatively slow and can grow with data, in exchange for faster training times. [35]

RIPPER Rule-based Learning. RIPPER (Repeated Incremental Pruning to Produce Error Reduction) is a *Rule-based learning algorithm* developed by William W. Cohen. The RIPPER algorithm classifies an instance according to a sequence of Boolean clauses linked by logical `AND` operators, which together imply the membership of the instance to a particular class. [31]

Naïve Bayes Learning. Finally, Naïve Bayes is a simple *Bayesian learning algorithm* developed by John and Langley. The Naïve Bayes probabilistic algorithm is based on applying Bayes' theorem with strong (naïve) independence assumptions. In other world, for this algorithm, the presence (or absence) of a particular feature is unrelated to the presence (or absence) of any other feature of a class. [36]

3.3 Client-side Application

The client-side component is responsible for extracting the permissions declared by an application and send them to the server-side application.

Retrieving the list of applications currently installed on an Android device – irrespective of whether they are user or system applications (see Section 2) – can be achieved by *PackageManager* class. The interface further enables the caller to retrieve various extra information related to each application, including the permissions they require [42].

Recall that Android explicitly prohibits tampering with the installation process of an application from Google Play Store. PMDS instead registers a handle for the event that a new application has been installed. A custom on-install scanner will scan the executable, but notably malware can only be detected *after* the application has been installed. However, the detection is designed to be faster than an average user is at opening the newly installed application.

We "hook" our scanner to the installation process by creating and registering a *BroadcastReceiver* handler for the *PACKAGE_ADDED* and *ACTION_PACKAGE_REPLACED* broadcast actions:

The first action is broadcast every time a new application package has been installed on the device, while the latter is called every time a new version of an

```
final PackageManager pm = getPackageManager();
final List<ApplicationInfo> listOfInstalledApps =
pm.getInstalledApplications(PackageManager.GET_META_DATA);

// Retrieve each installed app info:
for ( final ApplicationInfo ai : listOfInstalledApps ) {
  final String appName = pm.getApplicationLabel(ai).toString();
  final String appPackage = ai.packageName;
  List<String> appPermissions;

  final PackageInfo pi = pm.getPackageInfo(appPackage,
  PackageManager.GET_PERMISSIONS);

  if ( (pi != null) && (pi.requestedPermissions != null) ) {
    Collections.addAll(appPermissions, pi.requestedPermissions);
  }

  ...
}
```

Fig. 4. Assembling a list of installed applications on Android. The *getInstalledApplications()* method of the *PackageManager* class returns a list of *ApplicationInfo* objects. Each of these objects represents an installed application.

```
<receiver android:name="com.example.onInstallBroadcastReceiver"
          android:exported="false">
  <intent-filter android:priority="1000">
    <action android:name="android.intent.action.PACKAGE_ADDED" />
    <action android:name=
     "android.intent.action.ACTION_PACKAGE_REPLACED" />
    <data android:scheme="package" />
  </intent-filter>
</receiver>
```

Fig. 5. XML. Example of *BroadcastReceiver* that handles the on-install event, thanks to the *PACKAGE_ADDED* and *ACTION_PACKAGE_REPLACED* broadcast actions.

application package has been installed, thus replacing an existing version that was previously installed.

When the `BroadcastReceiver` is triggered, the permissions required by the installed APK package are extracted by the client-side component and sent to the server-side application.

The use of `BroadcastReceiver` allows us to minimize the use of the CPU and, therefore, the battery consumption. Since PMDS uses only the 130 system permissions available in the Android API documentation [41] on the server side, we can also optimize the packet exchange so that the client-side application needs transmit only a data sequence of 130 bits that indicates each permission

```java
public class onInstallBroadcastReceiver
             extends BroadcastReceiver {

  /**
   * Receiving an Intent broadcast.
   *
   * @param context  the Context in which the receiver is running.
   * @param intent  the Intent being received.
   */
  @Override
  public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();

    if ( action.equals( Intent.ACTION_PACKAGE_ADDED ) ||
         action.equals( Intent.ACTION_PACKAGE_REPLACED ) ) {
      // Retrieve the installed/updated package:
      final String appPackage =
      intent.getData().getEncodedSchemeSpecificPart();

      // [...] Scan the APK package [...]
    }
  }
}
```

Fig. 6. Java. Example of *BroadcastReceiver* handler for the on-install event, corresponding to the XML code in Figure 3.3.

requested by the application. The server-side application will answer with a single bit, predicting that the application intention is either malicious or benign. Note that other layers, such as TCP, the HTTPS protocol and other chatty formats, add additional space overhead on top of the protocol.

