Static Analysis for Android Malware Detection

Shubhranshi Kapoor

IS Lab, NTHU

Abstract

Static analysis relies on features extracted without executing code, while dynamic analysis extracts features based on code execution (or emulation). In general, static analysis is more efficient, while static analysis is often more informative, particularly in cases of highly obfuscated code. Static analysis of an Android application can rely on features extracted from the manifest file or the Java bytecode, while dynamic analysis of Android applications can deal with features involving dynamic code loading and system calls that are collected while the application is running. In this experiment, we analyzed the effectiveness of static features for detecting Android malware using machine learning techniques. We also carefully analyze the robustness of our scoring technique.

1. Introduction

Smartphone malware can come in the form of Trojan, botnet or spyware. Such applications are created with malicious intent, and can, for example, acquire a user’s private data [23]. Today, the majority of smartphones are based on the Android Operating System (OS). According to a recent report by International Data Corporation, Android dominates the smartphones market, with a market share of 88.2% in 2015 [24]. As announced at a press event by Google, there are approximately 1.4 billion active Android phone users. The large market for smartphones has drawn the attention of cybercriminals [25].

Android has various third party application stores which makes it easy for cybercriminals to repackage Android applications with malicious payloads. Such cybercriminals develop malicious software which is often designed to gain access to information within a smartphone. Reports estimate that during 2010 to 2014, the number of mobile malware applications have grown exponentially and most of this malware has targeted Android systems. According to a report by Kaspersky Labs, there were 291,800 new mobile malware programs that emerged in the second

quarter of 2015, which is 2.8 times more than in the first quarter. In addition, there were 1 million mobile malware installation packages in the second quarter, which is 7 times more than the first quarter of 2015 [20]. Due to this alarming increase in the number of Android malware applications, the analysis and detection of Android malware has become an important research area.

To collect the features used when analyzing malware, we can rely on static or dynamic analysis (or some combination thereof). Static analysis refers to features that are collected without executing the code. In contrast, in dynamic analysis we execute (or emulate) the code. Static analysis is usually more efficient, but dynamic analysis can be more informative, and dynamic analysis is often thought to be less susceptible to code obfuscation. Static analysis of Android malware can rely on Java bytecode extracted by disassembling an application. The manifest file is also a source of information for static analysis. One disadvantage of static analysis is that it is blind to dynamic code loading, that is, static analysis fails to deal with parts of the code that are downloaded during execution. In contrast, dynamic analysis can examine all code that is actually

executed by an application.

The static features we consider are permissions extracted from the manifest file. We analyze the effectiveness of this technique with the help of robustness analysis.

2. Background

2.1 Overview of Android OS

In the Figure, the Android software stack items in green are the written in C/C++ and the blue ones are written in Java which executed using the Dalvik VM [5]. Here the Android Linux Kernel is a modified Linux Kernel which includes features like wake locks (memory management for optimizing the memory consumption), Binder IPC Drivers and other features which play a key role in mobile embedded platform [21]. The libraries item plays a vital role in optimizing CPU, memory consumption as well as the audio and video codecs for the device.

Figure: Smartphone malware



Android runtime is the managed runtime that is capable to compiling Android applications during the installation time. This component comprises of Dalvik virtual machine and core libraries of Java. During an Android application compilation, the Java bytecode is converted into Dalvik bytecode (Dalvik executable code) using dx tool, which is executed on Dalvik virtual machine. The classes.dex file consists of the whole repackaged application code after removing the duplicate parts in the code. The Dalvik virtual machine is more powerful compared to the Java Virtual Machine in terms of multitasking ability. Application framework is an abstract layer to develop applications using the underlying reusable libraries and packages. Some major components of this layer are [8]:

∙ Activity Manager: This provides an interface for the users to interact with the applications

∙ Intent/Notification Manager: This acts as messaging objects to facilitate the inter-process communication with components

∙ Content Manger: They provide an interface to pass data in from process to another process

∙ Telephony Manager: This provides telephony information like the IMEI number.

