ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/314521542

Static and Dynamic Analysis of Android Malware

Conference Paper · January 2017

DOI: 10.5220/0006256706530662

CITATIONS
14

READS
515

3 authors, including:

Fabio Di Troia
San Jose State University
31 PUBLICATIONS **138** CITATIONS
SEE PROFILE

Mark Stamp
San Jose State University
152 PUBLICATIONS **2,083** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Secure Implementation of Modular Arithmetic Operations and hash search organization for IoT and Cloud Applications View project

Vigenère scores for malware detection View project

All content following this page was uploaded by Mark Stamp on 04 November 2018.

The user has requested enhancement of the downloaded file.

Static and Dynamic Analysis of Android Malware

Ankita Kapratwar, Fabio Di Troia and Mark Stamp
Department of Computer Science, San Jose State University, San Jose, U.S.A.

Keywords: Malware, Android, Static Analysis, Dynamic Analysis.

Abstract: Static analysis relies on features extracted without executing code, while dynamic analysis extracts features based on execution (or emulation). In general, static analysis is more efficient, while dynamic analysis can be more informative, particularly in cases where the code is obfuscated. Static analysis of an Android application can, for example, rely on features extracted from the manifest file or the Java bytecode, while dynamic analysis of such applications might deal with features involving dynamic code loading and system calls. In this research, we apply machine learning techniques to analyze the relative effectiveness of particular static and dynamic features for detecting Android malware. We also carefully analyze the robustness of the scoring techniques under consideration.

1 INTRODUCTION

According to a recent report by International Data Corporation, Android dominates the smartphone market, with a market share of 88.2% as of 2015 and more than 1.4 billion active Android phone users[1]. This large market for smartphones has not gone unnoticed by cybercriminals (Spreitzenbarth, 2014). There are many third party stores for Android applications, and it has become common for cybercriminals to repackage legitimate Android applications to include malicious payloads. Smartphone malware can come in many forms, including Trojans, botnets, and spyware. Such applications are created with malicious intent, and can, for example, acquire a user's private data (Saudi, 2015).

Reports estimate that during the 2010 to 2014 timeframe, the number of mobile malware applications grew exponentially, and most of this malware targeted Android systems. Figure 1 shows the increase in the number of total mobile malware applications and the share of these that are Android malware[2,3]. According to a report by Kaspersky Labs, there were 291,800 new mobile malware programs that emerged in the second quarter of 2015 alone, which is 2.8 times more than in the first quarter. In addition, there were one million mobile malware installation packages in the second quarter, which is seven times greater than the number in the first quarter[4].

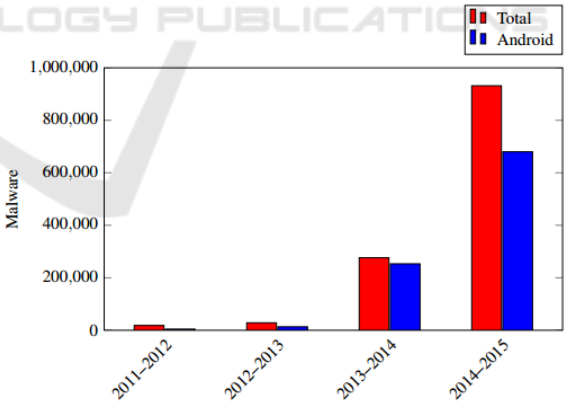


Figure 1: Growth of Mobile Malware.

Due to this alarming increase in the number of Android malware applications, the analysis and

[1] http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] https://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2011-mobile- threats-report.pdf

[3] http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[4] http://www.kaspersky.com/about/news/virus/2015/Kaspersky-Lab-Reporting-Mobilemalware-has-grown-almost-3-fold-in-Q2-andcyberespionage-attacks-target-SMB-companies

Kapratwar, A., Troia, F. and Stamp, M.
Static and Dynamic Analysis of Android Malware.
DOI: 10.5220/0006256706530662
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 653-662
ISBN: 978-989-758-209-7
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

detection of Android malware has become an important research topic. Many Android malware detection and classification techniques have been proposed and analyzed in the literature, some of which we briefly review later in this paper.

To collect the features used to analyze malware, we can rely on static or dynamic analysis—or some combination thereof. Static analysis relies on features that are collected without executing the code. In contrast, for dynamic analysis we execute (or emulate) the code. Static analysis is usually more efficient, since no code execution is required. Dynamic analysis can be more informative, since we only analyze code that actually executes. However, with dynamic analysis we may not not see all execution paths, which can limit our overall view of the code.

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 specific disadvantage of such 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.

