Using Dalvik opcodes for Malware Detection on Android

José Gaviria de la Puerta, Borja Sanz, Igor Santos and Pablo García Bringas

DeustoTech Computing, University of Deusto
jgaviria@deusto.es, borja.sanz@deusto.es,
isantos@deusto.es, pablo.garcia.bringas@deusto.es

Abstract. Over the last few years, computers and smartphones have become essential tools in our ways of communicating with each-other. Nowadays, the amount of applications in the Google store has grown exponentially, therefore, malware developers have introduced malicious applications in that market. The Android system uses the Dalvik virtual machine. Through reverse engineering, we may be able to get the different opcodes for each application. For this reason, in this paper an approach to detect malware on Android is presented, by using the techniques of reverse engineering and putting an emphasis on operational codes used for these applications. After obtaining these opcodes, machine learning techniques are used to classify apps.
Keywords: Android, Malware, Opcodes, Detection, Machine Learning.

1 Introduction

As we all known, in the last few years, mobile terminals, also known as smartphones, have become very popular devices. Nowadays, many smartphones have more computing capabilities and memory than many computers that are only a few years old.

Like any personal device, the smartphone uses an operating system, which is usually pretty friendly for that the users do not have problems when using it. Throughout history, different operating systems to these devices have emerged. In an interview with Andy Rubin, former Android boss, he stated that "there should be nothing that users can access on their desktop that they cannot access on their cell phone."[1]. Thanks to this sentence, we can demonstrate the progress that these small computers are having. By its hardware, which consists of a wide range of sensors such as camera, accelerometer and GPS, we are provided a wealth of information and data, which put a number of additional requirements to mobile operating systems.

People use mobile devices for a wide range of purposes as if they were desktop computers: web browsing, social networking, online banking, and more. The so-called smartphones also offer features that are unique to mobile phones like, for instance, SMS messaging, location data constantly updated, and ubiquitous access. As a result of their popularity and functionality, smartphones are a growing target of malicious activity.

Today, one of the most common ways to perform malicious actions is by using malicious code or malware. The term *malware* comes from the Anglo-Saxon words *MALicious* and *softWARE*, which comes to mean malicious software. Typically, such software poses as legitimate applications to run its malicious actions without the user's knowledge. One of the primary objectives of the malware is to make enormous profit, whether economic or theft of information, for as long as possible. In order to achieve this goal, these malicious applications try to stay hidden in the system without the user to see an anomalous behaviour in the device.

It is well known that malware has grown in recent years and that is has become one of the biggest threats in recent times. According to Kaspersky Labs antivirus company, about 145,000 new malware samples for mobile devices appeared in 2013, tripling the samples detected in the previous year[1]. Lately, malware has increased exponentially, specially in the Android platform.

The section 2 is a small state of the art with related researchs of android systems that have been carried out by the scientific community. In section 3 we present the scope of this experiment, and how we obtained the necessary information to perform it. Section 4 summarizes the different classifiers used for this experiment. Section 5 makes a statement of the results, and the parameters to consider with such experimental validation. Finally, the section 6 shows the conclusions obtained by the experiment and some possible lines of future work to be carried out.

2 Related Work

The Android operating system is designed to run each application on its own virtual machine, Dalvik. This type of implementation makes the system more robust and limits the damage caused by bad programming[2].

In addition, the Android systems are supplied with a permission system that does not allow third party applications to access resources that should not be accessible. These permissions are assigned to the application at installation time but the user is the one that has to make that final decision. He or she is the person to choose to install the application or not, depending on the permissions that the application has. Malicious applications developers can request more permissions than they really need, for example, to obtain private user information[3]. Along this line, a research which was conducted by Sanz et al. [4] showed us that just by taking a look at the permissions that the app required, they could classify it as malicious or benign. Other authors have also used this feature in their researches [5, 6]. Nevertheless, Android is a system that is constantly changing and now takes groups of permissions. With those clusters, these approaches lose effectiveness.

