Android Antivirus Scanner by Analyzing Operation Codes

Komal Narang[a], Kingkarn Sookhanaphibarn[a*], Prasong Praneetpolgrang[b]

[a]Multimedia Intelligent Technology Laboratory, School of Science and Technology, Bangkok University, Thailand

[b]School of Information Technology, Sripatum University, Bangkok 10900, Thailand

komal.n@bu.ac.th, kingkarn.s@bu.ac.th, prasong.pr@spu.ac.th

Keywords: Feature extraction, Term Frequency, Malicious code detection, Support Vector Machine

Abstract. This research presents a model for malware detection on mobile operating system based on analyzing the operation codes. The research processes are as follows: (1) collecting of both malicious and benign codes on android operating system, (2) extracting features based on the distribution of n-grams frequency where the parameter n = 3 is used, and (3) constructing a model for classification the malicious codes using the extracted features for both malicious and benign codes. In the experiment, 304 malicious codes and 553 benign codes were using to construct the model. The experiment shows that the model achieved more than 85.52% accuracy. For the sensitivity and specificity, the model achieved 71.26% and 90.52%, respectively.

Introduction

As the mobile computing devices such as smart phones and tablets are becoming more powerful every day, their demand and popularity is also rising. These powerful devices become substitute of comparatively larger devices such as laptops and even desktop computers. According to Gartner (www.gartner.com), an American IT research and advisory firm, there will be more than one billion android users by the end of 2014. Unfortunately, the rising popularity of android enabled devices has attracted malware programmers.

The numbers of malware applications in android are rapidly growing. Everyday these applications are coming up with new tactics to avoid detection from antivirus applications and breach the security. Android applications are written using Java language. With the support of numerous classes from Java library it is easier to write complex applications for android devices. An application written in high level language like Java contains a large number of library codes. Since, malware applications and benign application both uses the same Java library it is really difficult to analyze and differentiate these applications.

Recent Works on Antivirus Approaches: There were several works on antivirus by analyzing the operation codes but they focused on personal computers. Moskovitch et al. [1] introduced the important feature for detecting the malicious codes as follows: program header, reserved words, and the order of bytes. Abou-Assaleh et al. [2] proposed the use of common N-Gram (CNG) to create the profile of benign codes and the profile of malicious codes, after that an unknown code will be compared with the created profiles. Kotler and Maloof [3] tried to find the best classifier for detecting the malicious codes among *k*-Nearest Neighbors, Naïve Bayes, Support Vector Machine (SVM), and Decision Tree (J48) , and they found that SVM yield the best accuracy.

Android Applications: Android application package file (APK) is basically a zip file based on "Jar" format, similar to MSI in windows, used for distributing application software and middleware by Google for android devices. An APK package consists following files:

1) AndroidManifest.xml
2) Classes.dex
3) Res directory

First, every APK package contains a "AndroidManifest.xml" file which includes name, version, access permissions and other information for the application. Second, a "Classes.dex" file contains compiled Java source code. Android operating system uses Dalvik virtual machine which is not compatible with Java virtual machine. Therefore, to obtain Dalvik equivalent bytecode, Java source code is first compiled using "javac" (Java compiler) and then Java byte code is further compiled by Android dx compiler (as illustrated in Fig. 1) [4]. Res directory is a directory containing external resources like images, video, audio files, etc. There can be some additional files in the APK package which are optional such as external libraries or other sources.