4 Evaluation

Since it is the core of our Permission-based Malware Detection System (PMDS), our evaluation is focused on determining the efficiency of the server-side classifier.

4.1 Dataset and analysis

We use a dataset of 2950 samples, divided into 1500 unique benign samples (*i.e.* no updated versions of the same applications are included) and 1450 malicious ones. The benign samples were collected from the Google Play Store [46], while all the malicious samples were taken from both the Android Malware Genome Project [14] and Contagio Mobile [45]. We disregard applications that do not request additional permissions.

Figure 7 shows the most frequently requested permissions by the samples in our dataset, both benign and malicious, ordered by decreasing popularity. The *INTERNET* privilege was the most required permission for both benign and malicious applications, in concord with Zhou and Jiang's study [14]. On the other hand, our dataset shows significant difference between the groups in certain permissions, such as: *READ_PHONE_STATE*, *ACCESS_WIFI_STATE*, *READ_SMS*, *WRITE_SMS*, *SEND_SMS*, *RECEIVE_SMS*, *READ_CONTACTS* and *CALL_PHONE*.

4.2 Experimental set-up

In order to evaluate the accuracy of each machine learning classifier, we use the standard tenfold cross-validation. Cross-validation is a model validation method that divides data into two segments: one used to train the machine learning algorithm and one used to test it. Tenfold cross-validation takes 90% of the dataset for training and 10% for testing, and then repeats the procedure 10 times with different parts used for training and testing, and then outputs the average accuracy across the runs. In this way, we are testing the classifiers against previously unseen data (i.e. data not used for training the classifiers), which in our case represent zero-day or next-generation malware.

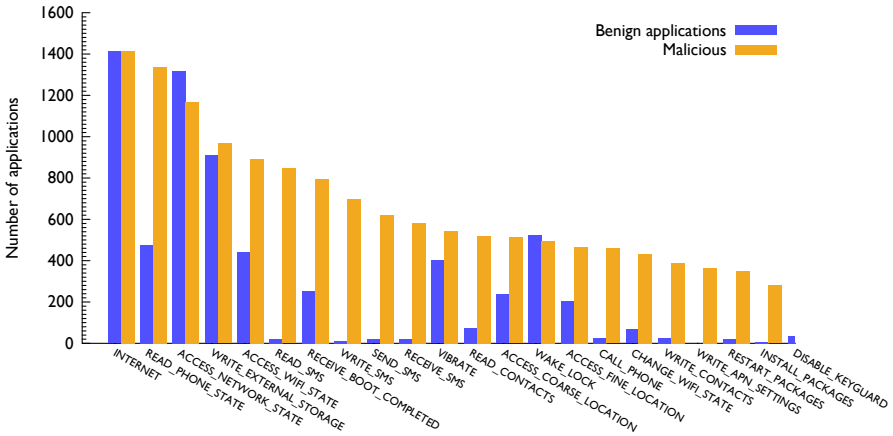

Fig. 7. Most frequently requested permissions by the Android applications in our dataset. The blue bars show the number of times the specific permissions have been requested by benign applications, while the orange bars show the number of times they have been requested by malicious applications.

In order to evaluate the results of the performed experiments, we use the following standard evaluation measures: *True Positive* (TP) – the number of applications correctly classified as malicious, *True Negative* (TN) – the number of

applications correctly classified as benign, *False Positive* (FP) – the number of applications mistakenly classified as malicious, and *False Negative* (FN) – the number of applications mistakenly classified as benign. Furthermore, we define the normalized term *True Positive Rate* (TPR) to mean the fraction of truly benign samples that were characterized as such (*i.e.* $TPR = \frac{TP}{TP+FN}$), and the term *False Positive Rate* ($FPR = \frac{FP}{FP+TN}$). We also define *Accuracy* (ACC) to mean the fraction of applications correctly classified out of the total amount of applications (*i.e.* $ACC = \frac{TP+TN}{TP+TN+FP+FN}$) and, finally, with the term *Error Rate* (ER) we mean the fraction of applications mistakenly classified out of the total amount of applications (*i.e.* $ER = \frac{FP+FN}{TP+TN+FP+FN}$). To illustrate the efficacy of a solution, we will use *Receiver Operating Characteristic (ROC) curves*, graphical plots of the TPR versus the fraction of false positives out of the total actual negatives (*i.e.* $\frac{FP}{TN+FP}$), at various threshold settings.