[1] `http://www.kaspersky.com/about/news/virus/2014/Mobile-malware-evolution-3-infection-attempts-per-user-in-2013?ClickID=c4azsxkfiallqkfvsvzavqvkz4ixn4q7fnqn`

A classical approach in mobile devices is to analyse the behaviour of different hardware components, such as the CPU or the battery, searching for anomalies[7, 8]. However, even if it is true that these approaches can help us discover strange behaviour in the system, they have the problem of the complexity in the programming, and also, the constant updates that a program could have in a month. These problems can impact in these elements but not for that reason are malicious applications.

Meanwhile, Schmidt et al. [9] developed a framework that used system calls as a feature for the classification of benign and malicious applications in the Android system. Still, this type of classification has the problem that it is very expensive to get such information. Furthermore, depending on the number of calls, the performance of this approach may be very low.

Shabtai et al. [10] did some research using the decompiled files of an Android application. In this study, they managed to classify malware using extracted features, such as the classes used by the application, as well as some machine learning techniques. For this approach we need a very large number of features extracted from the applications to get a good ranking.

One of the techniques that is often used in the creation of *Android malware* is to use a legitimate application and include a malicious code in it. Along this line, Zhou et al. [11] did some research on third party app stores. A major limitation of this approach is the necessity of the legitimate application to see if malicious code is inserted into it.

3 Experimentation scope

In this context, following the guidelines used for the detection of malicious code on desktops led by Santos et al [12], the authors propose a study that consists of: 1) Capture of information on the operational codes used by an application, 2) malicious or benign classification of an application, by using machine learning techniques.

After obtaining all applications, the main objective of the research is the ability, through supervised learning techniques, to detect with the fewest false positives and false negatives, the malicious apps generated to be used within the Android platform.

In the next subsections, we define the methodology that has been used for modelling the applications, as well as its characteristics that will be used to present this proof of concept. This methodology uses all the possibilities of the application created by the University of Waikato, Weka[2], to employ different ranking algorithms.

[2] Weka: Data Mining Software is a collection of machine learning algorithms for automated data mining tasks: http://www.cs.waikato.ac.nz/ml/weka/

3.1 Samples collection

On the one hand, to obtain benign code samples, we used the Selenium[3] application for automating web browser. With this automation we use the application web APIfy[4] for downloading Android applications from different categories.

On the other hand, the samples of malicious code have been used throughout the dataset provided by *Android Genome Project*[13].

3.2 Information Gathering

This phase has proceeded to create a platform in the programming language C#. This platform has been implemented as a plugin for obtaining different opcodes used by an apk to execute their actions. Studying these opcodes, a particular one called *RSUB_INT* has been found. It is only published on benign code applications, specially in 639 applications.

4 Machine learning and supervised classification

In this paper the authors present an experimental model that makes use of modeling techniques described above. The objective is to obtain the opcodes for such application that may be employed by some classifiers for classify applications.

In problems of machine learning and supervised classification, a phenomenon represented by a vector X in R^d which can be classified in K ways according to *label* Y, is studied.

To this end, we have $D_n = \{(X_i, Y_i)\}_{i=1}^{n}$ called *training set*, where X_i represents the events corresponding to the phenomenon X while Y_i is the label that puts it in the category that the classifier takes as correct. For example, in the present case, we are talking about an application X_i defined by a set of opcodes that represent it, where Y_i the category assigned to that application as estimated by the classifier.

In this learning case, all classifiers were done using the method of representation of «*vector space*» model for the representation as a vector of different frequencies of opcodes.

This method represents documents in natural language to a formal way using vectors in a multi-dimensional linear space. The basic form of this method is represented by the cosine of the angle between the two vectors generated for the similarity between the terms.

