MAMA: Manifest Analysis for Malware Detection in Android

Borja Sanz[1], Igor Santos[1], Carlos Laorden[1], Xabier Ugarte-Pedrero[1], Javier Nieves[1], Pablo G.Bringas[1], and Gonzalo Álvarez[2]

[1]S3Lab, University of Deusto
Avenida de las Universidades 24, 48007 Bilbao, Spain
{borja.sanz, isantos, claorden, xabier.ugarte,
jnieves, pablo.garcia.bringas@deusto.es

[2]Instituto de Física Aplicada,
Consejo Superior de Investigaciones Científicas (CSIC)
Madrid, Spain
gonzalo@iec.csic.es

Abstract. The use of mobile phones has increased in our lives because they offer nearly the same functionality as a personal computer. Besides, the number of applications available for Android-based mobile devices has increased. Google offers programmers the opportunity to upload and sell applications in the Android Market, but malware writers upload their malicious code there. In light of this background, we present here *Manifest Analysis for Malware detection in Android (MAMA)*, a new method that extracts several features from the Android Manifest of the applications to build machine-learning classifiers and detect malware.

1 Introduction

Smartphones have become very popular. They allow us to check the email, to browse the Internet, or to play games with our friends, wherever we are. But, in order to take advantage of every possibility these devices may offer, applications have to be previously installed in the devices.

In the past, the installation of applications was a source of problems for the users because there was not a centralized site to download their applications and they used to search them in the Internet. Several operating systems like Symbian, in an attempt to avoid piracy and protect the device, used an authentication protocol that certified the application and, usually, caused several inconveniences to the users (e.g., they could not install applications although they bought them).

Nowadays, new methods for distribution and installation have been developed thanks to the widely used Internet connection present in the mobile devices. Therefore, users can install any application they want, avoiding the connection of the device to a personal computer. The App Store of Apple was the first online store to bring this new paradigm to novel users. The model was praised and it became very successful, leading to other vendors such as RIM, Microsoft or Google to adopt the same business model and developing application stores for their devices. These factors have led a large number of developers to focus on these platforms.

However, malware has also arrived to the application markets. To this end, Android and iOS have different approaches to deal with malicious software. According to their response to the US Federal Communication Commission's July 2009[1], Apple applies a very strict review process by at least two reviewers. Android, on the other hand, relies on its security permission system and on the user's sound judgment. Unfortunately, users may not have security consciousness and there is a risk that they may not read the required permissions before installing an application.

Both the AppStore and Android Market include, in their terms of service, clauses that do not allow developers to upload malware to their markets, although they have both hosted malware. Therefore, we can conclude that these models by themselves are insufficient to ensure safety, and that new malware detection methods should be developed in order to enhance the security of the devices.

Machine learning classification has been widely used in malware detection (Schultz et al. 2001), (Devesa et al. 2010), (Santos, Nieves, and Bringas 2011), (Santos, Laorden, and Bringas 2011), (Santos et al. 2011). Several approaches (Rieck et al. 2008), (Tian et al. 2009) have been presented that focus on classifying executables specifying the malware category; e.g., Trojan horses, worms, viruses; or even the malware family of a certain sample.

Regarding Android, the number of new malware samples is also increasing exponentially and several approaches have already been proposed to detect malware. Shabtai, Fledel, and Elovici 2010 built several machine learning models using the following features: the count of elements, attributes and namespaces of the parsed Android Package File (`.apk`). To validate their models, they selected features using three selection methods: Information Gain, Fisher Score and Chi-Square. Their approach achieved 89% of accuracy classifying applications into only 2 categories: tools or games. Albeit this research was not focused in malware detection, their authors suggest the usage of this technique to detect malware as further work.

Besides, other proposals use dynamic analysis for the detection of malicious applications. Crowdroid (Burguera, Zurutuza, and Nadjm-Tehrani 2011) is an approach that analyses the behaviour of the applications. Blasing et al. 2010 created AASandbox, which is a hybrid dynamic-static approximation. The dynamic part is based on the analysis of the logs for the low-level interactions obtained during execution. A. Shabtai and Elovici 2010 also proposed a Host-Based Intrusion Detection System (HIDS) which uses machine learning methods that determines whether the application is malware or not.

Google has also deployed a framework for the supervision of applications called *Bouncer*. Oberheide and Miller 2012 revealed how the system works: it is based in QEMU and performs both static and dynamic analysis.

[1] http://online.wsj.com/public/resources/documents/wsj-2009-0731-FCCApple.pdf

Regarding the state of Android Malware, Zhou and Jiang 2012 conducted a throughout study on this subject, as well as its evolution. They concluded that Android malware shows a rapid increase in both the sophistication and number of new samples. Besides, they show that more than 80% of the malware samples are packed inside legitimate apps, and 93% of them exhibit botnet-like capability.