In this paper, we consider Android malware detection based on static and dynamic features. The static features we consider are based on permissions extracted from the manifest file, while our dynamic analysis is based on system calls extracted at runtime. We analyze the effectiveness of these techniques individually and in combination. We also perform a robustness analysis, and carefully consider the interplay between the static and dynamic features.

This paper is organized as follows. In Section 2, we discuss relevant background topics, including a brief overview of the Android operating system, a brief literature survey, and a high level view of the machine learning techniques used in this research. Section 3 discusses the dataset used and our methodology for extracting static and dynamic features. Section 4 provides our experimental results. Finally, in Section 5 we give our conclusion and suggestions for the future work.

2 BACKGROUND

In this section, we discuss relevant background topics. Our focus here is on previous related work, while we also give an overview of the Android OS, we take a brief look at different types of Android malware from a high-level perspective, and we discuss the various machine learning techniques that are used in our analysis.

2.1 Overview of Android OS

Figure 2 illustrates the Android software stack, where the items in green are the written in C/C++ while the blue items are written in Java and executed using the Dalvik VM[5]. The Android Linux Kernel is a modified Linux Kernel which includes wake locks, binder IPC drivers, and other features that play a critical role in a mobile embedded platform[6]. The libraries plays a role in optimizing CPU usage, memory consumption, and also contains the audio and video codecs for the device.

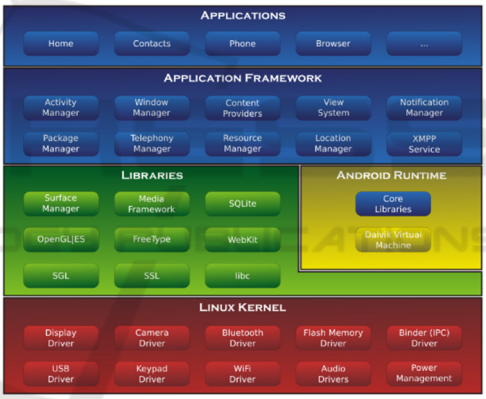


Figure 2: Android architecture (Abah, 2015).

The Android runtime layer consists of the Dalvik virtual machine and core Java libraries. During an Android application compilation, the Java bytecode is converted into Dalvik bytecode using dx tool, which is executed on the Dalvik virtual machine. The Dalvik virtual machine is more powerful than the Java Virtual Machine in terms of multitasking capabilities.

The application framework is an abstract layer used to develop applications that rely on the underlying reusable libraries and packages. Some major components of this layer include the following[7].

[5] https://os.itec.kit.edu/downloads/sa_2010_braehler-stefan_android-architecture.pdf

[6] https://source.android.com/devices/#Linuxkernel

[7] http://developer.android.com/guide/topics/manifest/manifest-intro.html

- The Activity Manager provides an interface for the users to interact with the applications.
- The Intent/Notification Manager deals with messaging objects to facilitate interprocess communication with components.
- The Content Manager provides an interface to connect data in one process with code running in another process.
- The Telephony Manager deals with telephony related information, such as the International Mobile Station Equipment Identity (IMEI) number.

Applications are built on top of the Application framework, which provides for interaction between users and the device. Applications are distributed as android package (apk) files. An apk file is a signed zip archive file that includes a classes.dex file, external libraries, and the AndroidManifest.xml. This manifest file describes the abilities or privileges granted to the application, and also provides information about various application components. For example, the activities, services, intents, and broadcast receivers must be declared in this xml file. For our purposes, the most important aspect of the manifest is that it contains a list of permissions, which allows the application to access certain device components. These permissions are explicitly granted by the user at install time.

2.2 Android Malware

Android malware applications primarily consist of Trojans. A typical Android Trojans might trick the user by using icons or user interfaces that mimic a benign application. Android Trojans often display a service level agreement during installation which obtains permissions to access a user's personal information, such as the phone number. The Trojan can then, for example, send SMSs to premium rate numbers in the background.

Android Trojans are also often used as spyware. Such malicious applications can gain access to a user's private information and send it to a private server. The main purpose of such spyware is to steal information such as phone location, bank or credit card details, passwords, text messages, contacts, on-line browsing activity, and so on. A more sophisticated implementation might also include botnet capabilities.

2.3 Related Work

In the research by Feng, et al. (Feng, 2014), the authors develop Appopscopy, a semantic language-based signature detection strategy for Android. In this approach, general signatures are created for each malware family. Signature matching is achieved using inter-component call graphs based on control flow properties. Further, the results are enhanced using static taint analysis. However, this approach seems to be fairly weak with respect to code obfuscation and dynamic code loading.

