Anomaly Detection using String Analysis for Android Malware Detection

Borja Sanz, Igor Santos, Xabier Ugarte-Pedrero, Carlos Laorden, Javier Nieves and Pablo G. Bringas

[1]S[3]Lab, University of Deusto
Avenida de las Universidades 24, 48007 Bilbao, Spain
{borja.sanz, isantos, xabier.ugarte, claorden, jnieves, pablo.garcia.bringas}@deusto.es

Abstract. The usage of mobile phones has increased in our lives because they offer nearly the same functionality as a personal computer. Specifically, Android is one of the most widespread mobile operating systems. Indeed, its app store is one of the most visited and the number of applications available for this platform has also increased. However, as it happens with any popular service, it is prone to misuse, and the number of malware samples has increased dramatically in the last months. Thus, we propose a new method based on anomaly detection that extracts the strings contained in application files in order to detect malware.

Keywords: malware detection, anomaly detection, Android, mobile malware

1 Introduction

Smartphones have become extremely useful gadgets, very rich in functionality. Their operating systems have evolved, becoming closer to the desktop ones. We can read our email, browse the Internet or play games with our friends, wherever we are. In addition, the smartphone functionality can be enhanced, similarly to desktop computers, through the installation of software applications.

In recent years, a new approach to distribute applications has gained popularity: application stores. These stores, which distribute software and manage the payments, have become very successful. Apple's AppStore was the first online store to bring this new paradigm to users and now offers more than 800,000 applications[1]. In a similar way, Google's Play Store, Android's official application store, hosts 675,000 apps[2].

Nevertheless, this success has also drawn attention to criminals and many malware samples have emerged. According to Kaspersky, more than 35,000 new samples were identified in 2012, which represents six times the number detected in 2011.

[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

Several approaches have been proposed to deal with this issue. Crowdroid [1] is an approach that analyses the behaviour of the applications through device usage features. Blasing et al. created AASandbox [2], which is a hybrid (i.e., dynamic and static) approximation. The approach is based on the analysis of the logs for the low-level interactions obtained during execution. Shabtai and Elovici [3] also proposed a Host-Based Intrusion Detection System (HIDS) which used machine learning methods to determine whether the 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. In previous work, we used permissions to train machine-learning algorithms in order to detect malware [4], obtaining more than 86% accuracy and 0.92 of Area Under ROC curve.

However, machine-learning classifiers (or supervised-learning methods) require a high number of labelled applications for each of the classes (i.e., malware or benign applications) to train the different models. Unfortunately, it is quite difficult to acquire this amount of labelled data for a real-world problem such as malware detection. In order to compose such data-set, a time-consuming process of analysis is mandatory that renders in a cost increment.

In light of this background, we present an anomaly detection method for the detection of malware in Android. This method employs the strings contained in the disassembled Android applications, constructing a bag of words model in order to generate an anomaly detection model that measures deviations from normality (i.e., legitimate applications).

In summary, our main contributions are: (i) we present a new technique for the representation of Android applications, based on the bag of words model formed by the strings contained in the disassembled applications; (ii) we propose a new anomaly-based malware detection method for Android; and (iii) we show that this approach can provide detection of malicious applications in Android using the strings contained in the disassembled application as features.

The remainder of this paper is organised as follows. Section 2 presents and details our approach to represent applications in order to detect malware in Android. Section 3 describes the anomaly detection techniques that our approach is using. Section 4 describes the empirical evaluation of our method. Finally, section 5 concludes and outlines the avenues of further work in this area.

2 Representation of Applications using String Analysis

One of the most widely-used techniques for classic malware detection is the usage of strings contained in the files [5, 6]. This technique extracts every printable string within an executable file. The information that may be found in these strings can be, for example, options in the menus of the application or malicious URLs to connect to. In this way, by means of an analysis of these data, it

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

is possible to extract valuable information in order to determine whether an application is malicious or not.