In light of this background, we present MAMA (Manifest Analysis for Malware detection in Android), a new technique for the detection of malicious Android executables. This approach employs several features extracted by analysing the Manifest file of the Android applications. In particular, we use the permissions and the feature *tags* within the manifest file. These features are then used to build well-known supervised machine-learning algorithms to detect malicious applications.

In summary, our main findings are:

- A new method for representing Android applications, based on the permissions and the features from the Manifest file.
- Adoption of well-known machine learning classifiers to provide detection of malicious applications in Android.
- We found out that machine-learning algorithms can provide detection of malicious applications in Android and that the best representation of executables is the combination of both permissions and features from the Manifest file.

The reminder of this paper is organized as follows. Section 2 presents and details MAMA, our new approach to detect malware in Android. Section 3 describes the empirical evaluation of our method. Finally, section 4 discusses the obtained results and outlines the avenues of further work in this area.

2 Materials and Methods

In this section, we describe our method for the detection of Android malware applications and the dataset used for the validation.

2.1 Dataset Description

In this subsection, we detail how the dataset has been composed. The requirements that the final dataset has to meet are the following:

- **It must be heterogeneous.** It should show diversity in the types of applications available in the Android market.
- **It must be proportional** to the number of samples that already exist of each type of application.

To this end, two different datasets were created. The first one is composed of malicious software, whilst the second one is formed by benign applications.

2.1.1 Malicious Software

To compile the malware dataset, the samples were obtained from the company VirusTotal[2]. VirusTotal offers a series of services called *VirusTotal Malware Intelligence Services*, which allow the researchers to obtain samples from their databases.

To generate the dataset, we first selected the samples. Initially, we collected 2,808 samples. Next, we normalized the values given by the different antivirus vendors. The goal of this step was to determine their reliability detecting malware in Android. To this end, we assumed that every sample that was detected as malware by at least one antivirus was, indeed, malware. Then, we evaluated the detection rate of each antivirus engine with respect to the complete malware dataset:

$$a_i = \frac{n_i}{m} \tag{1}$$

where n_i is the number of samples detected by the i-th antivirus and m is the total number of malware samples. Then, we evaluate each malware sample taking into account the weights of each antivirus.

For this evaluation, we apply the next metric:

$$m = \sum_{i=1}^{i \le \ell} \begin{cases} a_i & \mathit{if\ the\ i^{th}\ antivirus\ catches\ the\ sample} \\ 0 & \mathit{otherwise} \end{cases} \tag{2}$$

where ℓ is the total number of antivirus. Therefore, m rates the detection taking into account the antiviruses that detect the sample.

We determined a threshold below which a sample cannot enter the dataset, in order to ensure that the samples that belong to it are relevant enough. The threshold was set empirically to 0.1, which provided us a total number of 1,202 malware samples.

Besides, we focused on the results given by the different antiviruses to determine whether the samples were actually Android-based applications. This was performed by using the naming convention of the different antivirus engines. Finally, we also removed every duplicated sample.

We finally acquired a malware dataset composed of 333 unique samples. According to the report elaborated by LookOut[3], this dataset represents the 75% of the malware that existed in July, 2011.

2.1.2 Benign Software

To generate this dataset, we gathered 1,811 Android samples of diverse types. To classify them adequately, we categorized them using the same scheme that Android market follows. To this extent, we categorized the applications by means of an unofficial library called *android-market-api*[4]. Once the samples were categorized, we se-

[2] http://www.virustotal.com

[3] https://www.mylookout.com/_downloads/lookout-mobile-threat-report-2011.pdf

[4] http://code.google.com/p/android-market-api/

lected a subgroup of samples to be part of the final benign software dataset. The methodology employed was the following:

1. *Determine the number of total samples.* To facilitate the training of the machine-learning models, it is usually desirable for both categories to be balanced. Therefore, given that the number of malware samples is inferior to the benign ones, we reduced the number of benign applications to 333.
2. *Determine the number of samples for each benign category.* Second, we decided to balance the dataset according to the distribution in the Android market, and therefore, selected the number of applications consequently.
3. *Types of application.* There are different types of applications: native ones (developed by means of the Android SDK), web (developed through HTML, JavaScript and CSS) and widgets (simple applications displayed in the Android desktop). All these applications have different features. To generate the dataset, we made no distinction in the type of application and included samples of the different types in the final dataset.
4. *Selection of the samples for each category.* Once the number of applications for each category was determined, we randomly selected the applications using a Monte Carlo sampling method, avoiding different versions of the same application.

Following this methodology, we constructed the benign dataset. The number of samples for each category is shown in Table 1.

Table 1. Number of samples for each category.