In the research by Fuchs, el al. (Fuchs, 2009), the authors analyze a tool that they call Scandroid. This scheme extracts features based on data flow. Zhou, et al. (Zhou, 2012a), analyze permissions and apply heuristic filtering to detect Android malware.

Abah, et al. (Abah, 2015), propose an approach that relies on a k-Nearest Neighbor classifier. The features collected include incoming and outgoing SMS and calls, device status, running applications and processes, and so on. In the research by Aung, et al. (Aung, 2013), the authors propose a framework that relies on machine learning algorithms based on features obtained from Android events and permissions.

Aphonso, et al. (Afonso, 2015), propose a dynamic analysis technique that relies primarily on the frequency of system calls and API calls. The main drawback of this approach is that it can detect malware only in cases where the application meets certain API level.

Taintdroid (Enck, 2014) is another dynamic analysis system. This approach analyzes network traffic to search for anomalous behavior. Finally, Maline (Dim-Jasevic, 2015) is another dynamic detection tool based on Android system call analysis.

2.4 Machine Learning Algorithms

In this section, we briefly describe the categories of machine learning algorithms used in this research. For all of these algorithms, we have used the Weka[8] implementation.

2.4.1 Random Forest

Decision trees are one of the simplest learning techniques. However, a decision tree tends to overfit the training data, since it is a literal interpretation of the data, and provides no generalization of the

[8] https://weka.wikispaces.com/ARFF+%28book+version%29

training set. To partially alleviate this problem, multiple decision trees can be used, where each is trained on a subset of the training data. A random forest takes this idea one step further by also training on subsets of the classifiers (Breiman, 2013). Although much of the inherent simplicity of decision trees is lost in this process, random forests have proved to be a very strong machine learning technique over a wide variety of applications.

2.4.2 J.48

The J.48 algorithm is based on a specific implementation of the decision tree algorithm known as C4.5 (Ruggieri, 2000). In this algorithm, a node for the tree is created by splitting the dataset, where the data with highest information gain is chosen at each step.

2.4.3 Naïve Bayes

Naïve Bayes is a classic statistical discrimination technique, the key aspect of which is the assumption that all features are independent of each other[9]. Although this is unlikely to be true in reality, it greatly simplifies the computations, and Naïve Bayes has proven highly successful in many applications.

2.4.4 Simple Logistic

Simple Logistic is an ensemble learning algorithm. To evaluate the base learners, this approach utilizes logistic regression (Shalizi, 2016), using simple regression functions. Similar to linear regression, it tries to find a function that will fit the training data well by computing the weights that maximize the log-likelihood of the logistic regression function.

2.4.5 Sequential Minimal Optimization

The Sequential Minimal Optimization (SMO) technique is a specific implementation of Support Vector Machines (SVM) used in Weka. In SVM, the classification is determined based on a separator between two classes of labeled training data. In SVM, we maximize the "margin", i.e., the separation between the labeled training sets. Another feature of SVM is the so-called kernel trick, where data is, in effect, mapped to a higher dimensional space—with more space to work in, it is likely to be much easier to separate the training data. The SMO classifier uses either a Gaussian or a polynomial kernel (Guptil, 2013).

[9] http://software.ucv.ro/˜cmihaescu/ro/teaching/AIR/docs/Lab4-NaiveBayes.pdf

2.4.6 IBk

The IBk algorithm is an example of a lazy learner. This instance-based learner saves all of the training samples and compares the test samples to each of the members of the training set until it finds the closest match. This algorithm is Weka's version of the well-known k-nearest neighbor classifier[10]. The Weka implementation of IBk uses Euclidean distance as the default distance measure.

3 METHODOLOGY

This section describes the Malware and benign dataset used in the project and the methodology used to extract features from the dataset. We also discuss various implementation details of our approach.

3.1 Datasets

Since there does not appear to be a standard Android benign dataset, we generated our own. Our benign dataset application files were collected from the Google Play Store, which is considered relatively unlikely to contain malware applications. Further, each benign application was classified as such using Virustotal[11], a service which aggregates information from multiple antivirus engines, website scanners, and URL analyzers.

The malware dataset used in this research was acquired from the authors of Drebin (Arp, 2014). This dataset consists of applications obtained from various secondary Android markets, Android websites, malware forums, security blogs, and the Android Malgenome Project (Zhou, 2012b). Each element of the malware dataset was classified as mal- ware based on results from Virustotal. Table 1 gives the numbers of applications in our datasets.

Table 1: Dataset Description.