4.3 Standard machine learning classifiers

Table 1, and Figures 8 and 9, show the results obtained in our first batch of experiments, where we used the four representative machine learning algorithms (C4.5, K^*, RIPPER and Naïve Bayes).

	TP	**TN**	**FP**	**FN**	**TPR**	**FPR**	**ACC**	**ER**
C4.5	1340	1456	44	110	92.41 %	3.03 %	94.78 %	5.22 %
K*	1338	1478	22	112	92.28 %	1.52 %	95.46 %	4.54 %
RIPPER	1338	1465	35	112	92.28 %	2.4 %	95.02 %	4.98 %
Naïve Bayes	1155	1450	50	295	79.66 %	3.45 %	88.31 %	11.69 %

Table 1. Experimental results using four different classifiers: a *Decision Tree-based learner* (C4.5), a *Lazy Instance-based learner* (K^*), a *Rule-based learner* (RIPPER) and a *Bayesian learner* (Naïve Bayes). The classifiers automatically label the behavior of previously unseen applications as either benign or malicious.

As evident on the figures, the overall best results were obtained using K^*, with which we achieved a detection rate of 92.28% and a false positives rate of 1.52% – the minimum across all experiments. We achieved the highest detection rate (92.41%) using C4.5, while the highest accuracy (95.02%) using RIPPER. Naïve Bayes produced the least competitive results, both in terms of the detection rate and the false positive rate.

4.4 Boosted machine learning classifiers

The machine learning literature is endowed with a methodology called *boosting* that converts rough and moderately inaccurate classifiers into stronger combined classifiers by systematically reducing bias. We subjected three of the four classifiers that had been trained to Adaptive Boosting (AdaBoost), a *machine*

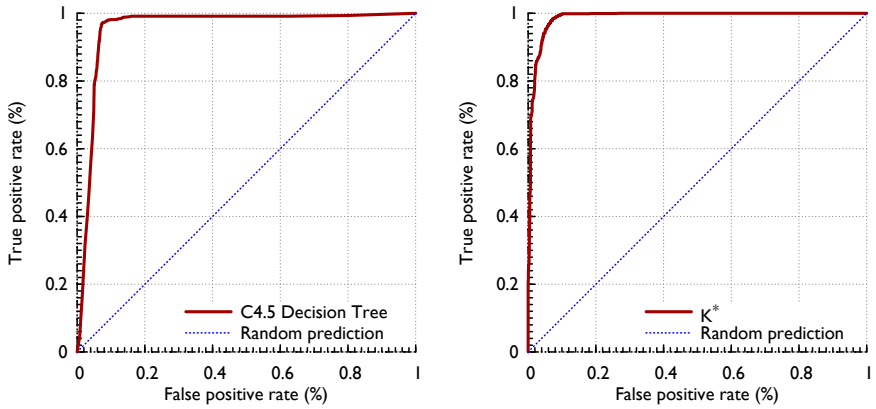

Fig. 8. The Receiver Operating Characteristic (ROC) Curve of our C4.5 (left) and K^* (right) classifiers, respectively.

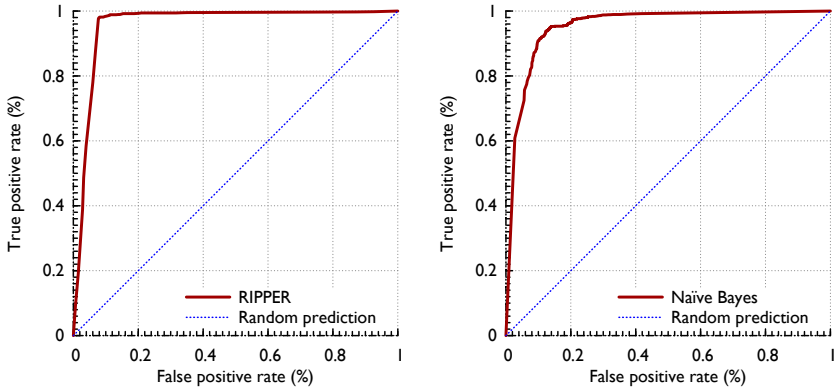

Fig. 9. The Receiver Operating Characteristic (ROC) Curve of our RIPPER (left) and Naïve Bayes (right) classifiers, respectively.

learning meta-algorithm developed by Yoav Freund and Robert Schapire. AdaBoost uses a boosting approach in which multiple classifiers (possibly through parameterization) are trained, and their output is joined into a weighted sum that can provide more accurate prediction [37].