4.1 Classification algorithms

In this research, we have chosen to compare the performance of different classification algorithms given the occasionally notable differences in effectiveness that can be observed in similar experiments conducted in other areas [14]. The

[3] http://docs.seleniumhq.org/
[4] http://apify.ifc0nfig.com/

algorithms used for the tests in Section 5 are the following: Random Forest, J48, Bayes Theorem-based algorithms, K-Nearest Neighbor (KNN), Sequential Minimal Optimization (SMO) and Simple Logistic.

- Random Forest. Random Forest is an aggregation classifier developed by Leo Breiman [15] which is formed by a bunch of decision trees considered in a way in which the introduction of a stochastic component improves de precision of the classifier , either in the construction of the trees or either in the training dataset.
- J48. J48 is an open source implementation for Weka of the C4.5 algorithm [16]. C4.5 creates decision trees given an amount of training information making use of the concept *information entropy* [17]. The training data consist of a group $S = s_1, s_2, ..., s_n$ of already classified samples $s_1 = x_1, x_2, ..., x_m$ in which $x_1, x_2, ..., x_m$ represent the attributes or characteristics of each sample. In each node of the decision tree, the algorithm will choose the attribute in the data that most efficiently divides the dataset in enriched choruses of a given class using the entropy difference or the already mentioned *normalized information gain* as selective criteria.
- Bayes Theorem-based algorithms. The Bayes Theorem, the base for the Bayesian inference, is a statistical method which determines, based on a number of observations, the probability of a certain hypotheses being true. For the classification needs here exposed, this is the most important capability of Bayesian networks: in our case, the probability of an app being malicious or bening The theorem is capable of adjusting the probabilities as soon as new observations are performed. Thus, Bayesian networks conform a probabilistic model that represents a collection of randomized variables and their conditional dependencies by means of a directed graph. We have trained our models with three different search algorithms: K2, Hill Climbing and TAN.
  In this group we have also considered the inclusion of Naïve Bayes. The idea is that if the number of independent variables managed is too big, it does not make any sense to make probability tables [18]. Then, the reduced model with simplified datasets give to the algorithm the appellative of *Naïve*.
- K-Nearest Neighbor (KNN). The KNN algorithm is one of the most simple classification algorithms amongst all of those available for the machine learning techniques. It takes decisions based on the results of the k closest neighbours to the analysed sample in the experimental n-dimensional space ($\forall k \in \mathbb{N}$).
  In this case, and taken into account the simplicity of the algorithm, we have explored even more values ($k = 1, 3, 5$) so as to determine if this enlargement would throw any kind of additional advantage.
- Sequential Minimal Optimization (SMO). SMO, invented by John Platt [19], is an iterative algorithm used for the solution of the optimization problems that appear when training *Support Vector Machines* (SVM). Basically, SMO divides the problem into a series of smaller subproblems which are analytically solved lately.

At this point, we have selected different kernels with these algorithms: a polynomial kernel, a normalized polynomial kernel, RBF and Pearson-VII.

– Simple Logistic. This algorithm is used to predict the result of a variable function of the independent variables, or predictor. The logistic regression formula is:

$$Y_i = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_{1,i} + \ldots + \beta_d X_{d,i})}} \tag{1}$$

Being Y_i the classification to predict by the model, in our case it would be *goodware* or *malware*. The variable X is the vector with the opcodes generated for a specific application, finding that $X_{d,i}$ is the value assigned to an n-gram of opcodes in d enforcement position that is in row i. Parameters $beta\ _{ast}$ are determined by the algorithm in the training phase.

So, having a downloaded and validate, using the VirusTotal platform[5], dataset as goodware, and secondly, using the Android Genome Project[13] dataset with malicious applications, it has been easy to label the purpose of each of the applications as *goodware* or *malware* to generate the *training datasets* used in the section 5.

5 Experimental validation

Next step, we recollect the information stored in each of the apps. First of all, a total of 1,494 applications from *Google Play* [6] were downloaded. In particular we got them from the lists of "the most free downloaded", in "the most free downloaded in Spanish", applications that "have generated more revenue", those of "lifestyle" and those of "tools".