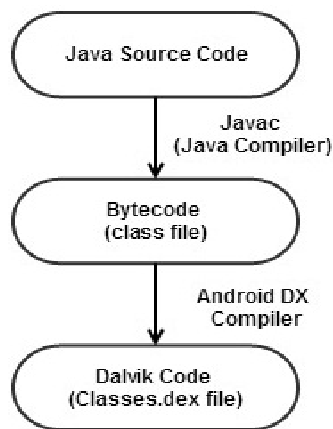

Fig. 1: Android complier

Our Methodology

Extracting Operation Codes (Opcodes) from APK package. As mentioned earlier that every APK package contains a classes.dex file which contains all the executable codes. Since APK package is basically a compressed zip file which can be accessed programmatically or using any zip archive application. In our approach we have extracted classes.dex files programmatically. After extracting dex files it is disassembled using "Dexdump" tool which is a part of Android Development Tool (ADT) bundle. Dexdump converts dex files in human readable format. The output of Dexdump contains meta-information and dalvik equivalent bytecode instructions for all the classes. Following (as shown in Fig. 2) is a sample output of Dexdump.

```
000418: 2b02 0c00 0000                         |0000: packed-switch v2, 0000000c // +0000000c
00041e: 12f0                                   |0003: const/4 v0, #int -1 // #ff
000420: 0f00                                   |0004: return v0
000422: 1220                                   |0005: const/4 v0, #int 2 // #2
000424: 28fe                                   |0006: goto 0004 // -0002
000426: 1250                                   |0007: const/4 v0, #int 5 // #5
000428: 28fc                                   |0008: goto 0004 // -0004
00042a: 1260                                   |0009: const/4 v0, #int 6 // #6
00042c: 28fa                                   |000a: goto 0004 // -0006
00042e: 0000                                   |000b: nop // spacer
000430: 0001 0300 faff ffff 0500 0000 0700 ... |000c: packed-switch-data (10 units)
```

Fig. 2: Sample output of Dexdump

The output of the dexdump contains dalvik bytecodes and meta-info which cannot be used to prepare 3-gram instructions. Hence it needed to be filtered for bytecodes and further these bytecodes are converted into dalvik equivalent opcode. To achieve this, a tool called "Byte2OpCode" was written (as shown in Fig. 3). Byte2OpCode fetches all the instructions and converts them in opcodes. All these opcodes are stored in "oc" file.

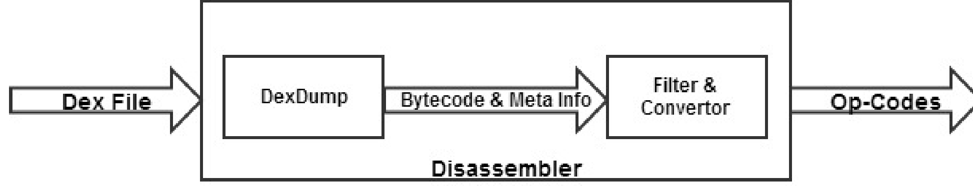

Fig. 3: Disassembling process of Byte2OpCode Tool

The Byte2Opcode tool first calls the dexdump by supplying dex file as input and stores the output in a temporary file with .bc extension. Then "bc" file is further processed for filtering and converting opcodes and results are stored in a file with .oc extension. Since this .bc file is not required further, it will be removed by Byte2OpCode tool. Following (as shown in Fig. 4) is the final output of Byte2OpCode tool.

0x54	0x1a	0x71	0x70	0x28	0x15	0x70	0x0e
0x71	0x0a	0x71	0x11	0x71	0x71	0x0a	0x38
0x12	0x11	0x71	0x28	0x12	0x3b	0x11	0x71
0x28	0x0d	0x1a	0x22	0x70	0x1a	0x6e	0x6e
0x1a	0x6e	0x6e	0x71	0x28	0x15	0x6e	0x1f
0x14	0x6e	0x1f	0x1a	0x71	0x1a	0x71	0x6e

Fig. 4: Sample output of Byte2OpCode tool

Analyzing the opcodes. We applied the n-gram model by using n=3 as reported in [2] that 3-gram is the optimal combination for analyzing the opcodes. The frequency of 3-gram will be considered as the feature of an application (or a program). We will use a window size of 3 words to compute the frequency of 3-gram sequence by sliding the window from the beginning of the file to the end. For example, the file of 48 words will be extracted of 46 sequences (3-gram) as shown in Fig. 4.