We boosted the previously evaluated machine learning algorithms with AdaBoost, and show the results in Table 2, and Figures 10 and 11.

In these experiments, we note that Naïve Bayes had the largest relative improvement in detection rate and false positives rate through boosting. The best results were obtained from the boosted version of the C4.5, with which we achieve a detection rate of 94.21% and a false positive rate of 3.93%. The improved de-

	TP	TN	FP	FN	TPR	FPR	ACC	ER
C4.5	1366	1443	57	84	94.21 %	3.93 %	95.22 %	4.78 %
RIPPER	1353	1442	58	97	93.31 %	4 %	94.75 %	5.25 %
Naïve Bayes	1345	1362	138	105	92.76 %	9.52 %	91.76 %	8.24 %

Table 2. Experimental results using AdaBoost in conjunction with the previous machine learning algorithms (i.e. C4.5, RIPPER and Naïve Bayes).

tection rate over the best non-boosted method thus comes at the price of higher false positive rate.

We note that False Positive Rate (FPR) must be minimized since malware detection engines are primarily engineered to filter out potentially harmful content. The FPR is low but not zero, and may potentially be reduced through the use of a whitelisting mechanism.

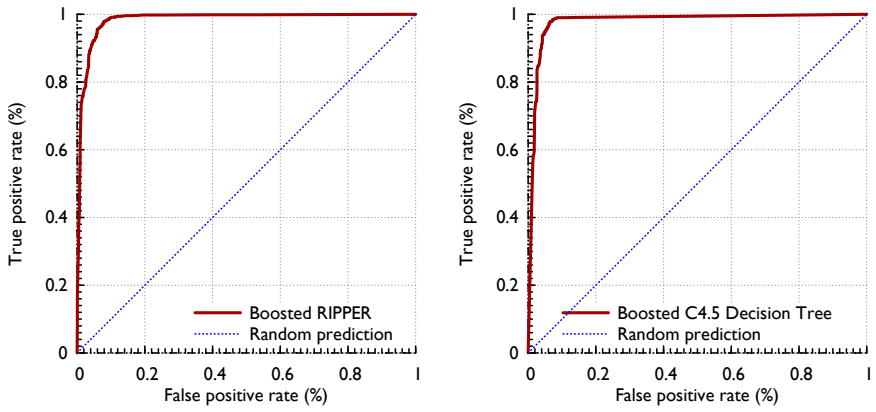

Fig. 10. The Receiver Operating Characteristic (ROC Curve) of our AdaBoost classifier using RIPPER (left) and C4.5 (right) as base classifiers respectively.

4.5 Improvements

Future work will address areas where improvement is needed. First, some permissions declared by Android applications imply different privileges; this delegation of privilege is not captured by the PMDS prototype.

For example, the *READ_EXTERNAL_STORAGE* permission, which allows an application to read from the External Storage, is implicitly granted by the system if the targeted API level is equal or lower than 3 or if the application requires the *WRITE_EXTERNAL_STORAGE* permission, which allows an application to write to the External Storage. Furthermore, the *READ_EXTERNAL_STORAGE*

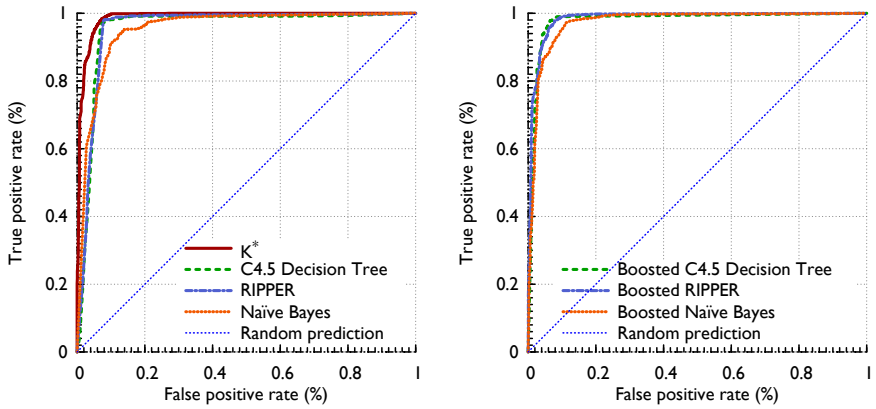

Fig. 11. Comparison of the Receiver Operating Characteristic (ROC Curve) across various base classifiers (left) and the AdaBoost variants (right).

permission is enforced only from API level 19, from which point this permission is not longer required to read files in the application-specific directories.