Arcade and Action	32	Multimedia & Video	23
Books	10	Music & Audio	12
Business	1	News & magazines	7
Card Games	2	Personalization	6
Casuals	10	Photography	6
Comics	1	Productivity	27
Communication	20	Puzzles	16
Education	0	Races	2
Enterprise	4	Sales	3
Entertainment	16	Society	25
Finance	3	Sports	5
Health	3	Tools	80
Libraries & Demos	2	Transportation	2
Lifestyle	4	Travels	8
Medicine	1	Weather	2
TOTAL: 333 applications			

2.2 Feature Engineering

In this section, we review the different feature sets that we have used in order to detect Android malware. We have gathered these features from the AndroidManifest.xml file that is within each Android application. The AndroidManifest.xml file has the structure shown in Fig. 1.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
    <compatible-screens />
    <supports-gl-texture />
    <application>
        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>
        <activity-alias>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </activity-alias>
        <service>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </service>
        <receiver>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </receiver>
        <provider>
            <grant-uri-permission />
            <meta-data />
            <path-permission />
        </provider>
        <uses-library />
    </application>
</manifest>
```

Fig. 1. Manifest file structure

These feature sets are: (i) the permissions required for the application, under the *uses-permission* tag and (ii) the features under the *uses-features* group in the Android Manifest File.

In order to obtain these features, we first extracted the permissions used by each of the applications. To this extent, we employed the *aapt* tool (Android Asset Packaging Tool), available within the set of tools provided by the Android SDK.

These features were selected for two main reasons: first, the process of gathering them has a low computing overhead and, second, the different behaviours that may be present within the application are manifested by them. In order to compare the feature sets, we have evaluated both feature sets separately and, then, combined in a unique feature vector.

2.2.1 Permissions of the Manifest File

In this section, we analyse the permissions requested by the applications in the dataset in order to measure their relevance in the malware detection process.

The structure for declaring a *uses-permission*[5] in the *AndroidManifest.xml* file is shown in Fig. 2.