Applications are built on top of the Application framework which provide an interaction between users and the device. These are distributed as Android Package files (.apk). This .apk file is a signed ZIP file which consists of the classes.dex file, external libraries and AndroidManifest.xml file describing the capabilities of the Android application. The AndroidManifest.xml file provides information about various application components. Various application components like the activities, services, intents, broadcast receivers must be declared in this xml file. This file contains a list of permissions which the appplication requires to access certain device components and the minimal API version necessary to run this application.

A system call is a mechanism with which a user application can request a service which belongs to the operating system kernel. Information flows within the multi- layered Android architecture. For instance, an Android application can request for sending a text message. The request is transformed into a request to the Telephony Manager service which is later received by the Android runtime. Here the request is transformed to a set of library calls. These library calls are then transformed into the system calls to the Android Linux kernel. An example of the system call would be sendmsg(). Similarly, after the system calls are executed, information flows back in the reverse direction.

2.2 Types of Android application malware

2.2.1 Trojan

Android malware applications which belong to this category pretend to be either as installers or SMS malware apps. The former apps trick the user to install, by designing icons or user interfaces’ of a benign installer. In reality, these apps display a service level agreement during installation which obtains permissions to users personal information like phone number and run a background process which sends SMS’s to premium rate numbers. The later kind of trojans simply have a single activity with a button, which on click sends premium rate SMS’s.

2.2.2 Spyware

This category of malicious applications intend to gain access to users private information and send it to a private server. The main purpose is to steal information like phone location, bank or credit card details, passwords, text messages, contacts, online browsing activity, etc. A more complicated malware can also trigger activities which are issued by the remote server.

2.3 Application detection techniques

2.3.1 Static Analysis

Static analysis is a technique to detect malicious behavior by analyzing the code segments. This technique is carried out without running the application in an Android emulator or device. However, this technique has a major drawback of code obfuscation and dynamic code loading. The advantages of static analysis is that the cost of computation is low, less time consuming and low resource consumption. However, code obfuscation makes the pattern matching a major drawback in detecting the malicious behavior. There are two main detection techniques for Static Analysis - Misuse Detection and Anomaly detection.

Misuse Detection:

This technique is also known as signature based detection technique. An application is detected as a malware if it matches a sequence of instructions or policies. In the research by Feng et al. [14] the authors have presented Appopscopy, a semantic language based signatures for detecting malicious Android applications. In this approach, signatures are created for each malware family. Signature matching is achieved using the inter component call graphs to decide the control flow properties. Further, the results are enhanced using the static taint analysis to decide if the data flow properties. However, in this approach, it is very complicated to define a signature that is able overcome the drawback of code obfuscation and dynamic code loading problems. In another research by Fuchs et al. [15] implement Scandroid where the security specific features are extracted along with data flow to check the with the malicious signatures. Zhou et al. [29] extracts permissions and applies heuristic filtering to detect Android application malware.

Anomaly Detection:

This technique relies machine learning algorithms to detect malicious behavior. Features extracted from known malware are used to train the model and predict a novel or unknown malware. Abah et al. [5] proposes a machine learning approach relies on K-Nearest Neighbor classifier to train the model with features such as incoming/outgoing SMS and calls, Device status and running applications/processes. In another research by Aung et al. [21] proposes a framework which relies on machine learning algorithms to for Android malware detection using features obtained from Android events and permission based to learn and classify malware and benign applications.

2.3.2 Dynamic Analysis

Dynamic analysis is a detection technique aimed at evaluating malware by executing the application in a real environment. The main advantage of this technique is it detects dynamic code loading and records the application behavior during runtime. This technique fails to determine the amount of code that is executed while running the application. There are chances that the applications can fail to execute the malicious code while recording the features. Additionally, this technique is hard to implement as compared to static analysis, due to the overhead of executing the application.

Aphonso et al. [4] has proposed a dynamic analysis technique which records the frequency of system calls and API calls to detect the malware and goodware. The main drawback of this system is that it will detect a malware only in case the application meets certain API level. Taindroid [13] is another dynamic analysis system which captures the network data for analyzing applications. In another research by the authors of Maline [12] have proposed a malware detection tool, based on tracing system calls and classify them based on machine learning algorithms.