The process adopted in our approach is the following. First, we disassemble the application using the open-source Android disassembler `smali`[4]. Afterwards, we search for the `const-string` operational code within the disassembled code.

Using this disassembler, the representation of Android binaries is semantically richer than common desktop binaries. For example, the strings extraction in desktop binaries is complex and, usually, malware writers use obfuscation techniques to hide relevant information. Instead, the obfuscation of strings in Android binaries is more difficult, given the internal structure of binaries in this platform.

In order to conform the strings, we tokenise the symbols found using the classic separators (e.g., dot, comma, colon, semi-colon, blank space, tab, etc.). In this way, we construct a text representation of an executable $\mathcal{E}$, which is formed by strings s_i, such that $\mathcal{E} = (s_1, s_2, ..., s_{n-1}, s_n)$ where n is the number of strings within a file.

$\mathcal{C}$ is the set of Android executables $\mathcal{E}$, $\{\mathcal{E} : \{s_1, s_2, ...s_n\}\}$, each comprising n strings $s_1, s_2, \ldots, s_n$. We define the weight $w_{i,j}$ as the number of times the string s_i appears in the executable $\mathcal{E}_j$; if s_i is not present in $\mathcal{E}$, $w_{i,j} = 0$. Therefore, an application $\mathcal{E}_j$ can be represented as the vector of weights $\boldsymbol{\mathcal{E}_j} = (w_{1,j}, w_{2,j}, ...w_{n,j})$.

In order to represent a string collection, a common approach in text classification is to use the Vector Space Model (VSM) [7], which represents documents algebraically, as vectors in a multidimensional space.

This space consists only of positive axis intercepts. Executables are represented by a string-by-executable matrix, where the $(i, j)^{th}$ element illustrates the association between the i^{th} string and the j^{th} executable. This association reflects the occurrence of the i^{th} string in the executable j. Strings can be individually weighted, allowing the strings to become more or less important within a given executable or the executable collection $\mathcal{C}$ as a whole.

We used the *Term Frequency – Inverse Document Frequency* (TF–IDF) [8] weighting schema, where the weight of the i^{th} string in the j^{th} executable, denoted by $weight(i, j)$, is defined by:

$$weight(i, j) = tf_{i,j} \cdot idf_i \tag{1}$$

where *term frequency* $tf_{i,j}$ is defined as:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{2}$$

where $n_{i,j}$ is the number of times the string s_i appears in a executable $\mathcal{E}_j$, and $\sum_k n_{k,j}$ is the total number of strings in the executable $\mathcal{E}_j$. The inverse term frequency idf_i is defined as:

[4] `http://code.google.com/p/smali/`

$$idf_i = \log\left(\frac{|\mathcal{C}|}{|\mathcal{C} : t_i \in \mathcal{E}|}\right) \tag{3}$$

where $|\mathcal{C}|$ is the total number of executables and $|\mathcal{C} : s_i \in \mathcal{E}|$ is the number of executables containing the string s_i.

3 Anomaly Detection Techniques

Anomaly detection models what it is a normal application and every deviation to this model is considered anomalous. Our method represents Android applications as points in the feature space, using the method described in the previous section. When an application is being inspected, our method starts by computing the values of the points in the feature space. This point is then compared with the previously calculated points of the 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 x and y is the sum of the lengths of the projections of the line segment between the points onto the coordinate axes:
$$d(x, y) = \sum_{i=0}^{n} |x_i - y_i| \tag{4}$$
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.
- *Euclidean Distance*: This distance is the length of the line segment connecting two points. It is calculated as:
$$d(x, y) = \sum_{i=0}^{n} \sqrt{v_i^2 - u_i^2} \tag{5}$$
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 consists of measuring the similarity between two vectors by finding the cosine of the angle between them [9]. Since we are measuring distance and not similarity, we have used $1 - CosineSimilarity$ as a distance measure:
$$d(x, y) = 1 - cos(\theta) = 1 - \frac{\boldsymbol{v} \cdot \boldsymbol{u}}{||\boldsymbol{v}|| \cdot ||\boldsymbol{u}||} \tag{6}$$
where $\boldsymbol{v}$ is the vector from the origin of the feature space to the first point x, $\boldsymbol{u}$ is the vector from the origin of the feature space to the second point y, $\boldsymbol{v} \cdot \boldsymbol{u}$ is the inner product of $\boldsymbol{v}$ and $\boldsymbol{u}$. $||\boldsymbol{v}|| \cdot ||\boldsymbol{v}||$ is the cross product of $\boldsymbol{v}$ and $\boldsymbol{u}$. 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 can compute the deviation of an application with respect to a set of benign ones.