```
<uses-permission android:name="string" />
```

Fig. 2. General template for a *uses-permission* within the AndroidManifest file.

In this way, there are several strings that are used for declaring the permission usage of the different Android applications such as *"android.permission.CAMERA"* or *"android.permission.SEND_SMS".*

We have analysed the number of permissions and their frequency in a previous step for knowing their distribution within our dataset. Fig. 3 shows that *INTERNET* is one of the most required permissions in both categories. However, we can notice that malware applications present a more intensive use of the permissions related with the sending and reception of text messages.

We can also realize that the differences between the number of malicious applications that require the *INSTALL_PACKAGES* permission and the number of benign applications that do it. This permission, combined with the Internet connection of the device may allow the attacker to install any type of application. Other kind of permissions, such as the ones designated to gather users' data, have also a high relevance in the malicious applications.

[5] http://developer.android.com/guide/topics/manifest/uses-permission-element.html

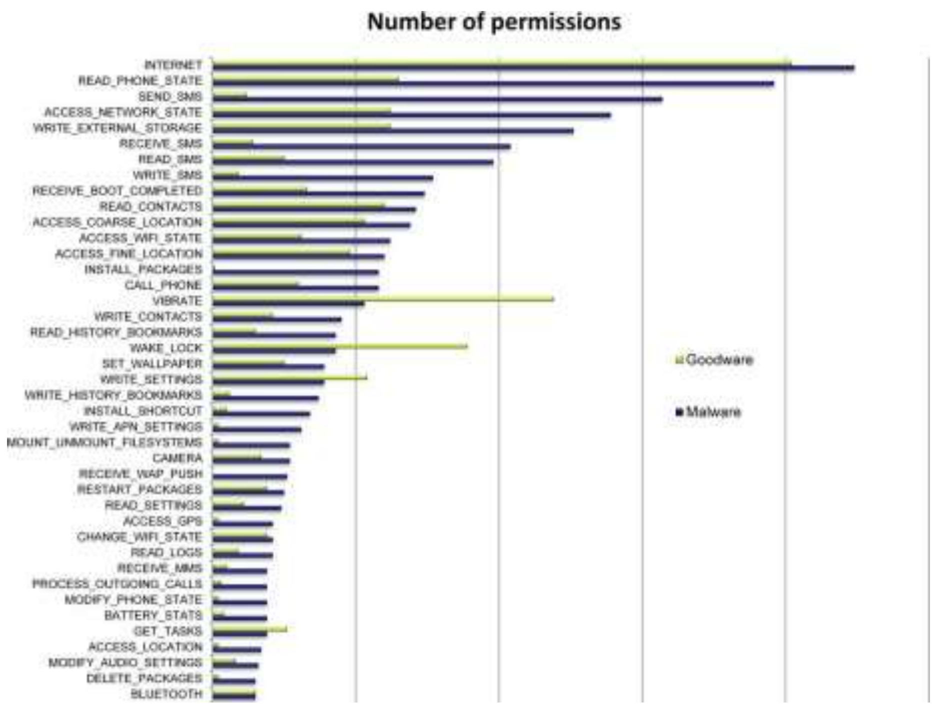

Fig. 3. Most used extracted permissions from the applications.

On the other hand, we have also analysed the number of permissions of each application.

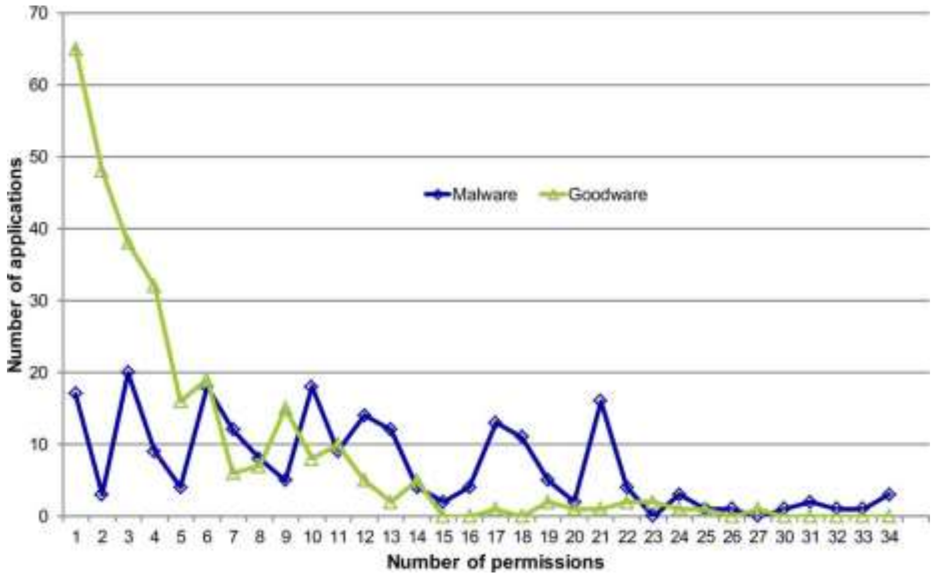

Fig. **4** shows that the number of permissions required by each of the categories (malware and benign) is similar. It can also be noticed that the number of benign applications that require 2 or 3 permissions is higher than the number of the malicious samples requiring that number of permissions. Besides, there is an important number of benign applications that only requires a single permission in order to run. With these data in mind, we can conclude that the number of permissions by itself does not seem relevant to determine whether an application is malware or not.

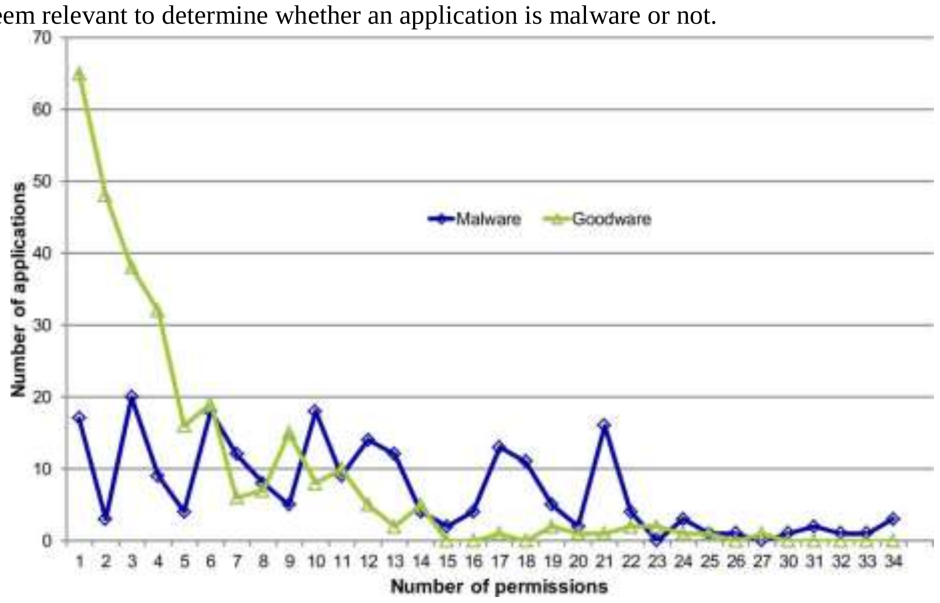

Fig. 4. Number of permissions required by the analysed applications.

In order to use them as input for machine-learning algorithms, we processed the *AndroidManifest.xml* file, searching for the *uses-permission* tag, and retrieved the string declaring the type of permission. After that, we generated an input vector for each of the 130 possible permissions[6] and a binary feature indicating whether the permission is present or not in its analysed Android application.

For example, Fig. 5 shows the permission declaration of an Android application.

```
…
<uses-permission android:name=" an-
droid.permission.SEND_SMS " />
<uses-permission android:name=" an-
droid.permission.INTERNET " />
<uses-permission android:name=" an-
droid.permission.READ_CONTACTS" />
…
```

Fig. 5. Example of permission declaration in an application.

[6] http://developer.android.com/reference/android/Manifest.permission.html

The input vector of permissions for this application would be composed of 127 zeros, representing the permissions not used and ones in the 3 used permissions (SEND_SMS, INTERNET, and READ_CONTACTS).

2.2.2 Other features of the Manifest File: *uses-feature* tag

Within the *AndroidManifest.xml* file, there are other features apart from the permissions. This information may be relevant for the task of detecting malware.

The structure for declaring a *uses-feature*[7] in the *AndroidManifest.xml* file is shown in Fig. 6.