Application	Number
Malware	103
Benign	97

3.2 Feature Extraction

We extracted static and dynamic features. First, we discuss the static case, and then we turn our attention to the more complex dynamic case.

[10] http://www.statsoft.com/Textbook/kNearest-Neighbors

[11] https://www.virustotal.com

3.2.1 Static Analysis

As mentioned above, an Android application is in the form of an Android package, or `apk`, archive, which is a `zip` bundle. The `apk` archive includes the manifest, along with various other resources and folders. To extract the features of interest, we first need to reverse engineer the `apk` files, which we accomplished using the APK tool in Virustotal.

The file `AndroidManifest.xml` contains several features that could possibly be used for static analysis. Here, we focus on the permissions requested by the application. The AndroidManifest.xml contains a list of all permissions required by the application. Android uses a proprietary binary xml format, so we designed our own custom xml parser to extract the permission features from `AndroidManifest.xml files`.

There are a total of 135 Android permissions. We construct a binary feature vector from the extracted permissions. We denote this feature vector as R = (r1, r2, . . . , r135), where

$$r_i = \begin{cases} 1 \text{ if the } i^{th} \text{ permission is present} \\ 0 \text{ otherwise.} \end{cases} \quad (1)$$

Given an Android application the following steps describe the process we use to extract the permissions features.

1. Reverse engineer the Android application. This reverse engineering is achieved using the APK tool in Virustotal[12].
2. Extract the permissions requested from the AndroidManifest.xml file using our custom xml parser.
3. Generate a binary feature vector, as in (1).
4. Finally, we built a permission vector dataset for all the applications in our dataset store it in an ARFF[13] file format.

Of the 135 possible permissions, many were never requested in any of the Android applications in our datasets. These permissions were removed from consideration, since they contribute nothing to the analysis. Furthermore, some features (i.e., permissions) provide little or no useful information. Thus, to further reduce the length of our feature vectors, we have used feature selection based on a straightforward information gain calculation, which we now describe.

Table 2: Permissions and Entropy Scores.

Information Gain	Permission
0.3507	MOUNT UNMOUNT FILESYSTEMS
0.2372	MANAGE DOCUMENTS
0.2051	READ PHONE STATE
0.1516	INSTALL LOCATION PROVIDER
0.1089	SET WALLPAPER
0.0995	VIBRATE
0.0922	WRITE CALL LOG
0.0838	WAKE LOCK
0.0813	SET PREFERRED APPLICATIONS
0.0722	REQUEST IGNORE BATTERY OPTIMIZATIONS

The information gain of each permission is calculated as

$$\text{gain}(c, r_i) = \text{entropy}(c) - \text{entropy}(c|\, r_i)$$

where c is the label (i.e., either malware of benign) and ri is the ith permission feature. Here entropy(c) is the information entropy. Table 2 shows the list of the top ten permissions (with respect to information gain) and their corresponding information gain. Note that higher values indicate more information is gained from the given attribute.

After eliminating permissions that never appeared and those that resulted in no information gain, we obtained a subset of 99 permissions. Further experiments enabled us to reduced these 99 non-redundant permissions. We found that using the top 87 permissions (with respect to information gain) we obtained the best results (based on the AUC, as discussed in Section 4.1, below), and hence we use a feature vectors of length 87 in all experiments reported below.

For example, a reduced permissions vector from one of the files in our malware dataset is given by

0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0

As another example, a reduced permissions vector from our benign dataset is given by

0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0

[12] https://www.virustotal.com

[13] https://weka.wikispaces.com/ARFF+%28book+version%29

3.2.2 Dynamic Analysis

As expected, an Android application interacts with the operating system through system calls. We have extracted system calls using dynamic analysis. To achieve this, we have made use of the Android emulator that is included with Android Studio[14]. Each Android application in our dataset has been executed in a separate emulator, with the frequency of each system call recorded.

We connect to the emulator instance using the Android Debug Bridge (ADB)[15], which is a command line tool found in the Android SDK. The ADB comes with a so-called Monkey Runner[16], which can be used to emulate random UI interactions. These events include clicks, volume interactions, touches, and so on, which trigger system calls. We record the resulting system calls using the monitoring tool Strace[17]

In detail, the emulation and data collection consists of the following steps.

1. Open the AVD Manager in Android Studio and click on Create New Device. This creates an emulator instance and runs it.
2. After the emulator is running, we open the terminal and navigate to the platform tools folder of the Android SDK. There we enter `adb help` to verify that the ADB is working as expected.
3. Next, we issue the command adb devices which lists the emulator ID that is running.
4. Assuming the Android application is named `ApplicationName.apk`, we give the command

```
adb install ApplicationName.apk
```

(via a batch file). At this point, we can verify that the application file has been installed in the emulator.

