Mobile Malware Detection in the Real World

Francesco Mercaldo[1], Corrado Aaron Visaggio[1], Gerardo Canfora[1], Aniello Cimitile[1]
[1]Department of Engineering, University of Sannio, Italy
{fmercaldo, visaggio, canfora, cimitile}@unisannio.it

ABSTRACT

Several works in literature address the mobile malware detection problem by classifying features obtained from real world application and using well-known machine-learning techniques. Several authors have published empirical studies aimed at assessing the quality of set of features. In this paper we propose BehaveYourself!, an Android application able to discriminate a trusted application by a malicious one extracting opcode-based features. Our application is open and flexible: it can be used as a starting point to define, and experiment with, additional features. We release BehaveYourself! to the research community at the following url: `http://www.ing.unisannio.it/cimitile/BehaveYourself.apk`

1. INTRODUCTION AND BACKGROUND

Smartphones are becoming more and more pervasive in everyday activities [1–4]. Current solutions to protect users from new threats in mobile platform are still inadequate [5–7]: the main problem is that a threat must be widespread for being successfully recognized.

The literature provides several empirical studies to detect Android malware [8,9]: results are encouraging, but authors do not deploy their solutions on a real-world mobile environment. In this way end users can not evaluate the proposed solution on their devices.

Researchers in [10] developed Andrubis, an analysis system for Android applications. They release also a mobile app able to communicate with the main server to receive the analysis results. Andrubis executes the application in a virtual environment, making the adopted solution slow in terms of responsiveness. Androguard [11] is a Python tool able to compare an application with a malware database using clustering and similarity distance. It does not provide a mobile app to check the installed applications on device against the signature database.

While the importance of the empirical studies related to mobile malware detection using machine learning techniques has been addressed by the research community, the resulting prototypes remain limited in terms of analysis capabilities and availability.

In this paper we propose BehaveYourself!, an application for Android environment able to detect the maliciousness of an application at installation time: our prototype, at the best of authors knowledge, represents the first example of Android antimalware features-based fully deployed as Android application. The effectiveness of the features involved are evaluated in two empirical studies [9, 12].

BehaveYourself! collects structural code metrics that may be indicators of malware, and the obtained values are compared with metrics resulting from a baseline computed from all the samples analyzed and stored into a central repository, while antimalware does not provide structural measurements to the user, but just a boolean evaluation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

2. APPROACH

In this section we describe the BehaveYourself! approach. Our prototype analyzes op-codes [13] which are usually used to change the application's control flow, since these op-codes can be indicators of the application's complexity. The underlying assumption is that the business logic of a trusted application tends to be more complex than malware one, because the trusted application code must implement a certain set of functions. On the contrary, the malware application is required to implement just the functions that activate the malicious behaviour.

Specifically, we consider 6 op-codes: (i) *Move (M)*: which moves the content of one register into another one; (ii) *Jump (J)*: which deviates the control flow to a new instruction based on the value in a specific register; (iii) *Packed-Switch* (P): which represents a switch statement. The instruction uses an index table; (iv) *Sparse-Switch* (S): which implements a switch statement with sparse case table, the difference with the previous switch is that it uses a lookup table; (v) *Invoke* (K): which is used to invoke a method, it may accept one or more parameters; and (vi) *If* (I): which is a Jump conditioned by the verification of a truth predicate.

We computed six features as follows, a feature represents a candidate metric to detect Android malware. Let F be one of the six features extracted, let X be one of the occurrences of the six extracted op-codes (i.e., *M,J,P,S,K and I*) from the i-th class of the application.

Then we count the occurrences of each of these op-codes in each class, and compute the features as follows:

$$F_x = \frac{\sum_{i=1}^{N} X_i}{\sum_{i=1}^{N}(M_i+J_i+P_i+S_i+K_i+I_i)}$$

where: (i) at the numerator X_i denotes the occurrences

of a given op-code in a class of the app, the sum spans over all the application classes; (ii) at the denominator we sum the occurrences of all the considered op-codes over the application classes.

3. IMPLEMENTATION

The BehaveYourself! daemon starts when the application is installed and it restarts automatically when device is booted.