2.4 Machine Learning Algorithms

2.4.1 Random Forest

This is an ensemble learning algorithm which classifies based on information aggregated from individual learner. This algorithm relies on the bagging approach where each classifier is built individually by working with a bootstrap sample of the input data. Normally in a decision tree algorithm, the decision is made considering all the features. However, in Random Forest Algorithm the decision is made by randomly selecting the features. This random selection, improves the scalability when there are large number of features. In addition, it reduces the interdependence between the feature attributes and makes the results is less susceptible to noise.

2.4.3 Naive Bayes

In this algorithm we assume that all the features are independent of each other. The classification is based on the calculating the maximum probability of the attributes which belong to a particular class. Let 𝑟 = (𝑟1, 𝑟2, , 𝑟𝑛) belong to class 𝐶, and 𝑃(𝐶) be the probability of the class and 𝑃𝐶|𝑟 be the probability of feature for a given class, then an application is considered as a goodware if, 𝑃(𝐶 = 𝑏𝑒𝑛𝑖𝑔𝑛|𝑅 = 𝑟) > 𝑃(𝐶 = 𝑠𝑢𝑠𝑝𝑖𝑐𝑖𝑜𝑢𝑠|𝑅 = 𝑟).

3. Methodology

This chapter describes the Malware and benign dataset used in the project. The next section details the methodology used to extract features from the dataset. The last section describes the implementation details of the approach.

3.1 Dataset

The benign dataset was created by self since there was no standard dataset available. The benign dataset .apk files was collected randomly from the Google Play Store [16] which is considered as the official market with the least possibility of malware applications. We obtained the malware dataset from the authors of Drebin [10]. This dataset mainly consists of applications obtained from various Android markets, Android websites, malware forums, security blogs and Android Malgenome Project [30]. The malware dataset is based on results acquired from Virtotal [28] service which aggregates information from different antivirus engines, website scanners and URL analyzers. Table 1 gives a brief about the dataset used for experiments.

*Table 1: Dataset Description*

*Application type Total number of applications Year of extraction*

*Malware 103 August 2010- October 2012*

*Benign 97 October 2015- November 2015*

3.2 Feature extraction

The appropriateness of extracted features determines the accuracy of the emulation results. The features are extracted in two phases described below.

3.2.1 Feature extraction using Static analysis

Android applications come in an Android package (.apk) archive. This .apk file is nothing but a zip bundle of AndroidManifest.xml, classes.dex and other resources and folders. For extracting these features we initially need to reverse engineer the .apk files. This is done using the apktool [28]. The AndroidManifest.xml file contains a lot of features that can be used for static analysis. One of the main feature is the permissions requested by the application. The AndroidManifest.xml contains the list of permissions required by the application. In order to extract these permissions, regular xml parsers cannot be used since Android has its own proprietary binary xml format. We designed a new xml parser capable of extracting permission feature from the AndroidManifest.xml file of the application.

Feature vector:

Let 𝑅 be a vector containing a set of 135 Android permissions. For every 𝑖𝑡ℎ application in the Android apps dataset, we generate a binary sequence 𝑅𝑖 = {𝑟1, 𝑟2, . . . , 𝑟𝑗} and 𝑟𝑗 = 1 if 𝑗𝑡ℎ permission exists, 0, otherwise.

The permissions identified are stored as a binary sequence of 0 or 1 in a comma separated form. This sequence typically contains comma separated permission bits which denote 1 if the corresponding permission is present or 0 if it is absent. In addition, we consider a variable 𝐶 where 𝐶 ∈ {𝑀𝑎𝑙𝑤𝑎𝑟𝑒,𝑒𝑛𝑖𝑔𝑛}. This variable 𝐶 indicates -1 for malware application and 1 for benign application. Following is an example of the permission vector for malware and benign application:

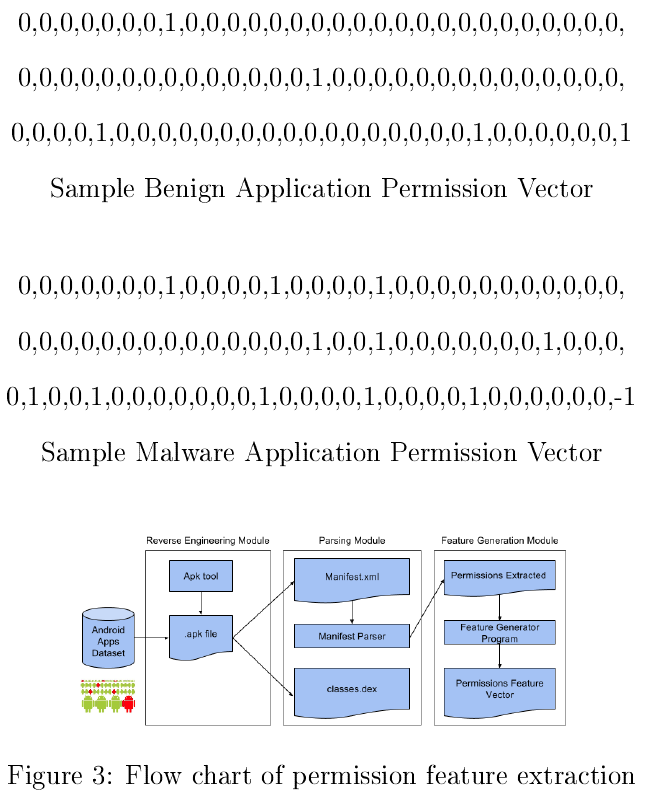


Figure 3 explains the feature extraction method. Following are the steps to extract data.

1. Create a dataset of all malware and benign files

2. Reverse engineer the android applications in the dataset. This reverse engineering is achieved using the APK tool.

3. We extract the permission request features from the AndroidManifest.xml file using our special AndroidManifest xml parser.

4. The permissions obtained for each Android application is then sent to the Feature vector generator program where the application is feature vector is generated using the method discussed above

5. We finally build a permission vector dataset for all the applications and store it in an ARFF [9] file format.

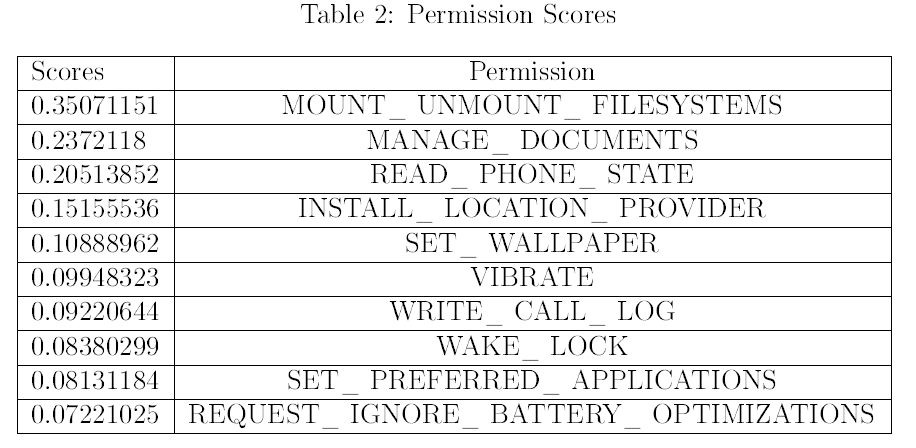
Feature selection:

For the feature vector obtained, there are many permissions which were redundant and never used in any of the Android applications. These redundant permissions are removed since they have the capacity of adverse effects for the classification process.

Thus, the main aim of feature selection is to reduce the feature set in such a way that the new set of features give similar results as the original set. For this purpose, we have used the feature selection method known as Information Gain. According to this scoring method, similarities in the pattern of permissions appearing in the Android application is calculated and then higher weights are provided to the permissions which are most effective.

The information gain of each permission is calculated by InfoGain(𝐶, 𝑟𝑗) = entropy(𝐶) − entropy(𝐶|𝑟𝑗) Here 𝑒𝑛𝑡𝑟𝑜𝑝𝑦(𝐶) is the information entropy. Also, 𝐴 and 𝐵 are random variables and 𝑃 is the probability.

