Instance-based Anomaly Method for Android Malware Detection

Borja Sanz, Igor Santos, Xabier Ugarte-Pedrero,Carlos Laorden, Javier Nieves and Pablo G. Bringas
S3Lab, University of Deusto, Avenida de las Universidades 24, Bilbao, Spain
{*borja.sanz, isantos, claorden, xabier.ugarte,jnieves, pablo.garcia.bringas*}*@deusto.es*

Keywords: Security, Malware, Android

Abstract: The usage 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. Android application distribution is based on a centralized market where the developers can upload and sell their applications. However, as it happens with any popular service, it is prone to misuse and, in particular, malware writers can use this market to upload their malicious creations. In this paper, we propose a new method that, based upon several features that are extracted from the AndroidManifest file of the legitimate applications, builds an anomaly detection system able to detect malware.

1 INTRODUCTION

Smartphones have become an indispensable gadget in our daily lives. We check our email, browse the Internet, or play games with our friends, wherever we are. However, in order to take advantage of every possibility they may offer, applications have to be previously installed in the devices. In the past, the installation of applications was uncomfortable for the users because the process was complicated: users had to look for the desired application in the Internet and, after finding it, they had to install it in their devices.

Afterwards, new methods for distribution and installation were developed taking advantage of the Internet connection available in mobile devices. Users can now install any application without a personal computer, by using application stores that are already installed in the devices. Apple's AppStore was the first online store to bring this new paradigm for users. Since then, other vendors such as RIM, Microsoft or Google have adopted the same model and deployed application stores for their devices. These factors have contributed to the popularity of smartphones and thus, the number of applications has increased. In particular, Apple's App Store offers more than 800,000 applications to their users[1] while Google's Play Store, Android's official application store, hosts 675,000 apps[2].

[1]`http://www.apple.com/pr/library/2013/01/28Apple-Updates-iOS-to-6-1.html`

[2]`http://officialandroid.blogspot.com.es/search?q=675000`

Unfortunately, application markets are also susceptible of hosting malware. In order to deal with these threats, Android and iOS use different approaches. While Apple applies a very strict review process for the submitted applications performed by at least two reviewers, Android relies on its security permission system and on the user's sound judgement. However, users may not have security consciousness and may not read the required permissions before installing an application (Mylonas et al., 2012). Despite these efforts, both vendors have hosted malware in their stores (Egele et al., 2011; Zhou and Jiang, 2012). Therefore, both models are not sufficient to ensure user's safety and new models should be developed and deployed in order to improve the security of the devices.

With regards to Android malware, Zhou et al. (Zhou and Jiang, 2012) created a big Android malware collection between 2010 and 2011. They obtained 23 samples in January 2011 and 1,260 in October 2011 (which represented an increase of over 5,000%). They conducted a thorough study on this subject and analysed its evolution in recent times. They concluded that Android malware has shown a rapid increase in both sophistication and number of new samples. Besides, they show that more than 80% of the malware samples repackage legitimate apps and 93% of them exhibit a botnet-like capability.

Several approaches have been proposed to detect these malicious software in Android. Shabtai et al. (Shabtai et al., 2010) trained 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 explicitly 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 et al., 2011) is an approach that analyses the behaviour of the applications through device usage features. Blasing et al. created AASandbox (Blasing et al., 2010), which is a hybrid dynamic-static approximation. The approach is based on the analysis of the logs for the low-level interactions obtained during execution. Shabtai and Elovici (Shabtai and Elovici, 2010) also proposed a Host-Based Intrusion Detection System (HIDS) which used machine learning methods to determine whether an application is malware or not. Google itself has also deployed a framework for the supervision of applications called Bouncer. Oberheide and Miller 2012[3] revealed how the system works: it is based in QEMU and performs both static and dynamic analysis.

Against this background, we present a new technique for the detection of malicious Android executables. This approach uses several features extracted by analysing the Manifest file of Android applications. In particular, we use the `uses-permission` and the `uses-feature` tags within the manifest file. Using these features of several legitimate applications, we build an instance-based anomaly detection method able to detect anomalous malicious applications.