5. Next, we enter the emulator shell by typing

```
adb -s emulator-5646 shell
```

at the terminal.

6. We launch the application and check the process ID using the command

```
ps <package name>.
```

7. The command

```
strace -P <ProcessID> -c -o
    <path in emulator>Filename.csv
    <package name>
```

begins the recording of system calls.

8. We start Monkey Runner using the command

```
adb shell -p <package name> -v 500 -s 42.
```

As mentioned above, this generates random events through the user interface. Simultaneously, Strace will record the frequency count of the system calls that are generated.

9. After the Monkey Runner instance stops, we ex- tract the log file using the command

```
adb pull <path in emulator>
    <path in destination>.
```

Of course, the precise sequence of system calls generated will vary, depending on the random selection made by the Monkey Runner. However, the frequency of the various system calls is relatively stable for a given application.

The frequency representation of system calls carries information about the behavior of the application (Burguera, 2011). A particular system call may be utilized more in a malicious application than in a benign application, and the system call frequency representation is intended to capture such information.

Let C = (c1, c2, . . . , cn) be the set of possible system calls available in the Android OS. Then element i in our system call feature vector contains the count for the number of occurrences of system call ci. For example, such a system call vector extracted from one instantiation of one of our benign applications is

0,0,0,0,0,0,2500,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1500,0,0,0,0,0,0,1100,
0,0,0,0,0,0,0,800,0,0,0,0,1,32,0,0,753,0,0,0,36,0,0,0,0,0,0,0,1,0,0,0,
0,0,60,0,0,0,90,0,0,0,0,0,0,0,1,0,0,0,0,298,0,0,966,0,56,0,0,0,0,0,0,0,
756,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,150,0,0,0,0,0,0,110,0,0,0,0,0,0,0,0,
0,0,0,1,0,0,0,660,0,0,0,0,0,0,0,0,0,0,0,55,0,0,0,0,0,60,0,0,0,0,0,0,0,
0,0,0,1,0,0,0,0,298,0,0,0,87,1,0,0,0,0,0,0,0,0,82,0,0,0,0,0,0,0,0,0,0,
0,0,0,1500,0,0,0,0,0,0,1250,0,0,0,0,0,0,0,885,0,0,0,0,65,0,0,0,0,0,0,0,25,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,9,0,0,0,0,0,0,0,1,0,0,0,0,298,0,0,0,82,1,0,
8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2580,0,0,0,0,0,0,1100,0,0,0,
0,0,0,0,800,0,0,0,0,1,0,0,0,0,0,0,0,0,00,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,60,0,0,0,
900,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,426,0,0,65

A system call vector from one of the Android malware application in our dataset is given by

0,0,0,0,0,0,0,8400,0,0,0,0,0,0,110,0,0,0,0,0,0,0,0,0,0,0,1500,0,0,0,0,0,0,0,1100,
0,0,0,0,0,0,0,0,800,0,0,0,0,0,1,32,0,0,6523,0,0,0,368,0,0,0,0,0,0,0,0,0,1,0,0,0,
0,0,60,0,0,0,90,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,298,0,0,966,0,5600,0,0,0,0,0,0,0,0,
756,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,150,0,0,0,0,0,0,110,0,0,0,0,0,0,0,0,0,0,
0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5865,0,0,0,0,0,600,0,0,0,0,0,0,0,0,0,0,
1,0,0,0,0,298,0,4260,0,0,0,0,0,0,0,0,0,0,0,0,0,82,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
1500,0,0,0,0,0,0,1250,0,0,0,0,0,0,0,885,0,0,0,0,6500,0,0,0,2238,0,0,0,250,
0,0,0,0,62,0,0,1,0,0,0,0,0,0,60,0,0,0,0,9,0,0,0,0,0,0,0,0,1,0,0,0,0,298,0,5024,0,
8785,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1500,0,0,0,0,0,0,
1100,0,0,0,0,0,0,0,0,800,0,0,0,0,0,0,252,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,98,0,426,0,0,855

[14] http://developer.android.com/tools/studio/

[15] http://developer.android.com/tools/help/adb.html

[16] http://developer.android.com/tools/help/monkey.html

[17] http://linux.die.net/man/1/strace

Table 3: System Configurations.