Another example is the *READ_CALL_LOG* permission, which allows an application to read the user's call log. If an application targets API level equals or lower than 15, this particular permission is implicitly granted by the system if the application requires also the *READ_CONTACTS* permission, which allows an application to read the user's contacts data. PMDS needs to be carefully designed to capture the nuances of privilege delegation.

It is possible that an application uses some permissions – and accordingly to performs some system actions – without explicitly declaring them in its *AndroidManifest.xml* file. Our PMDS prototype ignores this dilemma, but future work should account for implied permissions in order to provide more precise correlation between the group of permissions required by applications and their behavior.

5 Related Work

While many traditional malware detection methodologies are based on signatures, there is a growing trend towards apply machine learning and data mining techniques to detect unknown malicious code. The approaches taken thus far, however, have been mostly concentrated on malware for the Microsoft Windows platform [23–30].

Some prior works explore the possibility of detecting Android malware using permissions. These projects, however, are predominantly based on heuristics and do not deploy machine learning techniques. In the closest project, Huang *et al.* [16] ask the same research question as us: can malicious applications be detected using permissions? In order to retrieve the permissions, the authors

disassemble the APK packages, identify the Android system functions invoked, and then reconstruct the permissions being used. To evaluate their detection model, the authors use a dataset of 124,769 benign applications and 480 malicious ones, and 4 machine learning algorithms, respectively: *AdaBoost*, *Naïve Bayes*, *Decision Tree* and *Support Vector Machine*. The authors use several other features (*e.g.* the number of particular file formats and both the number of under-privileged and over-privileged permissions) in addition to the permissions to improve their detection mechanism. The authors claim that their experiments show that a single classifier is able to detect about 81% of malicious applications. Our evaluation of PMDS suggest that the method achieves comparatively higher detection rate.

Other related works that take advantage of permissions are VetDroid [22], the fingerprinting schemes DroidRanger [17], DroidMOSS [20], and work on generative models for risk scores based on requested permissions of Sarma *et al.* [21].

VetDroid [22] is a dynamic behavioral profiler framework which use the permissions to reconstruct sensitive behaviors in Android applications. VetDroid is able to reconstruct some malicious behaviors of Android applications, to ease malware analysis, and to find information leaks and vulnerabilities. The authors use a dataset of 600 malware, collected from the Android Malware Genome Project [14], and 1,249 free applications collected from the Google Play Store. While both VetDroid and PMDS try to correlate permissions to behavior, VetDroid is a tool to provide better behavior understanding and to help analysis (the output of the system is a report), rather than a complete detection system.

DroidRanger is a permission-based behavioral fingerprinting scheme to detect new samples of known Android malware families [17]. The authors propose a two layer scheme. Applications are first filtered based on the Android permissions required and then sieved through a heuristics-based filter. The authors use a dataset of 182,823 applications collected from 5 different marketplaces to evaluate their detection model. They claim that their experiments show that DroidRanger detects 119 infected applications in their dataset, with a false negative rate of 23.52% in the first version and of 5.04% in the second version. The authors focus only on the most used Android permissions of 10 Android malware families whereas PMDS investigates the whole vector. The works also differ in that PMDS leverages machine learning algorithms for its detection whereas DroidRanger takes a heuristic-based approach. A subset of the authors of DroidRanger also created DroidMOSS [20], which is focused entirely on detecting repackaged applications and uses fuzzy hashing to detect the changes made from the original legitimate program.

Sarma *et al.* propose the use of probabilistic generative models to compute a risk score depending on the permissions required by an application [21]. The authors use a dataset of 158,062 benign applications collected from the Android Market and 121 malicious ones. The authors claim that, for a very low warning rate of 0.05%, they were able to identify the 50% of the malware. Furthermore, using a different weight when training, they claim to achieve 71% of detection rate and 2.4% of warning rate. PMDS, in contrast, achieved 94% detection rate

in our evaluation – comprised of nearly 10× more malicious applications and 10× fewer benign ones – while retaining a lower warning rate. We believe the two approaches are also complementary and could potentially be fused to provide an even more effective detection system.