The choice of these lists for obtaining samples was completely random to have a heterogeneous applications dataset from Google's store. To verify that the samples do not contain malicious code, we have used the online platform VirusTotal. This platform uses 43 antivirus engines to scan the sample that previously has been sent. This analysis returns the total number of engines that have been detected as malicious and malware is for that engine. Since the experiment is desired to have all possible clean malware samples, it was decided that if only one antivirus engine detected it sample as malicious, it would be separated from the dataset. The ones that were detected as adware[7], were also separated from that dataset.

After the analysis with VirusTotal, we found that we had a 14.59% of applications considered malware or adware in our dataset. Specifically there were 218 apps which, according to the metrics we have said before, could not be within the dataset itself benign code. Given that for the dataset of malicious code we used the one provided by the *Android Genome Project*, which consists of 1,259

[5] https://www.virustotal.com/es/

[6] https://play.google.com/store

[7] Adware is a type of action hidden in applications, which send targeted advertisements to our device when you run an application

samples, so our dataset benign still had to be reduced in 17 more samples to be balanced . These samples were selected randomly from the total, leaving the final data set with 2,518 samples applications, half benign and half malignant.

For the information of the samples was done using the Dedexer tool, a disassembler for .dex files that is on the Android platform. Using this tool, we obtained the operational codes of the different samples analyzed. These codes are the minimum operating instructions means the Dalvik virtual machine to run applications on it. Figure 1 we see a sequence extracted from an Android application opcodes.

Fig. 1. Example of operational codes with variables used in an Android application.

```
const-string       v0, aCursor
invoke-interface   v3, v0, <ref Map.get(ref)
                   imp. @ _def_Map_get@LL>
move-result-object v0
check-cast         v0, <t: String>
const-string       v1, aHasmore
invoke-interface   v3, v1, <ref Map.get(ref)
                   imp. @ _def_Map_get@LL>
move-result-object v1
check-cast         v1, <t: String>
const-string       v3, aTrue_0
invoke-virtual     v3, v1, <boolean String.equalsIgnoreCase(ref)
                   imp. @ _def_String_equalsIgnoreCase@ZL>
move-result        v7
invoke-virtual     this, v0,
                   <boolean KiwiPurchaseUpdatesCommandTask.isNullOrEmpty(ref)
                   imp. @ _def_KiwiPurchaseUpdatesCommandTask_isNullOrEmpty@ZL>
move-result        v1
if-eqz             v1, loc_C1AD2
sget-object        v6, Offset_BEGINNING
```

With all the samples we have disassembled, we generated opcode files ranging from 10 KB to 32 MB for the goodware and from 5 KB to 20 MB in the case of malware. In these files we can meet the different operational codes used by both benign and malicious applications. From here there has been a Arff file for use with the Weka tool. For each of them, an experiment to demostrate its validity as predictors using different classifiers detailed in section 4.

Thus, this section will detail the results obtained when evaluating the different opcodes.

This evaluation will be performed according to the following parameters, usually employed to compare the performance of different algorithms in the field of machine learning:

– *True Positive Ratio* (TPR),which is calculated by dividing the number of bening apps correctly classified (TP) between the total samples taken ($TP+FN$).

$$TPR = \frac{TP}{(TP+FN)} \tag{2}$$

Table 1. Performance of classifiers when analyzed correctly categorize applications using opcodes.