The reminder of this paper is organised as follows. Section 2 details the generation of the dataset. Section 3 presents the permissions used in our approach. Section 4 describes our anomaly detection method. Section 5 shows the results obtained for the empirical evaluation. Section 6 details the related work. Finally, section 7 discusses the results and shows the avenues of further work.

2 DATASET DESCRIPTION

In this section, we describe the dataset collected to validate our method. The dataset is composed of both benign and malicious software. In order to compose the dataset, the following requirements were considered: (i) it must be heterogeneous, showing diversity in the types of applications available in the Android market, and (ii), it must be proportional to the number of samples of each category in the Android market.

2.1 Benign Software Dataset

We gathered 1,811 Android samples of diverse types and categorised them using the same scheme that Android market follows. To this end, we used an unofficial library called android-market-api[4] that retrieves the category of a given application. From the categorised samples, we selected a subsample to be part of the final benign software dataset. The methodology employed was the following:

1. *Determine the number of total samples*. In order to facilitate the training of machine-learning models, it is usually desirable for both categories to be balanced. Otherwise, the resulting model can be biased towards one of the classes, and the results may not be fully representative. Therefore, given that the number of malware samples is inferior to the benign ones, we reduced the number of benign applications to meet the number of the malicious ones.
2. *Determine the number of samples for each benign category*. We decided to balance the dataset according to the observed 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, which only launch a Webkit frame) and widgets (simple applications displayed in the Android desktop). All these applications have the same core, but different features. To represent the heterogeneity of Android, we included samples of each different type 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, avoiding different versions of the same application. Table 1 shows the distribution of the legitimate applications in the categories.

2.2 Malicious Software Dataset

The malicious samples were obtained thanks to the company VirusTotal[5]. VirusTotal offers a series of

[3]`http://jon.oberheide.org/files/summercon12-bouncer.pdf`

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

[5]`http://www.virustotal.com/`

Table 1: Number of benign software applications.

Category	*Number*	*Category*	*Number*
Action and Arcade	32	Multimedia and video	23
Books	10	Music and audio	12
Business	1	News and magazines	7
Cards Games	2	Personalization	6
Casuals	10	Photography	6
Comics	1	Productivity	27
Communication	20	Puzzles	16
Sports	5	Races	6
Enterprise	4	Sales	3
Entertainment	16	Society	25
Finance	3	Tools	80
Health	3	Transportation	2
Libraries and Demos	2	Travels	2
Lifestyle	4	Weather	8
Medicine	1		
	Total: 333		

services called *VirusTotal Malware Intelligence Services*, which allows researchers to obtain samples from their databases.

Initially, we collected 2,808 samples. Next, we normalized the different outputs of 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:

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

where n_i is the number of samples detected by the i^{th} antivirus and n is the total number of malware samples. Then, we evaluated each malware sample taking into account the weight of each antivirus. For this evaluation, we applied the next metric:

$$w = \sum_{i=1}^{\#\mathcal{A}} a_i \in \mathcal{A} \tag{2}$$

being $\mathcal{A} = (a_1, a_2, ..., a_\ell)$ the set of weights of the ℓ antiviruses that detect that particular sample. Therefore, w rates the detection taking into account the antiviruses that detect the sample. We determined a threshold to discard irrelevant samples from the dataset that was set empirically to 0.1, which provided us a total number of 1,202 malware samples. Finally, we also removed every duplicated sample so the final malware dataset was composed of 333 unique samples.

3 FEATURE ENGINEERING

In this section, we review the different feature sets we have considered for the detection of Android malware. We gathered these features from the `AndroidManifest.xml` file that is packed with every Android application. To this end, we first extracted the permissions used by each application using the Android Asset Packaging Tool (`aapt`), available within the set of tools provided by the Android SDK.

3.1 Permissions of the application

The structure for declaring a `uses-permission`[6] in the `AndroidManifest.xml` file is shown in Figure 1.

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

Figure 1: 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 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

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

possible permissions, with a binary feature indicating whether the permission is present or not in the analysed Android application. For example, Figure 2 shows the permission declaration of an Android application.

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

Figure 2: Example of permission declaration in an application.

The input vector of permissions for this application would be composed of 127 zeros, representing the permissions not used and 1s for the 3 `uses-permission` tags declared (`SEND_SMS`, `INTERNET`, and `READ_CONTACTS`).