Since we have to compute this measure with the points representing valid apps, 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.

4 Empirical validation

To evaluate our method, we used a dataset composed of 666 samples: 333 malicious applications and 333 legitimate apps. Malicious applications were gathered from the company VirusTotal[5]. VirusTotal offers a series of services called VirusTotal Malware Intelligence Services, which allow researchers to obtain samples from their databases.

To evaluate our anomaly-based approach, we followed the next configuration for the empirical validation:

1. *Cross validation*: We performed a 5-fold cross-validation over benign samples to divide them into 5 different divisions of the data into training and test sets.
2. *Computation of distances and combination rules*: We extracted the strings from the applications and used the 3 different measures and the 3 different combination rules described in Section 3 to obtain a final measure of deviation for each testing evidence. More accurately, we applied the following distances: (i) the Manhattan distance, (ii) the Euclidean distance and (iii) the Cosine distance. For the combination rules, we tested the following ones: (i) the mean value, (ii) the lowest distance and (iii) the highest value.
3. *Defining thresholds*: For each measure and combination rule, we established 10 different thresholds to determine whether a sample is valid or not.
4. *Testing the method*: We evaluated the method by measuring these parameters:
   - *True Positive Ratio* (TPR), also known as sensitivity.

$$TPR = \frac{TP}{(TP + FN)} \tag{7}$$

     where TP is the number of applications correctly classified as malware and FN is the number of applications misclassified as benign software.

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

Table 1. Results for the different combination measures using Manhattan Distance. The results in bold are the best for each combination rule and distance measure.

Comb.	Thres.	TPR	FPR	AUC	Acc.
Average	1042516.95554	1.00000	1.00000	0.29121	50.00%
	1297301.02116	0.56456	0.89189		33.63%
	1552085.08679	0.47988	0.82282		32.85%
	1806869.15241	0.37117	0.73273		31.92%
	2061653.21803	0.23964	0.55856		34.05%
	2316437.28365	0.15856	0.34835		40.51%
	2571221.34927	0.11051	0.17117		46.97%
	2826005.41490	**0.07568**	**0.03303**		**52.13%**
	3080789.48052	0.00240	0.02102		49.07%
	3335573.54614	0.00000	0.00300		49.85%
Max.	2115684.24601	1.00000	1.00000	**0.30224**	49.85%
	2386946.41796	0.90871	0.98198		50.00%
	2658208.58992	0.54294	0.87387		46.34%
	2929470.76187	0.43724	0.79880		33.45%
	3200732.93383	0.30931	0.67868		31.92%
	3471995.10578	0.21201	0.43243		31.53%
	3743257.27774	0.14114	0.19520		38.98%
	4014519.44969	0.08468	0.07207		47.30%
	4285781.62165	0.06006	0.00601		50.63%
	4557043.79360	**0.00000**	**0.00000**		**52.70%**
Min.	**0.00000**	**1.00000**	**1.00000**	0.29987	**50.00%**
	274472.12573	**0.54655**	**0.85285**		**50.00%**
	548944.25146	0.44805	0.75976		34.68%
	823416.37718	0.33333	0.64565		34.41%
	1097888.50291	0.20721	0.50751		34.38%
	1372360.62864	0.14114	0.32733		34.98%
	1646832.75437	0.09610	0.15315		40.69%
	1921304.88009	0.00300	0.03604		47.15%
	2195777.00582	0.00000	0.02102		48.35%
	2470249.13155	0.00000	0.00000		48.95%