Classiffier	**TPR**	**FPR**	**Precision (%)**	**ROC**
IBk 1	0,94408	0,04217	95,761	0,97198
IBk 3	0,93558	0,04202	95,727	0,97924
IBk 5	0,93344	0,04432	95,495	0,98102
Simple Logistic	0,92177	0,05535	94,379	0,97839
NaiveBayes	0,83789	0,20294	80,561	0,88461
BayesNet K2	0,82057	0,19325	81,006	0,87881
BayesNet TAN	0,89365	0,09793	90,181	0,94768
SMO PolyKernel	0,92995	0,04916	95,022	0,94039
SMO Norm. PolyKernel	0,90985	0,04828	95,002	0,93078
J48	0,92581	0,05901	94,071	0,94434
RandomTree	0,91454	0,06147	93,749	0,92654
RandomForest I=10	0,95147	0,05091	94,953	0,98879
RandomForest I=50	0,94853	0,03645	96,322	0,99208
RandomForest I=100	0,94829	0,032	96,758	0,99255

– *False Positive Ratio* (FPR),which is calculated by dividing the number of samples corresponding to malicious app whose classification (FP) were missed by the total number of samples ($FP+TN$).

$$FPR = \frac{FP}{(FP+TN)} \tag{3}$$

– *Precisión* (P), which is calculated by dividing the total hits by the total number of instances in the dataset.

$$P = \frac{TP+TN}{TP+FP+TN+FN} \tag{4}$$

– *Area Under ROC Curve* (AUC) [14], that establishes the relationship amongst the false negatives and the false positives. The ROC Curve it is usually used to generate statistics that represent the performance or the effectiveness in a wider sense of a classifier.

As such, the results are displayed in Table 1. The best results have been obtained for the Random Forest classifier with the number of trees equal to 100, with a classification accuracy of 96.758% and value of the area under the ROC curve of 0.99255.

In contrast, it has been observed that classifiers that have worked have been worse Naive Bayes (with an overall accuracy of 80.561 % and a value of $AURC$ of 0.88461) and BayesNet with K2 algorithm (with a total accuracy of 81.006 % and a value of $AURC$ of 0.87881).

In Table 2 we can see the results of classification of malicious applications using application permissions on our dataset, conducted by Sanz et al. [4]. In this table we can see one of the best classifiers using application permissions is

Table 2. Performance of classifiers when analyzed correctly categorize applications using permissions.

Classiffier	**TPR**	**FPR**	**Precision (%)**	**ROC**
IBk 1	0,94615	0,04924	95,103	0,98597
IBk 3	0,93701	0,05281	94,717	0,98737
IBk 5	0,93312	0,05893	94,127	0,98709
Simple Logistic	0,96235	0,0587	94,288	0,9894
NaiveBayes	0,95392	0,21255	81,854	0,95679
BayesNet K2	0,96981	0,22208	81,463	0,96868
BayesNet TAN	0,97689	0,09284	91,378	0,98233
SMO PolyKernel	0,96616	0,05957	94,221	0,95329
SMO Norm. PolyKernel	0,9745	0,05433	94,751	0,96009
J48	0,95195	0,06457	93,686	0,96116
RandomTree	0,93178	0,06179	93,824	0,95412
RandomForest I=10	0,961	0,04892	95,189	0,99148
RandomForest I=50	0,95806	0,04551	95,511	0,99361
RandomForest I=100	0,94829	0,04654	95,423	0,99382

Random Forest with number of trees equal to 100, wich is taking the area under the ROC curve of 0.99382 and an accuracy of 95.423%.

We can also see that one of the worst is SMO classifiers using Normalized PolyKernel with an area under the ROC curve of 0.95329 and an accuracy of 94.751 %.

Comparing the two methods it can be seen that the two approaches follow the same trend roughly in values obtained. On the one hand we can see that the area under the ROC curve values tend to be 95% or higher in most of the classifiers. On the other hand if you consider that the best classifier for both approaches is the Random Forest Tree number 100.

6 Conclusions and future work

The opcodes are instructions performed by an application. All Apps must use these codes to perform the activities for which they are scheduled. In this paper we use the operational codes with machine learning techniques for classification of malicious apps on the Android operating system, with a comparison with application permissions. To validate our approach, we collected 1,259 samples totally benign applications from Google Play and 1,259 Android malware samples obtained from the «*Android Genome Project*». After that, the operational codes were extracted and models were created to evaluate each configuration of classifiers by the area under the ROC curve.