We selected this feature set for two main reasons: first, the gathering process has a low computing overhead and, second, these features represent the behaviour that an application may implement.

3.2 Uses-Features of the Manifest File

The `AndroidManifest.xml` file shows other features apart from permissions. Moreover, 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 Figure 3.

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

Figure 3: Declaration of features in an application.

The `android:name` attribute determines the feature (e.g., camera, gps) used by the application, the `android:required` attribute determines whether that feature is mandatory for the correct handling of the application or not, and the `glEsVersion` attribute determines the version of OpenGL, if used. These attributes are represented in the Manifest file under the tag `<uses-feature>` and determine some of the features, both software and hardware, that are required for the correct execution of an application. 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 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, due to the optional character of these features, many applications lack these fields.

In our dataset, the features extracted are related to the use hardware such as localization by means of the GPS, Wi-Fi, or proximity sensors. In light of this context, we 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 vectors 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 hardware and 3 software) possible features, with a binary feature indicating whether the feature is present or not in the analysed Android application.

As an example, Figure 4 shows the declaration of `uses-features` for an Android application.

Figure 1: Example of declaration of features in application

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

Figure 4: Example of the declaration of the `uses-features` in an application.

The input vector of features for this application will be composed of 35 zeros, representing the not used `uses-features` and 1s representing the 2 used features (`android.hardware.camera` and `android.hardware.bluetooth`).

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

4 ANOMALY BASED METHOD

Anomaly detection approaches model normality and try to identify outlier occurrences. In this way, every deviation to this model is considered anomalous. Through the representation described in the previous section, our method represents Android applications as points in the feature space. When an application is being inspected, our method starts by computing the features to represent the sample as a point in the feature space. This point is then compared with the previously calculated points of legitimate applications. To this end, distance measures are required. In this study, we have used the following distance measures:

- *Manhattan Distance*: This distance between two points v and u is the sum of the lengths of the projections of the line segment between the points onto the coordinate axes:

  $$d(x,y) = \sum_{i=0}^{n} |x_i - y_i| \tag{3}$$

  where x is the first point; y is the second point; and x_i and y_i are the i^{th} component of the first and second point, respectively.

- *Euclidean Distance*: This distance is the length of the line segment connecting two points. It is calculated as:

  $$d(x,y) = \sum_{i=0}^{n} \sqrt{v_i^2 - u_i^2} \tag{4}$$

  where x is the first point; y is the second point; and x_i and y_i are the i^{th} component of the first and second point, respectively.

- *Cosine Similarity*: It is a measure of similarity between two vectors by finding the cosine of the angle between them (Tata and Patel, 2007). Since we are measuring distance and not similarity we have used $1 - CosineSimilarity$ as a distance measure:

  $$d(x,y) = 1 - cos(\theta) = 1 - \frac{\vec{v} \cdot \vec{u}}{||\vec{v}|| \cdot ||\vec{u}||} \tag{5}$$

  where $\vec{v}$ is the vector from the origin of the feature space to the first point x, $\vec{u}$ is the vector from the origin of the feature space to the second point y, $\vec{v} \cdot \vec{u}$ is the inner product of $\vec{v}$ and $\vec{u}$. $||\vec{v}|| \cdot ||\vec{v}||$ is the cross product of $\vec{v}$ and $\vec{u}$. This distance ranges from 0 to 1, where 1 means that the two evidences are completely different and 0 means that the evidences are the same (i.e., the vectors are orthogonal between them).

By means of these measures, we are able to compute the deviation of an application with respect to a set of legitimate applications. Other distance measures, such as Mahalanobis distance were discarded due to their complexity. One characteristic of our approach is its simplicity, which allows its deployment in smartphones with low processing capabilities.

Since we have to compute the distance of any application to the points representing valid applications, a combination metric is required in order to obtain a final distance value which considers every measure performed. To this end, our system employs 3 simplistic rules: (i) select the mean value, (ii) select the lowest distance value and (iii) select the highest value of the computed distances.

In this way, when our method inspects an application, a final distance value is acquired, which will depend on both the chosen distance measure and a combination rule.