Table 2 shows the list of top 10 permissions and corresponding scores using the above method. Here higher values indicate more information gained from the attribute.



After calculating the above scores using Information Gain method we started to reduce the number of 135 permission in such a way that we obtain an AUC greater than or equal to the original set. On applying the information gain algorithm, we excluded those permissions which scored 0 and obtained a subset of 99 top ranked features. We further reduced this feature set by 87 top permissions since it fetched a higher AUC than the original set.

4. Experiment

4.1 Evaluation Metrics

Accuracy of a test is evaluated on how well the test is able to distinguish between a malware and goodware. An ROC efficiently demonstrates the effectiveness of machine learning classifier by varying the threshold. This is plotted considering as a sensitivity or True positive rate (TPR) versus specificity aslo known as False positive rate (FPR). The color represents the threshold value for a each pair of true positive rate and false positive rate. If a particular instance highly belongs to the class, its threshold will be closer to 1. Hence, for a higher threshold of instance, darker will be the color in the ROC. The Area under the Curve (AUC) is the percentage of correct test results in while classifying the testing data. AUC value of 1 represents a perfect test whereas the one with 0.5 represents the least accurate test [18].

4.2 Discussion of Experiment results

4.2.1 Machine Learning Algorithm Analysis

In order to decide with the machine learning algorithm to be used for our analysis we carried out this experiment. Figure 5 shows the AUC values of different algorithms analyzed with individual system calls and permission bits. We have considered the feature vector obtained before using the feature selection algorithm. In this experiment we train the model with feature vector obtained from Section 3.2 of the paper. The main aim of this experiment is to compare the results obtained from several algorithms with Random Forest to verify the selection of correct machine learning algorithm. From the experiment results we can see that the Random Forest Algorithm with 100 trees, has given an AUC value closer to 1.0. When trained with static data we saw average accuracy of the results.

4.2.2 Permission based data analysis

We analyze the results obtained on static analysis. We use the scoring technique from Section 3.2.1.2 for permission based analysis and found that the 87 highest ranked features have produced an AUC of 0.972. The Random Forest Algorithm has given the AUC value closer to 1.0. When trained with static analysis data we saw an increase in the accuracy of the results.

References

[1] 2011 Mobile Threats Report,Retrieved March 23, 2016, from https:

//www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2011-

mobile-threats-report.pdf

[2] 2015 Mobile Threat Report, Retrieved March 23, 2016, from https://www.

pulsesecure.net/lp/mobile-threat-report-2014/

[3] J. Abah, O. Waziri,M. Abdullahi, U. Arthur and O. Adewale, A machine learning

approach to anomaly-based detection on Android platforms,International

Journal of Network Security and Its Applications, 7(6):15–35,2015

[4] V. M. Afonso, M. F. de Amorim, A. R. A. Gregio, G. B. Junquera and P. L. de

Geus, Identifying Android malware using dynamically obtained features,Journal

of Computer Virology and Hacking Techniques, 11(1):9–17, 2015

[5] Analysis of Android Architecture, Retrieved October 11, 2015, from

https://os.itec.kit.edu/downloads/sa\_2010\_braehler-stefan\_androidarchitecture.

pdf

[6] Android Debug Bridge, Retrieved October 12, 2015, from http://developer.

android.com/tools/help/adb.html

[7] Android Studio Overview, Retrieved October 12, 2015, from http://developer.

android.com/tools/studio/index.html

[8] App Manifest, Retrieved October 19, 2015, from http://developer.android.

com/guide/topics/manifest/manifest-intro.html

[9] ARFF(book version), Retrieved November 16, 2015, from https://weka.