Basically BehaveYourself! analysis consists of five steps, as Figure 1 explains:

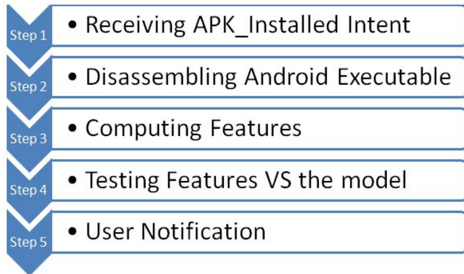

Figure 1: the BehaveYourself! evaluation process.

The application requires the declaration of the *android.-intent.action.PACKAGE_ADDED* intent filter in the *Manifest.xml* file, in order to be able to receive a broadcast (**Step 1**) when an application is installed.

The *dalvik disassembler* is able to extract from the dalvik executable file the smali classes (**Step 2**). Regarding the *dalvik disassembler* implementation, we use the smali/baksmali [14] library. This library is an assembler/ disassembler for the dex format used by dalvik, the Android's Java VM implementation.

In order to make available the smali/baksmali library in Android environment must be recompiled the Java VM version of the library. To convert Java class files into a .dex (Dalvik Executable) file we invoke the *dx* [15] compiler, a tool available from the Android SDK [16]. We recompiled the library for the Android platform using the *dx* [15] compiler.

Using the following command: *dx –dex –output=<output-file> <input-file>* we build the .jar file of smali/baksmali library (i.e. *<input-file>*) into the corresponding .dex file (i.e., *<output-file>*) optimized for running on Dalvik VM.

The obtained decompiler is invoked using the *dalvikvm -classpath dexPath/bak.dex org.jf.baksmali. main /app.apk -o /outDir* command, where *-classpath dexPath/bak.dex* identifies the path where the decompiler is stored, *org.jf.baksmali. main* represents the main class of decompiler, *app.apk* is the application under testing, and finally with *-o outDir* it is identified the folder to store the disassembled *smali* files.

The decompiler is downloaded at run-time from the BehaveYourself! *asset* folder.

The *features extractor* module is able to compute the feature values. It is based on the mobile porting of Lucene [17], an information retrieval library (**Step 3**). We use the class *NGramTokenizer* to extract the N-gram with N equal to 1, i.e. the opcode occurrency.

The *testing module* is able to check if the computed features are coherent with respect to learned model (**Step 4**). The feature testing is accomplished using the mobile porting of Weka library [18], a popular suite of machine learning software Java-based. The model is built with the J48 algorithm using the 10-fold cross validation.

Finally, to notify the user (**Step 5**) about the maliciousness or the trustiness of the analysed application we use the *NotificationManager* provided by *android.app* Android package.

From the application interface it is possible to visualize details about the feature occurrences on the last samples analyzed (Fig. 2).

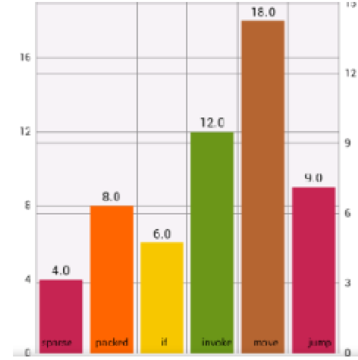

Figure 2: Features histogram resulting from an analyzed application.

BehaveYourself! does not require the device root, but it requires to set the *chmod 777* permission to *dalvik-cache* folder.

We tested BehaveYourself! on the Android emulator (4.1.2 version), using the Galaxy Nexus configuration on a machine equipped with Linux Mint 15.

4. EVALUATION

The evaluation dataset includes 5,560 Android trusted applications and 5,560 Android malware applications, the trusted samples were retrieved from Google Play [19], while the malicious ones from the Drebin Project [20, 21].

The features produced a precision equal to *0.949* in malware identification, where the precision is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved. We also test malware dataset using Androguard and Andrubis tools, obtaining respectively a precision equal to *0.22* and *0.98* [22]. We recall here that Androguard performs a static analysis, while Andrubis analyses at runtime the samples, i.e. performs a dynamic analysis. Androguard does not provide an interface for mobile devices, while Andrubis can not be implemented as real time antimalware on mobile device because it performs a dynamic analysis on a sandbox. Results are coherent with ones obtained in [9, 12, 22].