```
<uses-feature
  android:name="string"
  android:required=["true" | "false"]
  android:glEsVersion="integer" />
```

Fig. 6. Declaration of features in an application.

The *android:name* determines the element used by the application, the *android:required* determines whether that feature is mandatory for the correct handling of the application or not, and *glEsVersion* determines the version of OpenGL, if used. These features are represented within the Manifest file with the tag *<uses-feature>* and determine some of the features, both software and hardware, that are mandatory for the execution. In this way, the use of Bluetooth or the camera are determined by the tags *android.hardware.bluetooth* and *android.hardware.camera*.

Besides, these elements of the Manifest file only inform about the behaviour of the application and are not even mandatory. Even though this information is not mandatory, it is used sometimes by other services or applications in order to improve the interaction between the applications. Nevertheless, the optional character of these features makes a high number of applications to lack these fields.

In our dataset, the features extracted are related to the use of different hardware systems such as localization usage by means of the GPS, Wi-Fi, or proximity and light sensors.

In light of this context, we have considered this information relevant in order to determine whether an application is malware or not, because it adds some information complementary to permissions and provides us with a behavioural view of the inspected application.

In order to use these features as input vector for machine-learning, we processed the *AndroidManifest.xml* file searching for the *uses-features* tag and gathered the string declaring the type of feature. After that, we generated an input vector for each of the 37 (34 of hardware and 3 from software) possible features and a binary feature indicating whether the feature is present or not in the analysed Android application.

As an example (Fig. 7), shows the declaration of *uses-features* for an Android application.

[7] http://developer.android.com/guide/topics/manifest/uses-feature-element.html

```
…
<uses-feature an-
droid:name="android.hardware.camera" an-
droid:required="false" />
<uses-feature an-
droid:name="android.hardware.bluetooth" />
…
```

Fig. 7. Example of features used in an application.

The input vector of features for this application will be composed of 35 zeros, representing the not used uses features and ones in the 2 used features ("*android.hardware.camera*" and "*android.hardware.bluetooth*").

2.3 Machine learning algorithms

Machine learning is an area within Artificial Intelligence that develops and designs new algorithms to generalize behaviours using data (Bishop 2006). Traditionally, these algorithms are categorized with regards to the availability of labelled instances in the training dataset. We have used supervised algorithms, which are the ones that employ a dataset that has been previously labelled (in our case, into malware and benign software). In this section we detail the algorithms employed.

2.3.1 K-Nearest Neighbours

The K-Nearest Neighbour (KNN) (Fix and Hodges Jr 1952) classifier is one of the simplest supervised machine learning models. This method classifies an unknown specimen based on the class of the instances closest to it in the training space by measuring the distance between the training instances and the unknown instance.

Even though several methods exist in order to choose the class of the unknown sample, the most common technique is to simply classify the unknown instance as the most common class amongst the K-nearest neighbours.

2.3.2 Decision Trees

Decision Tree classifiers are a type of machine-learning classifiers that are graphically represented as trees. Internal nodes denote conditions regarding the variables of a problem, whereas final nodes or leaves represent the ultimate decision of the algorithm (Quinlan 1986).

Different training methods are typically used for learning the graph structure of these models from a labelled dataset. We used Random Forest, an ensemble (i.e., combination of classifiers) of different randomly-built decision trees (Breiman 2001), and J48, the WEKA (Garner 1995) implementation of the C4.5 algorithm (Quinlan 1993).

2.3.3 Bayesian networks

Bayesian Networks (Pearl 1982), which are based on the Bayes Theorem, are defined as graphical probabilistic models for multivariate analysis. Specifically, they are directed acyclic graphs that have an associated probability distribution function (Castillo, Gutiérrez, and Hadi 1997). Nodes within the directed graph represent problem variables (they can be either a premise or a conclusion) and the edges represent conditional dependencies between such variables. Moreover, the probability function illustrates the strength of these relationships in the graph (Castillo, Gutiérrez, and Hadi 1997).

The most important capability of Bayesian Networks is their ability to determine the probability that a certain hypothesis is true (e.g., the probability of an application of being malware).

2.3.4 Support Vector Machines (SVM)

SVM algorithms divide the n-dimensional space representation of the data into two regions using a hyperplane. This hyperplane always maximizes the margin between those two regions or classes. The margin is defined by the farthest distance between the examples of the two classes and computed based on the distance between the closest instances of both classes, which are called supporting vectors (Vapnik 2000).

Instead of using linear hyperplanes, it is common to use the so-called kernel functions. These kernel functions lead to non-linear classification surfaces, such as polynomial, radial or sigmoid surfaces (Amari and Wu 1999).

3 Results

In this section we will analyse the results obtained for each of the methods outlined above. First we describe the evaluation method that has been employed to then present the results.

3.1 Configuration

For the evaluation of the different machine learning algorithms we used the tool WEKA (Waikato Environment for Knowledge Analysis[8]). Specifically, the algorithms used in this tool can be seen in Table 2. In those cases in which no configuration parameters are specified, the configuration used was the default.

The dataset was divided using the *k*-fold cross-validation technique (Kohavi 1995), (Devijver and Kittler 1982). It divides *k* times the input dataset in *k* complementary subsets using one shaping sample data set, called test set, while the rest of subsets forming the joint training. To obtain the error ratio for the final sample, the arithmetic mean of the error rates obtained for each of the *k* iterations is calculated. In our case, the cross validation was configured with 10 folds.

[8] http://www.cs.waikato.ac.nz/ml/weka

Table 2. Configuration of the algorithms

Algorithm	Configuration
Naïve Bayes	N/A
Bayesian Network (BN)	K2 and TAN
Support Vector Machine (SVM)	Polynomial and Normalized Polynomial Kernel
J48	N/A
KNN	K = 1, 3 and 5
RandomForest	N = 10, 50 and 100

3.2 Measures

The evaluation was performed by measuring the following metrics:

- **True Positive Ratio (TPR).**

$$TPR = \frac{TP}{TP + FN} \tag{3}$$

  where *TP* is the number of malware cases correctly classified (true positives) and *FN* is the number of malware cases misclassified as legitimate software (false negatives).
- **False Positive Ratio (FPR).**

$$FPR = \frac{FP}{FP + TN} \tag{4}$$

  where *FP* is the number of benign software cases incorrectly detected as malware and *TN* is the number of legitimate executables correctly classified.
- **Accuracy.** It is the total number of the classifier's hits divided by the number of instances in the whole dataset:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{5}$$

- **Area under the ROC Curve (AUC).** AUC establishes the relation between false negatives and false positives (Singh, Kaur, and Malhotra 2009). The ROC curve is obtained by plotting the TPR against the FPR.

3.3 Results

In this section, we detail the obtained results when only permissions are used, when only features are used and when a combination of both permissions and features are used.

3.3.1 Permissions

The results obtained in the analysis of the permissions are shown in Table 3. We can notice similar results in the different families of algorithms, although those belonging to the family of Bayesian algorithms obtained a significantly worse outcome.

In general, we obtain a detection rate around 85%, while it is true that the AUC values are more varied. We can conclude that the best results are obtained with Random Forest, using a population of 100 trees, achieving an 87% of malware detection accuracy and an AUC of 0.95.

Table 3. Obtained results when using only permissions as training characteristics.

Algorithm	TPR	FPR	AUC	Accuracy (%)
Naïve Bayes	0.87	0.42	0.77	72.78
BN K2	0.90	0.41	0.79	74.49
BN TAN	0.91	0.41	0.85	74.99
SVM: Polynomial Kernel	0.74	0.07	0.83	83.34
SVM: Normalized-Polynomial Kernel	0.79	0.05	0.87	87.14
KNN K =1	0.80	0.06	0.93	87.28
KNN K =3	0.77	0.10	0.93	83.77
KNN K =5	0.76	0.10	0.92	83.14
KNN K =10	0.73	0.20	0.89	76.65
J48	0.72	0.10	0.87	80.90
Random Forest 10	0.80	0.06	0.94	87.07
Random Forest 50	0.79	0.05	0.95	87.28
Random Forest 100	**0.80**	**0.05**	**0.95**	**87.41**

3.3.2 Features

The obtained results in the analysis of the features are reported in Table 4. The accuracy of the models generated with the features is significantly better for this feature set respect to the permission-based set.

In view of these results, we can conclude that the features that are declared in the application are at least as significant as the permissions when determining whether a sample is malicious or not.

However, these features are not enough on their own, because they are not a mandatory attribute and it may be the case that the application does not contain any feature of this type. Therefore, the next step is the combination of both attributes.

Table 4. Obtained results when using only features as training characteristics.

Algorithm	TPR	FPR	AUC	Accuracy (%)
Naïve Bayes	0.81	0.18	0.87	81.38
BN K2	0.84	0.13	0.88	85.30
BN TAN	0.84	0.12	0.88	85.89
SVM: Polynomial Kernel	0.84	0.12	0.86	86.10
SVM: Normalized-Polynomial Kernel	0.83	0.11	0.86	85.66
KNN K =1	0.82	0.11	0.90	85.34
KNN K =3	0.81	0.10	0.91	85.73
KNN K =5	0.82	0.11	0.91	85.45
KNN K =10	0.82	0.12	0.91	85.38
J48	0.82	0.08	0.90	87.09
Random Forest 10	0.82	0.10	0.91	85.99
Random Forest 50	0.82	0.10	0.91	86.06
Random Forest 100	**0.82**	**0.10**	**0.91**	**86.09**

3.3.3 Permissions and Features combined

After obtaining the results of both the permissions and the features, we combined the information provided by both. In this way, we are able to obtain information that, separately, would be possible to infer. The results can be seen in Table 5.

Table 5. Obtained results when using permissions and features as training characteristics.

Algorithm	TPR	FPR	AUC	Accuracy (%)
Naïve Bayes	0.86	0.10	0.91	88.05
BN K2	0.89	0.10	0.93	89.28
BN TAN	0.92	0.10	0.96	91.11
SVM: Polynomial Kernel	0.95	0.07	0.94	94.07
SVM: Normalized Polynomial Kernel	0.95	0.06	0.94	94.50
KNN K =1	0.96	0.06	0.97	95.22
KNN K =3	0.97	0.11	0.98	92.85
KNN K =5	0.96	0.12	0.97	91.91
KNN K =10	0.96	0.14	0.97	90.82
J48	0.91	0.10	0.93	90.60
Random Forest 10	0.94	0.06	0.98	93.96
Random Forest 50	0.95	0.05	0.98	94.79
Random Forest 100	**0.94**	**0.05**	**0.98**	**94.83**

In particular, the best results are obtained with the algorithm Random Forest with 100 trees, with an area under the ROC curve of 0.98 and with an accuracy of 94.83%.

4 Comparison with Related Work

To combat the problem of malware that has risen in recent years in Android, researchers have begun to explore this area, using the experience acquired in other platforms.

We can distinguish two different approaches. The first one attempts to detect anomalies, while the second one is based on signature scanning, that is, detecting known patterns present in the malicious applications.

Shabtai and Elovici 2010 presented “Andromaly”, a framework for detecting malware on Android mobile devices. This framework collected 88 features and events and, then, applied machine-learning algorithms to detect abnormal behaviours. Their dataset was composed of 4 self-written malware, as well as goodware samples, both separated into two different categories (games and tools). Their approach achieved a 0.99 area under ROC curve and 99% of accuracy.

Despite these results, their framework had to collect a huge number of features and events, overloading the device and, consequently, draining the battery. Our approach only needs information extracted from AndroidManifest.xml, which is within `.apk` files, making the extraction process almost trivial. Although our results are not as sound as theirs, our approach requires much less computational effort and our dataset is larger and sparser in malware samples than theirs.

Regarding the signature based approach, Schmidt, Camtepe, and Albayrak 2010 focused on a static and light-weight analysis of the samples. They used the system calls as features and simple classifiers to detect malicious behaviours. They obtained their best result using the KNN algorithm, achieving a 99% of accuracy.

Both approaches do not prevent the installation of malware in devices. Android platform evaluates the permissions of the application at installation time. After that, they are not evaluated anymore. Thus, it is important to categorize the application before the installation. Our approach improves this point, obtaining similar results to those obtained in the anomaly detection.

Using a similar approach to ours, Peng et al. 2012 ranks the risks in Android using probabilistic generative models. They selected the permissions of the applications as key feature. Specifically, they chose the top 20 most frequently requested permissions in their dataset, composed by 2 benign software collections, obtained from the Google Play (157,856 and 324,658 samples, respectively) and 378 unique samples of malware. They obtained a 0.94 area under ROC curve as best result. We complemented the information provided by the permissions with the *uses-features*, enhancing the results and approaching the results to those obtained by previous methods.

In summary, our approach prevents the malware installation on the devices, instead of monitoring the execution of the applications, thus saving device resources.

5 Discussion

Manifest files in Android applications include two important sources of information: *uses-permissions* and *uses-features*. Permissions are the most recognizable security feature in Android. A user has to accept them to install any application. Features, otherwise, are optional fields in the manifest file that provide information about the resources and the behaviour of the application. In this work we evaluated the capacity of these two feature sets to detect malware using machine-learning techniques.