As the application permissions are very quick to get, operational codes require more computation time, thus penalizing the performance of the approach. In contrast, nowadays there are no longer unique permissions as before, but there are groups of permissions, making this approach not so optimal. Therefore, today you can use this method as one of the first approaches to malware detection on Android.

As future work, the analysis of 639 samples containing the opcode «*RSUB_INT*» is proposed. By doing this we would be able to see why this opcode only appears in benign samples.

This study may result in the generation of a pattern. This would be used to detect unknown malware faster and it could also limit the amount of possible malicious codes in many applications.

References

1. Waters, D.: Google bets on Android future (February 2008) `http://news.bbc.co.uk/2/hi/technology/7266201.stm`.
2. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: USENIX Security Symposium. (2011)
3. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing android's permission system. In: Computer Security–ESORICS 2012. Springer (2012) 1–18
4. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P.G., Álvarez, G.: Puma: Permission usage to detect malware in android. In: International Joint Conference CISIS'12-ICEUTE´ 12-SOCO´ 12 Special Sessions, Springer (2013) 289–298
5. Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: Towards formal analysis of the permission-based security model for android. In: Wireless and Mobile Communications, 2009. ICWMC'09. Fifth International Conference on, IEEE (2009) 87–92
6. Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A formal model to analyze the permission authorization and enforcement in the android framework. In: Social Computing (SocialCom), 2010 IEEE Second International Conference on, IEEE (2010) 944–951
7. Jacoby, G.A., Davis IV, N.J.: Battery-based intrusion detection. In: Global Telecommunications Conference, 2004. GLOBECOM'04. IEEE. Volume 4., IEEE (2004) 2250–2255
8. Buennemeyer, T.K., Nelson, T.M., Clagett, L.M., Dunning, J.P., Marchany, R.C., Tront, J.G.: Mobile device profiling and intrusion detection using smart batteries. In: Hawaii International Conference on System Sciences, Proceedings of the 41st Annual, IEEE (2008) 296–296
9. Schmidt, A.D., Bye, R., Schmidt, H.G., Clausen, J., Kiraz, O., Yuksel, K.A., Camtepe, S.A., Albayrak, S.: Static analysis of executables for collaborative malware detection on android. In: Communications, 2009. ICC'09. IEEE International Conference on, IEEE (2009) 1–5
10. Shabtai, A., Fledel, Y., Elovici, Y.: Automated static code analysis for classifying android applications using machine learning. In: Computational Intelligence and Security (CIS), 2010 International Conference on, IEEE (2010) 329–333
11. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of the second ACM conference on Data and Application Security and Privacy, ACM (2012) 317–326

12. Santos, I., Brezo, F., Sanz, B., Laorden, C., Bringas, P.G.: Using opcode sequences in single-class learning to detect unknown malware. IET information security **5**(4) (2011) 220–227
13. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Security and Privacy (SP), 2012 IEEE Symposium on, IEEE (2012) 95–109
14. Singh, Y., Kaur, A., Malhotra, R.: Comparative analysis of regression and machine learning methods for predicting fault proneness models. International Journal of Computer Applications in Technology (2009) **35**(2) (2009) 183–193
15. Breiman, L.: Random forests. Machine Learning **45** (2001) 5–32 10.1023/A:1010933404324.
16. Quinlan, J.: C4. 5: programs for machine learning. Morgan kaufmann (1993)
17. Salzberg, S.L.: C4.5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993. Machine Learning **16** (1994) 235–240 10.1007/BF00993309.
18. Jiang, L., Wang, D., Cai, Z., Yan, X.: Survey of improving naive bayes for classification. In Alhajj, R., Gao, H., Li, X., Li, J., Zaïane, O., eds.: Advanced Data Mining and Applications. Volume 4632 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2007) 134–145 10.1007/978-3-540-73871-8_14.
19. Platt, J.C.: Sequential minimal optimization: A fast algorithm for training support vector machines (1998)