Basically, the frequency of a 3-gram sequence is the number of occurance in the file. It will be called "absolute frequency". However, this frequence value depend on the size of a file. We then normalize it by the mean where the mean is the average of 3-gram sequence in all files. We called a new frequency "relative frequency". Its calcuation is shown in Eq. (1). Let $f_{x_i}^{d_j}$ be the frequency of 3-gram sequence x_i in file d_j. On the other hand, $f_{x_i}^{d_j}$ is the absolute frequency. Suppose that there are m files in total, and the set of m files is $D = \{d_1, d_2, d_3, \dots, d_m\}$.

$$f'^{d_j}_{x_i} = \frac{f_{x_i}^{d_j}}{\mu} \tag{1}$$

$$\text{where } \mu = \frac{\sum_{d_j \in D} f_{x_i}^{d_j}}{m}$$

Classifying the malicious codes. A set of the relative frequency of 3-gram sequences is up to 20 thousand sequences. As a result, we applied the Principal Component Analysis (PCA) [5] to reduce the data dimension before classified by SVM [6].

Experiments and Results

Data set. We collected two groups of files: 553 benign files and 304 virus files.

1) Benign files: The prudent applications which are reliable and safe for android devices are classified as benign applications. All the system files are benign application because these applications come as factory default.
2) Virus files: These detrimental applications are capable of performing some harmful operations in android device. Some of these applications are capable of stealing confidential information stored in android device and can send it to remote server without the knowledge of user such as Android.opFake which sends the messages on premium rate without the consent of the user.

Experimental procedure. We considered the performance of our antivirus model against a set of unknown files. Thus the performance indicators are accuracy, sensitivity, and specificity [6]. The data set is divided into a training set and a testing set which the number of files is 90% for training and 10% for testing. For generalization, we use the cross-validation of k=10 [7] to run our model against the data set. For the use of PCA, we have to find the optimal number of component to yield the best accuracy; then, we run the model with the number of components between 2 to 857.

Experimental result. The experiment shows that the model when used of 200 components has accurate detection of 85.52%. The sensitivity and specificity of the model are 71.26% and 90.52%, respectively.

Conclusion and Future Work

We developed a new model of malicious-code detection by analyzing opcodes on mobile devices. The merit of our work is to reduce the user burden in updating the virus signatures. In the experiment, our model can tolerate with unknown viruses of 85.52% where the sensitivity of virus detection is up to 90.52% but the specificity is about 71.26%. On the other hand, the proposed antivirus program will have a false alarm less than 10%. For the implementation, our approach will be built into two-tier architecture, where an android device sends the dex file to the server, and then the server performs the scan and sends the results back to android device. Whenever server scans a file the result and hash code of the file is stored in a database. Maintaining such database provides benefits like reduced overhead on server and better performance. Distributing this database also enables android device to perform a quick offline scan for known viruses.

Acknowledgements

This work is supported by National Research Council of Thailand (NRCT) in 2013.

References

[1] Moskovitch, Robert, Yuval Elovici, and Lior Rokach. "Detection of unknown computer worms based on behavioral classification of the host." Computational Statistics & Data Analysis 52, no. 9 (2008): 4544-4566.

[2] Abou-Assaleh, Tony, Nick Cercone, Vlado Keselj, and Ray Sweidan. "N-gram-based detection of new malicious code." In Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International, vol. 2, pp. 41-42. IEEE, 2004.

[3] Kolter, Jeremy Z., and Marcus A. Maloof. "Learning to detect malicious executables in the wild." Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2004.

[4] Nolan, Godfrey. Decompiling Android. Apress, 2012.

[5] Jolliffe, Ian. Principal component analysis. John Wiley & Sons, Ltd, 2005.

[6] Maimon, Oded Z., and Lior Rokach, eds. Data mining and knowledge discovery handbook. Vol. 1. New York: Springer, 2005.

[7] Kohavi, Ron. "A study of cross-validation and bootstrap for accuracy estimation and model selection." In IJCAI, vol. 14, no. 2, pp. 1137-1145. 1995.