6 Conclusion

In the wake of the exponential growth of the Android mobile platform there is rapid proliferation of Android malware. In this paper we have proposed a new detection technique for Android malware, called Permission-based Malware Detection System (PMDS). Our work focuses exclusively on how well permissions in Android are indicative of undesirable behavior. We built a client-server architecture for PMDS, the crux of which is a server-side machine learning classifier that automatically identifies (potentially) dangerous behaviors of previously unseen applications based on the combination of permissions they require.

Our experimental results shows the feasibility of using well-established classifiers in order to provide heuristic detection on unknown and zero-day or next-generation malware which are not detected by standard detection systems. PMDS was able to detect more than 92–94% of previously unseen malware with a false positive rate of 1.52–3.93%. Our approach shows that a simple permission-based mechanism may be used alongside classic detection algorithms to thwart the rampant mobile malware and thus raise the security bar in the mobile space.

Acknowledgments

We thank Gianfranco Tonello of VirIT at TG Soft for support, Marjan Siriani for helpful comments and the anonymous reviewers for constructive feedback. Our work was supported in part by grants from Emory University, and grant-of-excellence #120032011 from the Icelandic Research Fund.

References

1. The International Telecommunication Union. The World in 2014: ICT Facts and Figures. `http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2014-e.pdf` (2014)
2. Gartner Forecast: PCs, Ultramobiles, and Mobile Phones, Worldwide, 2011-2018, 2Q 2014. `http://www.gartner.com/document/2780117` (2014)
3. Svajcer, Vanja: Sophos Mobile Security Threat Report (2014).
4. Panda Security: Annual Report PandaLabs 2013. `http://press.pandasecurity.com/wp-content/uploads/2010/05/Quarterly-Report-PandaLabs-April-June-2013.pdf` (2013).

5. F-Secure: F-Secure Mobile Threat Report Q3 2013. http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q3_2013.pdf (2013).

6. G Data SecurityLabs: G Data Mobile Malware Report H2 2013. https://blog.gdatasoftware.com/uploads/media/GData_MobileMWR_H2_2013_EN.pdf (2013).

7. Strategy Analytics: Global Smartphone Installed Base by Operating System for 88 Countries: 2007 to 2017 (2012). http://www.strategyanalytics.com/default.aspx?mod=reportabstractviewer&a0=7834

8. IDC: More Smartphones Were Shipped in Q1 2013 Than Feature Phones, An Industry First According to IDC (2013). http://www.idc.com/getdoc.jsp?containerId=prUS24085413

9. Leavitt, Neal: Malicious code moves to mobile devices. IEEE Computer, vol. 33, num. 12, pp. 16–19. IEEE (2000).

10. Foley, Simon N, Dumigan, Robert: Are handheld viruses a significant threat? Communications of the ACM, vol. 44, num. 1, pp. 105–107. ACM (2001).

11. Dagon, David, Martin, Tom, Starner, Thad: Mobile Phones as Computing Devices: The Viruses are Coming! Pervasive Computing, IEEE, vol. 3, num. 4, pp. 11–15. IEEE (2004).

12. Hypponen, Mikko: State of cell phone malware in 2007. USENIX (2007). http://www.usenix.org/events/sec07/tech/hypponen.pdf

13. Lawton, George: Is it finally time to worry about mobile malware? Computer, vol. 41, num. 5, pp. 12–14. IEEE (2008).

14. Zhou, Yajin, Jiang, Xuxian: Dissecting Android Malware: Characterization and Evolution. IEEE Symposium on Security and Privacy (SP), pp. 95–109. IEEE (2012). http://www.malgenomeproject.org

15. Spreitzenbarth, Michael, Freiling, Felix: Android Malware on the Rise. University of Erlangen, Germany, Tech. Rep. CS-2012-04 (2012).

16. Huang, Chun-Ying, Tsai, Yi-Ting, Hsu, Chung-Han: Performance Evaluation on Permission-Based Detection for Android Malware. Advances in Intelligent Systems and Applications-Volume 2, pp. 111–120. Springer (2013).

17. Zhou, Yajin, Wang, Zhi, Zhou, Wu, Jiang, Xuxian: Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. Proceedings of the 19th Annual Network and Distributed System Security Symposium (2012).

18. Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 175–186, New York, NY, USA, 2014. ACM.

19. Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on Android markets. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 37–54. Springer Berlin Heidelberg, 2012.

20. Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

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

22. Zhang, Yuan, Yang, Min, Xu, Bingquan, Yang, Zhemin, Gu, Guofei, Ning, Peng, Wang, X Sean, Zang, Binyu: Vetting undesirable behaviors in android apps with permission use analysis. Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM (2013).

23. Siddiqui, Muazzam, Wang, Morgan C, Lee, Joohan: A Survey of Data Mining Techniques for Malware Detection using File Features. Proceedings of the 46th Annual Southeast Regional Conference on XX, pp. 509–510. ACM (2008).

24. Ye, Yanfang, Wang, Dingding, Li, Tao, Ye, Dongyi: IMDS: Intelligent Malware Detection System. Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 1043–1047. ACM (2007).

25. Schultz, Matthew G, Eskin, Eleazar, Zadok, F, Stolfo, Salvatore J: Data Mining Methods for Detection of New Malicious Executables. IEEE Symposium on Security and Privacy (SP), pp. 38–49. IEEE (2001).

26. Kolter, J Zico, Maloof, Marcus A: Learning to Detect and Classify Malicious Executables in the Wild. The Journal of Machine Learning Research, vol. 7, pp. 2721–2744. JMLR. org (2006).

27. Tabish, S Momina, Shafiq, M Zubair, Farooq, Muddassar: Malware Detection using Statical Analysis of Byte-Level File Content. Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics, pp. 23–31. ACM (2009).

28. Kiem, H, Thuy, NT, Quang, TMN: A Machine Learning Approach to Anti-virus System. Proceedings of Joint Workshop of Vietnamese Society of AI, SIGKBS-JSAI, ICS-IPSJ and IEICE-SIGAI on Active Mining, Hanoi-Vietnam, pp. 61–65 (2004).

29. Firdausi, Ivan, Lim, Charles, Erwin, Alva, Nugroho, Anto Satriyo: Analysis of machine learning techniques used in behavior-based malware detection. Second International Conference on Advances in Computing, Control and Telecommunication Technologies (ACT), pp. 201–203. IEEE (2010).

30. Dua, Sumeet, Du, Xian: Data mining and machine learning in cybersecurity. Taylor & Francis (2011).

31. Cohen, William W: Fast effective rule induction. ICML, vol. 95, pp. 115–123. ICML (1995).

32. Quinlan, John Ross: C4.5: programs for machine learning. Morgan Kaufmann, vol. 1 (1993).

33. Holmes, Geoffrey, Donkin, Andrew, Witten, Ian H: Weka: A machine learning workbench. Proceedings of the Second Australian and New Zealand Conference on Intelligent Information Systems, pp. 357–361. IEEE (1994).

34. Witten, Ian H, Frank, Eibe: Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann (2005).

35. Cleary, John G, Trigg, Leonard E: K^*: An Instance-based Learner Using an Entropic Distance Measure. ICML, pp. 108–114. ICML (1995).

36. John, George H, Langley, Pat: Estimating continuous distributions in Bayesian classifiers. Proceedings of the Eleventh conference on Uncertainty in artificial intelligence, pp. 338–345. Morgan Kaufmann (1995).

37. Freund, Yoav, Schapire, Robert E: A desicion-theoretic generalization of online learning and an application to boosting. Computational learning theory, pp. 23–37. Springer (1995).

38. The Android Open Source Project: Application Fundamentals. `http://developer.android.com/guide/components/fundamentals.html`

39. The Android Open Source Project: System Permissions. `http://developer.android.com/guide/topics/security/permissions.html`

40. The Android Open Source Project: App Manifest. `http://developer.android.com/guide/topics/manifest/manifest-intro.html`

41. The Android Open Source Project: Android Permissions. `http://developer.android.com/guide/topics/security/permissions.html`

42. The Android Open Source Project: PackageManager. `http://developer.android.com/reference/android/content/pm/PackageManager.html`

43. The University of Waikato: Attribute-Relation File Format (ARFF). `http://www.cs.waikato.ac.nz/ml/weka/arff.html`

44. The University of Waikato: ARFF. `http://weka.wikispaces.com/ARFF`

45. Mila: Contagio Mobile. `http://contagiominidump.blogspot.it`

46. Google: Google Play Store. `https://play.google.com/store`