5. CONCLUSIONS

As previous empirical studies demonstrate [9, 12] mobile malware detection using op-code frequency features could be a successful solution to recognize malware.

In this paper we propose BehaveYourself!, an Android antimalware able to catch an application at installation time which uses op-codes frequency as discriminating factor between trusted and malware applications.

6. REFERENCES

[1] "The global adoption and diffusion of mobile phones." http://pirp.harvard.edu/pubs_pdf/kalba/kalba-p08-1.pdf, last visit 13 December 2015.

[2] M. L. Bernardi and M. Cimitile, "Model driven development of cross-platform mobile applications," in *The 11th IASTED International Conference on Software Engineering (SE 2012)*, 2012.

[3] M. Cimitile, M. Risi, and G. Tortora, "Automatic generation of multi platform web map mobile applications.," in *DMS*, pp. 84–89, 2011.

[4] H. Jaakkola, M. Gabbouj, and Y. Neuvo, "Fundamentals of technology diffusion and mobile phone case study," *Circuits, Systems and Signal Processing*, vol. 17, no. 3, pp. 421–448, 1998.

[5] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pp. 329–334, ACM, 2013.

[6] K. Sharma, T. Dand, T. Oh, and W. Stackpole, "Malware analysis for android operating," in *8th Annual Symposium on Information Assurance (ASIAâĂŹ13)*, vol. 31, 2013.

[7] G. Canfora, A. Di Sorbo, F. Mercaldo, and C. A. Visaggio, "Obfuscation techniques against signature-based detection: a case study," in *Proceedings of 1st Workshop on Mobile System Technologies (MST), May 22, 2015, Milano, Italy*, p. To appear, May 2015.

[8] G. Canfora, F. Mercaldo, and C. A. Visaggio, "A classifier of malicious android applications," in *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pp. 607–614, IEEE, 2013.

[9] G. Canfora, F. Mercaldo, and C. A. Visaggio, "Mobile malware detection using op-code frequency histograms," *International Conference on Security, and Cryptography (SECRYPT)*, 2015.

[10] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer, "Andrubis-1,000,000 apps later: A view on current android malware behaviors," in *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[11] "Androguard." https://github.com/androguard/androguard, last visit 3 December 2015.

[12] G. Canfora, F. Mercaldo, and C. A. Visaggio, "Evaluating op-code frequency histograms in malware and third-party mobile applications," *Lecture Notes in Computer Science, Springer*, 2015.

[13] "Dalvik opcodes." http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html.

[14] "smali/baksmali." https://github.com/JesusFreke/smali, last visit 3 December 2015.

[15] "Tools help android developers." http://developer.android.com/tools/help/index.html, last visit 3 December 2015.

[16] "Installing the android sdk." http://developer.android.com/sdk/installing/index.html, last visit 3 December 2015.

[17] "Apache lucene." https://lucene.apache.org/core/, last visit 3 December 2015.

[18] "Weka 3." http://www.cs.waikato.ac.nz/ml/weka/, last visit 3 December 2015.

[19] "Google play." https://play.google.com/store, last visit 3 December 2015.

[20] M. Spreitzenbarth, F. Echtler, T. Schrek, F. C. Freiling, and J. Hoffman, "Mobilesandbox: looking deeper into android applications," in *Proc. the 28th ACM Symposium on Applied Computing (SAC)*, 2013.

[21] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Efficient and explainable detection of android malware in your pocket," in *Proc. of 17th Network and Distributed System Security Symposium, NDSS*, vol. 14.

[22] F. Mercaldo, C. A. Visagggio, A. Oropallo, and P. Pirone, "Evaluating the commercial and research antimalware tools against malware in the wild and third-party markets: A technical report." https://www.researchgate.net/publication/275334543_Evaluating_the_commercial_and_research_antimalware_tools_against_malware_in_the_wild_and_third-party_markets_A_technical_report, last visit 9 January 2016.