wikispaces.com/ARFF+%28book+version%29

[10] D. Arp, M. Spreitzenbarth, M. Huebner, H. Gascon and K. Rieck, Drebin: Efficient

and Explainable Detection of Android Malware in Your Pocket,21th Annual

Network and Distributed System Security Symposium (NDSS), February 2014

[11] I. Burguera, U. Zurutuza and S. Nadjm-Tehrani,Proceedings of the 1st ACM

workshop on Security and privacy in smartphones and mobile devices , 15–26,

2011

[12] M. Dimjasevic, S. Atzeni,I. Ugrina and Z. Rakamari, Evaluation of Android

Malware Detection Based on System Calls, Retrieved December 11, 2015, from

<http://www.cs.utah.edu/docs/techreports/2015/pdf/UUCS-15-003.pdf>

[13] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. P. Cox, J. Jung, P. Mc-

Daniel and A. N. Sheth, TaintDroid: An Information-Flow Tracking System for

Realtime Privacy Monitoring on Smartphones,ACM Transactions on Computer

Systems, 32(2):1–29, 2014

[14] Y. Feng, S. Anand, I. Dillig and A. Aiken, Apposcopy: semantics-based detection

of Android malware through static analysis,Proceedings of the 22nd ACM SIGSOFT

International Symposium on Foundations of Software Engineering ,576–

587,2014

[15] A. P. Fuchs, A. Chaudhuri and J. S. Foster, SCanDroid: Automated Security

Certification of Android Applications, Technical Report CSTR-4991, Department

of Computer Science, University of Maryland,College Park, November 2009

[16] Google Play, Retrieved September 8, 2015, from https://play.google.com/

store?hl=en

[17] J. B. Guptil, Examining application components to reveal android malware, Retrieved

on March 12,2016 from, https://www.google.com/webhp?sourceid=

chrome-instant&ion=1&espv=2&ie=UTF-8#

[18] D. J. Hand, R. J. Till, A Simple Generalisation of the Area Under the ROC

Curve for Multiple Class Classification Problems, 45(2):171-186, 2001

[19] Juniper networks Mobile Threat Center Third Annual Mobile Threats Report:

March 2012 through March 2013, Retrieved March 23, 2016, from

http://www.juniper.net/us/en/local/pdf/additional-resources/3rdjnpr-

mobile-threats-report-exec-summary.pdf

[20] Kaspersky Lab Reporting: Mobile malware has grown almost 3-fold in Q2,

and cyberespionage attacks target SMB companies, Retrieved March 23, 2016,

from http://www.kaspersky.com/about/news/virus/2015/Kaspersky-

Lab-Reporting-Mobile-malware-has-grown-almost-3-fold-in-Q2-andcyberespionage-

attacks-target-SMB-companies

[21] Linux Kernel, Retrieved September 16, 2016, from https://source.android.

com/devices/#Linuxkernel

[22] Mobile Security, Retrieved March 23, 2016, from http://home.mcafee.com/

advicecenter/?id=ad\_ms\_wimm&ctst=1

[23] M. M. Saudi, F. Ridzuan, N. Basir, N. F. Nabila, S. A. Pitchay and I. N. Ahmad,

Android Mobile Malware Surveillance Exploitation Via Call Logs: Proof of Concept,

17th UKSIM-AMSS International Conference on Modelling and Simulation ,

176–181, 2015

[24] Smartphone OS Market Share, 2015 Q2, Retrieved March 23, 2016, from http:

//www.idc.com/prodserv/smartphone-os-market-share.jsp

[25] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp and J. Hoffmann, Mobile-

Sandbox: combining static and dynamic analysis with machine-learning techniques,

International Journal of Information Security,14(2):141–153,2014

[26] strace(1) - Linux man page, Retrieved December 11, 2015, from http://linux.

die.net/man/1/strace

[27] UI/Application Exerciser Monkey, Retrieved January 10, 2016, from http://

developer.android.com/tools/help/monkey.html

[28] Virustotal, Retrieved October 23, 2015, from

https://www.virustotal.com

[29] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang and P. Ning, Detecting malicious

apps in official and alternative Android markets, Proceedings of the second ACM

conference on Data and Application Security and Privacy,2012

[30] Y. Zhou and X. Jiang, Dissecting Android Malware: Characterization and Evolution,

Proceedings of the 33rd IEEE Symposium on Security and Privacy, 95–109,

2012