Host Machine	
Model	Dell Inspiron 15R
Processor	Intel™ @ 1.80GHz
RAM	8.00 GB
System Type	64bit OS
Operating System	Windows 10
Guest Machine	
Operating System	Ubuntu 12.04 LTS
Memory	226.00 GB
System Type	32bit OS
Android Emulator Configuration	
Platform	Android Studio 1.5.1
Device	Nexus 5
Target	Android 4.4.2- API level 19
CPU/ABU	Intel Atom(x86)
RAM	1536 MB
SD Card	200 MB

4 EXPERIMENTS

We conducted several sets of experiments. First, we carried out experiments to compare the effectiveness of various machine learning algorithms in the Android malware detection context. Second, the effectiveness of classification based on the dynamic system call frequency data was analyzed. Third, the effectiveness of classification based on the static analysis of permissions data was evaluated. Finally, experiments were carried out based on combined permission and system call data. Furthermore, in each of the latter three cases, we carefully quantify the robustness of the scoring technique.

All experimental results given in this paper are based on 10-fold cross validation. That is, our malware set is randomly partitioned into 10 subsets, say, S1, S2,..., S10. Then subsets S2 through S10 are used for training, with subset S1 and the benign set reserved for testing. This training and scoring process is repeated nine more times, with a different subset reserved for testing in each iteration. The scoring results from all 10 "folds" are accumulated and considered together as one experiment. Cross validation serves to reduce the effect of any bias in the data, and it also maximizes the number of scores obtained from a given dataset.

The system configuration used for all of the experiments reported in this paper is given in Table 3.

4.1 Evaluation Metric

To evaluate the success of our experiments, we rely on the area under the ROC curve (AUC). Given a scatterplot of scores for benign and malware cases, an ROC curve is a graph of the true positive rate (TPR) versus the false positive rate (FPR) as the threshold varies through the range of scores. An AUC of 1.0 indicates the ideal case, where there exists a threshold that completely separates the benign and malware scores, while an AUC of 0.5 indicates that the binary classifier is no better than flipping a coin. In general, the AUC can be interpreted as the probability that a randomly selected positive instance scores better than a randomly selected negative instance (Hand, 2001). One advantage of the AUC as compared to measuring accuracy is that no explicit thresholding is required when computing the AUC. In fact, the AUC takes all possible thresholds into account.

Table 4: Comparison of Machine Learning Algorithms.

Algorithm	Static AUC	Dynamic AUC
RF 100	0.9660	0.8840
RF 10	0.9630	0.8229
J.48	0.9210	0.7398
Naïve Bayes	0.9580	0.5000
Simple Logistic	0.9480	0.5990
BayesNet TAN	0.9450	0.7990
BayesNet K2	0.9580	0.7990
SMO Poly	0.8970	0.6050
SMO NPoly	0.8970	0.6390
IBk 1	0.9160	0.6140
IBk 3	0.9350	0.5700
IBk 5	0.9340	0.6104
IBk 10	0.9410	0.5844

4.2 Results

In this section, we first compare various machine learning algorithms. Then we turn our attention to detailed analyses of detection based on static, dynamic, and combined feature sets.

4.2.1 Comparison of Machine Learning Algorithms

Table 4 shows the AUC values of different algorithms available on Weka based on (dynamic) system calls and (static) permissions. This same information is given in the form of a bar graph in Figure 3.

From these results, we see that a Random Forest with 100 trees gives the best results. Consequently, we use this algorithm in the remainder of the experiments reported in this paper.

4.2.2 System Calls and Permissions Analysis

To analyze system calls, we train on the dynamically extracted feature vector containing system call