5 EMPIRICAL VALIDATION

To evaluate our method, we used the dataset described in section 2, composed by 666 samples. Specifically, we followed the next configuration for the empirical validation:

1. *Cross validation*: We performed a 5-fold cross-validation over the benign samples to divide them into 5 different divisions of the data into training and test sets.
2. *Calculating distances and combination rules*: We extracted the `uses-permissions` and `uses-features` of the applications and combined 3 different measures with 3 different combination rules described in section 4 to obtain a final measure of deviation for each testing instance. More accurately, we applied the following distances: (i) the Manhattan Distance, (ii) the Euclidean Distance, and (iii) the Cosine Similarity. For the combination rules we tested the following ones: (i) the mean value, (ii) the lowest distance and (iii) the highest distance.
3. *Defining thresholds*: For each measure and combination rule, we established 10 different thresholds to determine whether a sample is valid or not. The lowest threshold was configured to produce no false negatives, while the highest one was set to produce no false positives.
4. *Testing the method*: We evaluated the method by measuring these parameters:
   - *True Positive Ratio* (TPR), also known as sensitivity: $TPR = TP/(TP + FN)$ where TP are the number of applications correctly classified

(true positives) and FN is the number of applications misclassified as valid ones.

- *False Positive Ratio* (FPR), that is the number of legitimate applications misclassified as malware: $FPR = FP/(FP+TN)$ where FP is the number of valid apps incorrectly detected as malicious while TN is the number of valid apps correctly classified.
- *Accuracy*, which is the total number of hits divided by the number of the instances in the dataset: $Accuracy = (TP+TN)/(P+N)$.
- *Area Under ROC Curve* (Singh et al., 2009), establishes the relation between false negatives and false positives.

Table 2 shows the obtained results. When we applied Manhattan distance, we obtained the best AUC value (0.88), using average as the combination rule. The accuracy obtained was around 85% for this combination. Using euclidean distance, we obtained more than 0.90 of AUC and 87.57% of accuracy. Finally, using cosine distance, we obtained the best results: 0.91 of AUC and nearly 90% of accuracy.

In general, the results obtained surpassed 0.8 of AUC and 80% of accuracy for all distances, considering the average as a combination rule in the three cases.

6 RELATED WORK

In order to tackle the problem of growing malware in Android, researchers have begun to explore this area using the experience acquired in other platforms. We can distinguish two different approaches. Dynamic approaches execute the sample in an isolated environment and collect data about its execution. These approaches require high computational efforts and are not suitable for the deployment on smartphones. Besides, static approaches analyse the samples without executing them. Some attempts are based on signature scanning, that is, detecting known patterns present in malicious applications, while others try to implement generic approaches to distinguish patterns in benign or malicious applications.

Shabtai and Elovici (Shabtai et al., 2012) 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 pieces of 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 required the acquisition of a huge number of features and events, overloading the device and, consequently, draining the battery. Our approach, in contrast, extracts the data from the `AndroidManifest.xml` file, which is a trivial process. Although our results are not as sound as theirs, our approach requires much less computational efforts. In addition, our dataset is larger and sparser in malware samples than theirs.

Regarding the signature based approach, Schmidt, Camtepe, and Albayrak (Schmidt et al., 2010) focused on a static and light-weight analysis of the samples. They used system calls as features and simple classifiers to detect malicious behaviours. Both approaches do not prevent the installation of malware in the devices. Our system evaluates each application before its installation, considering several features extracted from the manifest file, obtaining similar results to those obtained in previous work.

Peng et al. (Peng et al., 2012) proposed an approach to rank the risk of Android applications 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 Android application store Google Play (157,856 and 324,658 samples, respectively) and 378 unique samples of malware. They obtained a 0.94 AUC as best result. Nevertheless, the unbalanced nature of their dataset makes it difficult to directly compare the results with our approach. In fact, our approach is based on anomaly detection, as it measures the deviation of any sample to a set of benign applications. In addition, 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 installation of malware on the devices, instead of monitoring the execution of the applications, thus saving device resources and preventing undesirable consequences.

7 CONCLUSIONS AND FUTURE WORK

Smartphones and tablets are flooding both consumer and business markets and, therefore, these devices manage a large amount of information. Thus, malware writers have found in these devices a new source of income and therefore the number of malware samples has grown exponentially in these platforms.