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

Comb.	Thres.	TPR	FPR	AUC	Acc.
Average	1292555.44	0.99880	1.00000	0.321240159	49.94%
	1494979.97	0.53814	0.86787		33.51%
	1697404.49	0.39880	0.76577		31.65%
	1899829.02	0.27327	0.56757		35.29%
	2102253.55	0.19700	0.25526		47.09%
	2304678.08	0.12613	0.13213		49.70%
	2507102.61	**0.10511**	**0.04805**		**52.85%**
	2709527.14	0.02402	0.01802		50.30%
	2911951.67	0.01502	0.00601		50.45%
	3114376.19	0.00000	0.00000		50.00%
Max.	2486071.46	1.00000	1.00000	**0.326824121**	50.00%
	2643030.09	0.88829	0.95796		46.52%
	2799988.72	0.49189	0.82282		33.45%
	2956947.35	0.28889	0.63664		32.61%
	3113905.98	0.19339	0.30030		44.65%
	3270864.60	**0.13874**	**0.10811**		**51.53%**
	3427823.23	0.08889	0.06306		51.29%
	3584781.86	0.02222	0.01201		50.51%
	3741740.49	0.01201	0.00000		50.60%
	3898699.12	0.00000	0.00000		50.00%
Min.	0.00	1.00000	1.00000	0.320127335	50.00%
	308138.33	0.59760	0.91892		33.93%
	616276.67	0.53754	0.84384		34.68%
	924415.01	0.43664	0.75676		33.99%
	1232553.35	0.29730	0.60961		34.38%
	1540691.69	0.21021	0.35135		42.94%
	1848830.03	0.12613	0.12312		50.15%
	2156968.37	**0.10511**	**0.02703**		**53.90%**
	2465106.71	0.01502	0.00601		50.45%
	2773245.05	0.00000	0.00000		50.00%

Table 3. Results for the different combination measures using Cosine Distance. The results in bold are the best for each combination rule and distance measure.

Comb.	Thres.	TPR	FPR	AUC	Acc.
Average	0.83693246	1.00	1.00	**0.884413242**	50.00%
	0.85505219	0.99	1.00		49.70%
	0.87317191	0.99	1.00		49.61%
	0.89129164	0.98	1.00		49.13%
	0.90941137	0.98	0.96		50.90%
	0.92753109	0.98	0.80		58.86%
	0.94565082	0.97	0.52		72.40%
	0.96377055	**0.94**	**0.27**		**83.51%**
	0.98189027	0.77	0.11		82.79%
	1.00001000	0.00	0.00		50.00%
Max.	**1.00000000**	**1.00**	**1.00**	0.5	**50.00%**
	1.00000111	0.00	0.00		50.00%
	1.00000222	0.00	0.00		50.00%
	1.00000333	0.00	0.00		50.00%
	1.00000444	0.00	0.00		50.00%
	1.00000556	0.00	0.00		50.00%
	1.00000667	0.00	0.00		50.00%
	1.00000778	0.00	0.00		50.00%
	1.00000889	0.00	0.00		50.00%
	1.00001000	0.00	0.00		50.00%
Min.	0.00000000	1.00	1.00	0.854307461	50.00%
	0.11111222	1.00	0.98		51.20%
	0.22222444	1.00	0.95		52.70%
	0.33333667	1.00	0.92		53.75%
	0.44444889	1.00	0.84		57.69%
	0.55556111	0.94	0.70		61.98%
	0.66667333	0.87	0.33		76.94%
	0.77778556	**0.58**	**0.04**		**77.30%**
	0.88889778	0.41	0.02		69.64%
	1.00001000	0.00	0.00		50.00%

– *False Positive Ratio* (FPR), which is the number of legitimate applications misclassified as malware.

$$FPR = \frac{FP}{(FP + TN)} \tag{8}$$

where FP is the number of valid applications incorrectly detected as malicious and TN is the number of valid applications correctly classified.

– *Accuracy*, which is the total number of hits divided by the number of instances in the dataset.

$$Accuracy = \frac{TP + TN}{(TP + FP + TN + FN)} \tag{9}$$

– *Area Under ROC Curve* [10], establishes the relation between FNR and FPR for the different thresholds stablished.

Table 1 shows the results obtained using the Manhattan distance, Table 2 shows the results for the Euclidean distance and Table 3 shows the results for the Cosine distance. In this way, both the Manhattan and Euclidean distances achieved very low accuracy results. However, the results of our anomaly-based system using the Cosine similarity are considerably sounder. In particular, it obtained a best result of 83.51% of accuracy, with an FPR of 27% and a TPR of 94%, using the average combination rule.

5 Conclusions and future work

Smartphones and tablets are flooding both consumer and business markets and, therefore, manage a large amount of information. For this reason, 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.

In this paper, we present a new malicious software detection approach that is inspired on anomaly detection systems. In contrast to other approaches, this method only needs to previously label goodware and measures the deviation of a new sample with 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 samples that exist today is assumable by existing systems, but is growing very rapidly. This approach reduces the necessity to collect malware samples and is trained using benign ones. In addition, our method is based on features that are extracted from string analysis of the application, making possible to prevent the installation of malicious software.

However, this approach also has several limitations. Through an internet connection, a benign application can download a malicious payload and change its behaviour. To detect these kind of changes, a dynamic approach is necessary to monitor the behaviour of the applications. Nevertheless, these approaches require considerable computational effort.

Future work is oriented in three main directions. First, there are other features that could be used to improve the detection ratio. These features could be obtained from the `AndroidManifest.xml` file. Second, other distance measurements and combination rules could be tested. Finally, the effectiveness of the method relies on the appropriate choice of the thresholds, making necessary to improve these metrics.

References

1. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, ACM (2011) 15–26
2. Blasing, T., Batyuk, L., Schmidt, A.D., Camtepe, S.A., Albayrak, S.: An android application sandbox system for suspicious software detection. In: Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on, IEEE (2010) 55–62
3. Shabtai, A., Elovici, Y.: Applying behavioral detection on android-based devices. Mobile Wireless Middleware, Operating Systems, and Applications (2010) 235–249
4. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P., Alvarez, G.: Puma: Permission usage to detect malware in android. In: Proceedings of the 5^{th} International Conference on Computational Intelligence in Security for Information Systems (CISIS). (2012)
5. Santos, I., Penya, Y., Devesa, J., Bringas, P.: N-Grams-based file signatures for malware detection. In: Proceedings of the 11^{th} International Conference on Enterprise Information Systems (ICEIS), Volume AIDSS. (2009) 317–320
6. Santos, I., Devesa, J., Brezo, F., Nieves, J., Bringas, P.: Opem: A static-dynamic approach for machine-learning-based malware detection. In: International Joint Conference CISIS12-ICEUTE12-SOCO12 Special Sessions. Herrero, .; Snel, V.; Abraham, A.; Zelinka, I.; Baruque, B.; Quintin, H.; Calvo, J.L.; Sedano, J.; Corchado, E. (Eds.). Advances in Intelligent Systems and Computing. Volume 189. (2012) 97–108
7. Baeza-Yates, R.A., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
8. Salton, G., McGill, M.: Introduction to modern information retrieval. McGraw-Hill New York (1983)
9. Tata, S., Patel, J.M.: Estimating the selectivity of tf-idf based cosine similarity predicates. ACM SIGMOD Record **36**(2) (2007) 7–12
10. 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 **35**(2) (2009) 183–193