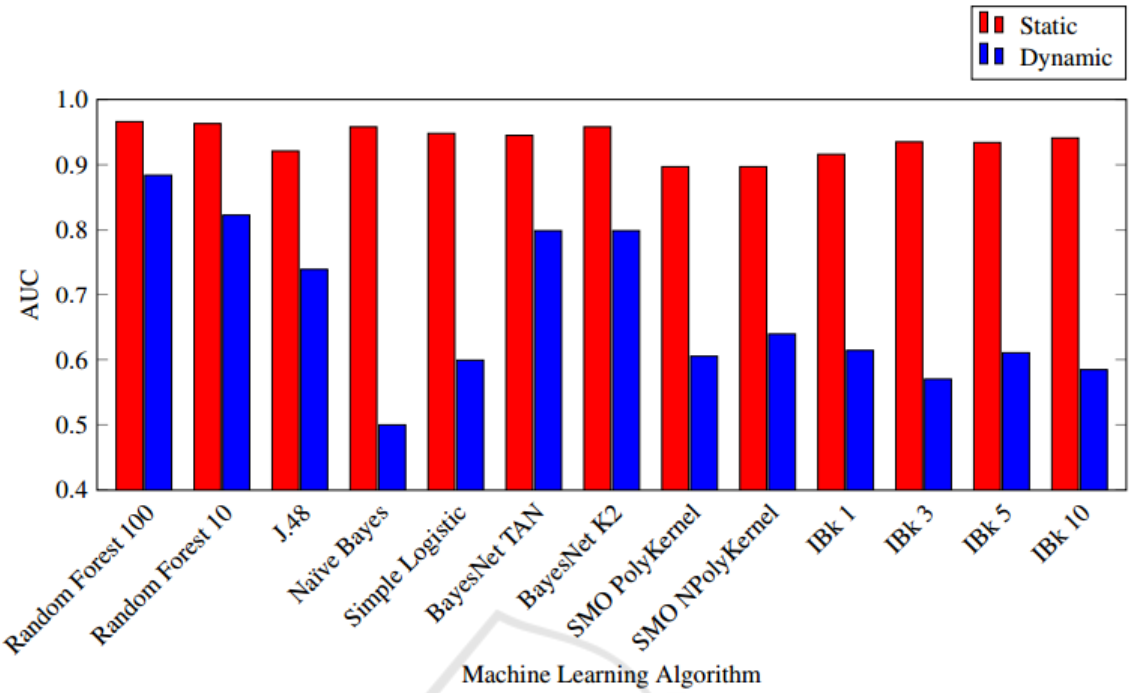


Figure 3: AUC Comparison of Machine Learning Algorithms.

frequencies. The feature extraction process is described above in Section 3.2.2. For this experiment, we obtain an AUC of 0.884, which implies that the system calls feature alone does not yield particularly strong detection result.

We also evaluated our (static) permission feature in a similar manner. Recall that this feature extraction process is described in Section 3.2.1. In this case, we obtain an AUC of 0.972. This results is quite strong and shows that a fairly simple static feature can be used to detect Android malware with high accuracy.

4.2.3 Robustness Analysis

Next, we want to analyze the robustness of each of these scoring techniques—individually, and in combination. Here, we mimic the effect of a malware developer who tries to make the permissions and system calls of Android malware look more similar to those of a benign application. Since the number of permissions and system calls tends to be much larger in mal-ware applications, we analyze the robustness of our scoring techniques when these numbers are reduced in the malware applications.

The results in Figure 4(a) show the effect of reducing the number of permissions. The analogous results for system calls are given in Figure 4(b).

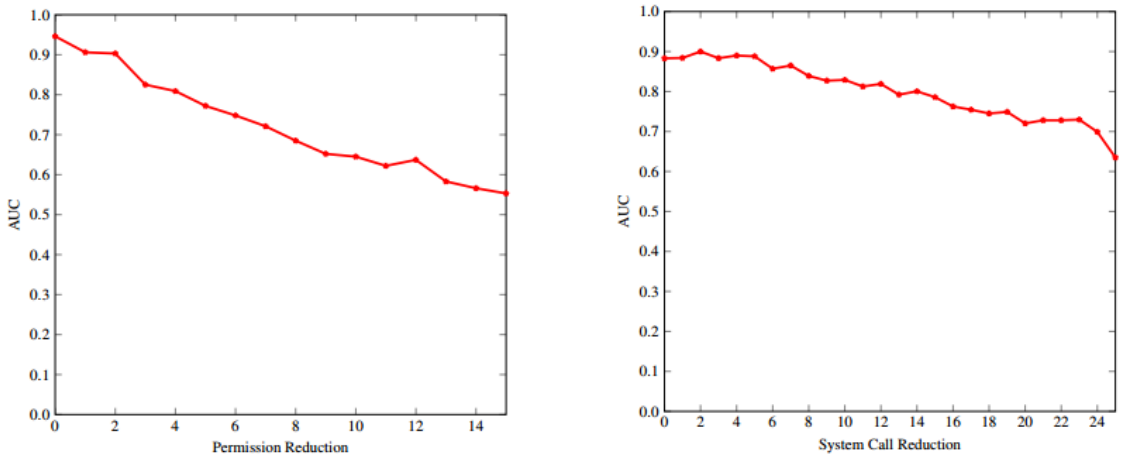
As can be seen from Figure 4, reducing the number of system calls has a limited effect, while even a slight reduction in the number of permissions can have a large effect.

The static and dynamic features considered here can easily be combined, and hence it is important to analyze their robustness in combination. This experiment has been conducted, with the results given in the form of 3-dimensional graph in Figure 5.