Table 2: Results for different combination measures and distance rules. The results in bold are the best for each combination rule and distance measure.

(a) Manhattan distance.

Comb.	Thres.	TPR	FPR	AUC	Acc.
	8755.51128	1.00000	1.00000		50.00%
	17468.67613	**0.90991**	**0.21321**		**84.83%**
	26181.84099	0.62042	0.10811		75.62%
	34895.00584	0.51952	0.06006		72.97%
Average	43608.17069	0.42042	0.03303	**0.88167**	69.37%
	52321.33555	0.31471	0.01802		64.83%
	61034.50040	0.27387	0.00601		63.39%
	69747.66525	0.10751	0.00300		55.23%
	78460.83011	0.01862	0.00000		50.93%
	87173.99496	0.00000	0.00000		50.00%
	47404.00000	1.00000	1.00000		50.00%
	54503.66667	0.98498	0.99099		49.70%
	61603.33334	0.90811	0.94595		48.11%
	68703.00000	0.74595	0.78679		47.96%
Max.	75802.66667	0.62943	0.53153	0.65852	54.89%
	82902.33334	**0.52853**	**0.04204**		**74.32%**
	90002.00001	0.35796	0.00901		67.45%
	97101.66667	0.02643	0.00000		51.32%
	104201.33334	0.00480	0.00000		50.24%
	111301.00001	0.00000	0.00000		50.00%
	0.00000	1.00000	1.00000		50.00%
	3470.00000	0.54294	0.13814		70.24%
	6940.00000	**0.52853**	**0.09009**		**71.92%**
	10410.00000	0.41502	0.06006		67.75%
Min.	13880.00000	0.34054	0.02703	0.72431	65.68%
	17350.00001	0.17898	0.00000		58.95%
	20820.00001	0.10450	0.00000		55.23%
	24290.00001	0.02583	0.00000		51.29%
	27760.00001	0.00661	0.00000		50.33%
	31230.00001	0.00000	0.00000		50.00%

(b) Euclidean distance.

Comb.	Thres.	TPR	FPR	AUC	Acc.
	70.35688	1.00000	1.00000		50.00%
	95.24643	0.97057	0.47147		74.95%
	120.13597	0.93814	0.23423		85.20%
	145.02552	**0.88649**	**0.13514**		**87.57%**
Average	169.91506	0.54655	0.09309	**0.906084463**	72.67%
	194.80461	0.46066	0.04805		70.63%
	219.69415	0.37898	0.03303		67.30%
	244.58370	0.27628	0.00601		63.51%
	269.47324	0.04444	0.00300		52.07%
	294.36279	0.00000	0.00000		50.00%
	217.72460	1.00000	1.00000		50.00%
	230.60165	0.98559	0.99399		49.58%
	243.47870	0.93393	0.96096		48.65%
	256.35575	0.82763	0.88288		47.24%
Max.	269.23281	0.68468	0.62462	0.67797888	53.00%
	282.10986	**0.58799**	**0.15015**		**71.89%**
	294.98691	0.44505	0.01201		71.65%
	307.86396	0.05165	0.00000		52.58%
	320.74102	0.00721	0.00000		50.36%
	333.61807	0.00000	0.00000		50.00%
	0.00000	1.00000	1.00000		50.00%
	19.63557	0.64985	0.30030		67.48%
	39.27114	0.56336	0.23423		66.46%
	58.90671	0.54294	0.13814		70.24%
Min.	**78.54228**	**0.53333**	**0.09610**	0.729316704	**71.86%**
	98.17784	0.43063	0.06306		68.38%
	117.81341	0.34054	0.02703		65.68%
	137.44898	0.13634	0.00000		56.82%
	157.08455	0.01562	0.00000		50.78%
	176.72012	0.00000	0.00000		50.00%