To validate our method, we collected malware and benign samples of Android applications. Then, we extracted the aforementioned features for each application and trained the models, showing that the combination of these two features can provide high accuracy detecting malware. Nevertheless, there are several considerations regarding the viability of our approach.

Our method seeks to prevent the installation of malware on the device, but there are several ways to avoid this protection. For example, an application with permissions to connect to the Internet (which is one of the most required permissions) can download the payload after installation and, then, attack the system. Our approach characterizes the application, but not its behaviour. For this, it would require dynamic techniques such as described above ones.

Forensic experts are developing reverse engineering tools for Android applications. Researchers could use them to retrieve new features to enhance the used information to train the machine-learning models. Besides, despite the high detection rate, the obtained results have a high false positive rate. Therefore, this method can be used as a first step previous to other more extensive analysis, such as a dynamic analysis.

Future work of this Android malware detection tool is oriented in two main directions. First, there are other features from the applications that could be used to improve the detection ratio that do not require executing the sample. Forensics tools for Android applications should be developed in order to obtain new features. Second, dynamic analysis provides additional information that could improve malware detection systems. Unfortunately, computational resources in smartphones are limited and not sufficient for this kind of analysis.

Acknowledgements

This research was partially supported by the Basque Government under the research project “BRANKA4U: Evolución de los servicios bancarios hacia el futuro” granted by the ETORGAI 2011 program.

References

Amari, S., and S. Wu. 1999. “Improving Support Vector Machine Classifiers by Modifying Kernel Functions.” *Neural Networks* 12 (6): 783–789.

Bishop, C.M. 2006. *Pattern Recognition and Machine Learning*. Springer New York.

Blasing, T., L. Batyuk, A.D. Schmidt, S.A. Camtepe, and S. Albayrak. 2010. "An Android Application Sandbox System for Suspicious Software Detection." In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference On*, 55–62.

Breiman, L. 2001. "Random Forests." *Machine Learning* 45 (1): 5–32.

Burguera, I., U. Zurutuza, and S. Nadjm-Tehrani. 2011. "Crowdroid: Behavior-based Malware Detection System for Android." In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 15–26.

Castillo, E., J.M. Gutiérrez, and A.S. Hadi. 1997. *Expert Systems and Probabilistic Network Models*. Springer Verlag.

Devesa, J., I. Santos, X. Cantero, Y. K. Penya, and P. G. Bringas. 2010. "Automatic Behaviour-based Analysis and Classification System for Malware Detection." In *Proceedings of the 12th International Conference on Enterprise Information Systems (ICEIS)*.

Devijver, P.A., and J. Kittler. 1982. *Pattern Recognition: A Statistical Approach*. Prentice/Hall International.

Fix, E., and J.L. Hodges Jr. 1952. *Discriminatory Analysis-nonparametric Discrimination: Small Sample Performance*. DTIC Document.

Garner, S.R. 1995. "Weka: The Waikato Environment for Knowledge Analysis." In *Proceedings of the 1995 New Zealand Computer Science Research Students Conference*, 57–64.

Kohavi, R. 1995. "A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection." In *International Joint Conference on Artificial Intelligence*, 14:1137–1145.

Oberheide, J., and J. Miller. 2012. *Dissecting the Android Bouncer*.

Pearl, Judea. 1982. "Reverend Bayes on Inference Engines: a Distributed Hierarchical Approach." In *Proceedings of the National Conference on Artificial Intelligence*, 133–136.

Peng, H., C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. 2012. "Using Probabilistic Generative Models for Ranking Risks of Android Apps." In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 241–252.

Quinlan, J.R. 1986. "Induction of Decision Trees." *Machine Learning* 1 (1): 81–106.
1993. *C4. 5: Programs for Machine Learning*. Morgan kaufmann.

Rieck, K., T. Holz, C. Willems, P. Düssel, and P. Laskov. 2008. "Learning and Classification of Malware Behavior." *Detection of Intrusions and Malware, and Vulnerability Assessment*: 108–125.

Santos, Igor, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo García Bringas. 2011. "Opcode Sequences as Representation of Executables for Data-mining-based Unknown Malware Detection." *Information Sciences* ? (?): ?–?

Santos, Igor, Carlos Laorden, and Pablo G. Bringas. 2011. "Collective Classification for Unknown Malware Detection." In *Proceedings of the 6th International Conference on Security and Cryptography (SECRYPT)*, 251–256.

Santos, Igor, Javier Nieves, and Pablo G. Bringas. 2011. “Semi-supervised Learning for Unknown Malware Detection.” In *Proceedings of the 4th International Symposium on Distributed Computing and Artificial Intelligence (DCAI). 9th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS)*, 415–422.

Schmidt, A.D., A. Camtepe, and S. Albayrak. 2010. “Static Smartphone Malware Detection.” In *Proceedings of the 5th Security Research Conference (Future Security 2010), ISBN*, 978–3.

Schultz, MG, E. Eskin, F. Zadok, and SJ Stolfo. 2001. “Data Mining Methods for Detection of New Malicious Executables.” In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 38–49.

Shabtai, A., and Y. Elovici. 2010. “Applying Behavioral Detection on Android-based Devices.” *Mobile Wireless Middleware, Operating Systems, and Applications*: 235–249.

Shabtai, Asaf, Yuval Fledel, and Yuval Elovici. 2010. “Automated Static Code Analysis for Classifying Android Applications Using Machine Learning.” *2010 International Conference on Computational Intelligence and Security* (December): 329–333. doi:10.1109/CIS.2010.77.

Singh, Y., A. Kaur, and R. Malhotra. 2009. “Comparative Analysis of Regression and Machine Learning Methods for Predicting Fault Proneness Models.” *International Journal of Computer Applications in Technology* 35 (2): 183–193.

Tian, R., L. Batten, R. Islam, and S. Versteeg. 2009. “An Automated Classification System Based on the Strings of Trojan and Virus Families.” In *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference On*, 23–30.

Vapnik, V.N. 2000. *The Nature of Statistical Learning Theory*. Springer.

Zhou, Y., and X. Jiang. 2012. “Dissecting Android Malware: Characterization and Evolution.” In *Proceedings of the 33$^{rd\$}$ IEEE Symposium on Security and Privacy (S&P 2012)*.