From the results in Figure 5, we can clearly see the interplay between permissions and system calls is somewhat more complex than might be expected from merely viewing the permissions and system calls independently, as in Figure 4. While it is necessary that the malware writer reduce the number of permissions, unless this is accompanied by a significant reduction in the number of system calls, fairly strong detection results can still be obtained in the combined case.

5 CONCLUSION AND FUTURE WORK

For Android malware detection, we have observed that a simple static feature based on permissions is significantly more informative than a dynamic feature based on system calls. This is, perhaps, somewhat surprising, since in much of the malware detection literature, system calls are treated as essentially the



(a) Reduced Permissions (b) Reduced System Calls

Figure 4: Robustness of Permissions and System Calls Separately.

"gold standard" for detection (Tamada, 2007; Vemparala, 2016; Wang, 2009).

The robustness analysis in this paper shows that even a slight reduction in the number of permissions can have a substantial benefit, from the malware writer's perspective. Furthermore, although the dynamic system call feature is not particularly strong, it is relatively robust, so that it can serve a useful purpose when combined with other features.

For future work, the combined feature set can be evaluated using other machine learning techniques. Also, our dynamic features were collected using Monkey Runner, which could fail to execute the malicious parts of the code. A more intelligent approach to extracting the system calls might yield stronger detection results—at the cost of greater complexity and more work.

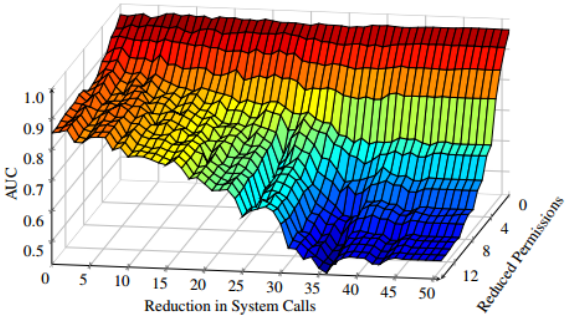


Figure 5: Robustness of System Calls and Permissions in Combination.

REFERENCES

Abah, J., e. a. (2015). A machine learning approach to anomaly-based detection on android platforms. *International Journal of Network Security and Its Applications, 7(6):15–35*.

Afonso, V., M. e. a. (2015). Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques, 11(1):9–17*.

Arp, D., e. a. (2014). Drebin: Efficient and explainable detection of android malware in your pocket. *21th Annual Network and Distributed System Security Symposium (NDSS)*.

Aung, Z., e. a. (2013). Permission-based android malware detection. *International Journal of Scientific Technology Research, Volume 2, Issue 3*.

Breiman, L., e. a. (2013). Random forests. Burguera, I., e. a. (2011). Crowdroid: behavior-based malware detection system for android. *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp.15–26*.

Dimjasevic, M., e. a. (2015). Evaluation of android malware detection based on system calls.

Enck, W., e. a. (2014). Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems, 32(2):1–29*.

Feng, Y., e. a. (2014). 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*.

Fuchs, P., e. a. (2009). Scandroid: Automated security certification of android applications. *Technical Report CSTR-4991, Department of Computer Science, University of Maryland,College Park*.

Guptil, B. (2013). Examining application components to reveal android malware.

Hand, J., e. a. (2001). A simple generalisation of the area under the roc curve for multiple class classification problems, 45(2):171-186.

Ruggieri, S. (2000). Efficient c4.5.

Saudi, M., e. a. (2015). Android mobile malware surveillance exploitation via call logs: Proof of concept. *17th UKSIM-AMSS International Conference on Modelling and Simulation, pp. 176–181*.

Shalizi, C. (2016). Logistic regression. *Advanced Data Analysis from an Elementary Point of View, Chapter 12*.

Spreitzenbarth, M., e. a. (2014). Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security, 14(2):141–153*.

Tamada, H., e. a. (2007). Design and evaluation of dynamic software birthmarks based on api calls. *Nara Institute of Science and Technology, Technical Report*.

Vemparala, S. (2016). Malware detection using dynamic birthmarks. *2nd International Workshop on Security & Privacy Analytics (IWSPA 2016), co-located with ACM CODASPY 2016, March 9–11*.

Wang, X., e. a. (2009). Detecting software theft via system call based birthmarks. *Proceedings of 25th Annual Computer Security Applications Conference*.

Zhou, Y., e. a. (2012a). Detecting malicious apps in official and alternative android markets. *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*.

Zhou, Y., e. a. (2012b). Dissecting android malware: Characterization and evolution. *Proceedings of the 33rd IEEE Symposium on Security and Privacy, 95–109*.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

View publication stats