(c) Cosine distance

Comb.	Thres.	TPR	FPR	AUC	Acc.
	0.07126978	1.00	1.00		50.00%
	0.17446314	0.89	0.16		86.28%
	0.27765650	**0.86**	**0.08**		**89.04%**
	0.38084985	0.79	0.04		87.24%
Average	0.48404321	0.33	0.01	**0.914959103**	66.22%
	0.58723657	0.33	0.01		66.22%
	0.69042993	0.33	0.01		66.22%
	0.79362328	0.33	0.01		66.22%
	0.89681664	0.33	0.01		66.22%
	1.00001000	0.00	0.00		50.00%
	0.34282500	1.00	1.00		50.00%
	0.41584556	0.99	0.98		50.12%
	0.48886611	0.96	0.97		49.70%
	0.56188667	0.87	0.80		53.18%
Max.	**0.63490722**	**0.87**	**0.80**	0.529312195	**53.06%**
	0.70792778	**0.87**	**0.80**		**53.06%**
	0.78094833	**0.87**	**0.80**		**53.06%**
	0.85396889	**0.87**	**0.80**		**53.06%**
	0.92698944	**0.87**	**0.80**		**53.06%**
	1.00001000	0.00	0.00		50.00%
	-0.10000000	1.00	1.00		50.00%
	0.02222333	**0.64**	**0.04**		**79.85%**
	0.14444667	0.33	0.01		66.22%
	0.26667000	0.33	0.01		66.22%
Min.	0.38889333	0.33	0.01	0.803523343	66.22%
	0.51111667	0.33	0.01		66.22%
	0.63334000	0.33	0.01		66.22%
	0.75556333	0.33	0.01		66.22%
	0.87778667	0.33	0.01		66.22%
	1.00001000	0.00	0.00		50.00%

In this paper, we presented a new malicious software detection approach that is inspired in anomaly detection systems. In contrast to other approaches, this method only needs to previously label goodware and measures the deviation of a new sample respect to normality (applications without malicious intentions). Although anomaly detection systems tend to produce high error rates (specially, false positives), our experimental results show low FPR values. The number of malicious samples discovered up-to-date is limited: signature scanning methods are effective and efficient solutions for current malware. Nevertheless, we consider that malware authors will soon apply obfuscation techniques making difficult the detection process. This possibility was explored by Rastogi et al. (Rastogi et al., 2013). In this way, our approach reduces the necessity to collect malware samples (i.e., it is not necessary to update a signature database), as it is based on anomaly detection. In addition, our method is based on features that are extracted from the manifest file, making possible to prevent the installation of malicious software.

However, this approach presents several limitations. By means of an internet connection, a benign application can download a malicious payload and change its behaviour. In order to detect this behaviour, a dynamic approach is required. Unfortunately, dynamic approaches cannot be deployed in current smartphones due to their computational and battery limitations.

Future work is oriented in two main directions. On the one hand, other distance measures and combination rules could be tested. On the other hand, there are other static features that could be used to improve the detection ratio, that could be obtained from the `AndroidManifest.xml` file or from the binary class (e.g., strings, API calls). The use of different features could reduce the risk of incorrectly classifying benign applications that have permission usage declarations similar to malicious samples.

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

Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., and Albayrak, S. (2010). An android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 55–62. IEEE.

Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. (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*, pages 15–26. ACM.

Egele, M., Kruegel, C., Kirda, E., and Vigna, G. (2011). Pios: Detecting privacy leaks in ios applications. In *Proceedings of the Network and Distributed System Security Symposium*.

Mylonas, A., Kastania, A., and Gritzalis, D. (2012). Delegate the smartphone user? security awareness in smartphone platforms. *Computers & Security*.

Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., and Molloy, I. (2012). Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 241–252. ACM.

Rastogi, V., Chen, Y., and Jiang, X. (2013). Evaluating android anti-malware against transformation attacks.

Schmidt, A.-D., Camtepe, A., and Albayrak, S. (2010). Static smartphone malware detection. In *proceedings of the 5th Security Research Conference (Future Security 2010), ISBN*, pages 978–3.

Shabtai, A. and Elovici, Y. (2010). Applying behavioral detection on android-based devices. *Mobile Wireless Middleware, Operating Systems, and Applications*, pages 235–249.

Shabtai, A., Fledel, Y., and Elovici, Y. (2010). Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333. IEEE.

Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y. (2012). andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, pages 1–30.

Singh, Y., Kaur, A., and Malhotra, R. (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.

Tata, S. and Patel, J. M. (2007). Estimating the selectivity of tf-idf based cosine similarity predicates. *ACM SIGMOD Record*, 36(2):7–12.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE.