J Comput Virol Hack Tech (2015) 11:9–17
DOI 10.1007/s11416-014-0226-7

ORIGINAL PAPER

Identifying Android malware using dynamically obtained features

Vitor Monte Afonso · Matheus Favero de Amorim · André Ricardo Abed Grégio · Glauco Barroso Junquera · Paulo Lício de Geus

Received: 5 August 2014 / Accepted: 1 October 2014 / Published online: 17 October 2014
© Springer-Verlag France 2014

Abstract The constant evolution of mobile devices' resources and features turned ordinary phones into powerful and portable computers, leading their users to perform payments, store sensitive information and even to access other accounts on remote machines. This scenario has contributed to the rapid rise of new malware samples targeting mobile platforms. Given that Android is the most widespread mobile operating system and that it provides more options regarding application markets (official and alternative stores), it has been the main target for mobile malware. As such, markets that publish Android applications have been used as a point of infection for many users, who unknowingly download some popular applications that are in fact disguised malware. Hence, there is an urge for techniques to analyze and identify malicious applications before they are published and able to harm users. In this article, we present a system to dynamically identify whether an Android application is malicious or not, based on machine learning and features extracted from Android API calls and system call traces. We evaluated our system with 7,520 apps, 3,780 for training and 3,740 for testing, and obtained a detection rate of 96.66 %.

V. M. Afonso (✉) · M. F. de Amorim · A. R. A. Grégio · P. L. de Geus
University of Campinas, Campinas, SP, Brazil
e-mail: vitor@lasca.ic.unicamp.br

M. F. de Amorim
e-mail: matheus@lasca.ic.unicamp.br

A. R. A. Grégio
e-mail: gregio@lasca.ic.unicamp.br

P. L. de Geus
e-mail: paulo@lasca.ic.unicamp.br

G. B. Junquera
Samsung Institute for Informatics Development (SIDI), Campinas, SP, Brazil
e-mail: glauco.b@samsung.com

1 Introduction

Mobile devices have been ubiquitously widespread as personal and professional tools whose computing power is approaching that of ordinary desktop computers. Consequently, smartphone users are able to do more complex tasks with their devices, such as producing documents and spreadsheets, making video conferences and managing their Internet Banking accounts. These users are now storing all sorts of sensitive information on their devices (e.g., bank credentials, corporate documents), effectively creating an interesting and potentially lucrative scenario for cybercriminals. To take advantage of this situation, attackers are ramping up the creation of malicious applications that affect mobile devices.

Since Android is the most widespread operating system for mobile devices [7], it is the main target of mobile malware. According to Juniper [11], the amount of malicious applications discovered between March 2012 and March 2013 has increased 614 %, considering all mobile platforms. In addition, the same report states that 92 % of every malware that affects mobile devices targets the Android operating system. Users obtain Android applications mostly from markets, including Google Play—Google's official market—and others known as "alternatives". In order to infect users' devices, attackers submit to markets malware that look like legitimate applications, such as games. In fact, many of the available malware are repackaged versions of legitimate applications, i.e., applications modified to include malicious code and republished in the markets.

Some works in the literature refer to the presence of malicious applications both in the official market and in alternative ones [8,24]. They show that the official market does a better job at filtering out malicious applications, but nonetheless is still used as a vector to infect users. Addressing this issue requires the development and deployment of improved

techniques to analyze and identify malicious Android applications.

To that effect, several approaches based on static and dynamic analysis have been proposed to detect malicious Android applications [4,14,17,22,24], but all of them present shortcomings regarding their detection scope or ability. Firstly, approaches that rely on static analysis of the application's code have a hard time dealing with highly obfuscated samples [12] and only analyze code packed with the application file, missing code that can be downloaded and executed at runtime [13]. Secondly, although Android malware samples do not make use of obfuscation techniques as heavy as those affecting Windows desktops, the natural evolution of Android malware will inevitably lead to the improvement of obfuscation techniques currently used [15], turning static analysis into a difficult proposition. Moreover, dynamic analysis approaches usually suffer from not being able to observe the malicious behavior of some samples due to their ever growing awareness of the analysis environment, to the lack of appropriate stimulation under the analysis environment or else to the inability of the malware sample under analysis to obtain some required external data.

In general, detection techniques for Android malware use statically extracted data from the manifest file or from Android API function calls, as well as dynamically obtained information from network traffic and system call tracing. However, most articles available in the literature whose focus lies on malware identification either use small datasets or require manual steps at some stage of the process. In this paper, we present a system that identifies malicious Android applications based on a machine learning classifier, using dynamically obtained features. These features are extracted from Android API function calls and system call traces. We trained our classifier with 3,780 samples and tested it with 3,740 samples (with both datasets including malicious and benign applications), which was then able to correctly classify 96.66 % of those samples. The results obtained were compared to the ones from other Android malware detection approaches and demonstrate the relevance of our system. Using a larger dataset, we obtained results similar to the state-of-the-art, including static and dynamic approaches. Furthermore, we show that features extracted from API function calls, which, as far as we know, were not used by other automatic and dynamic approaches, are very good for the identification of malware.

The main contributions of this paper are:

- We developed an analysis system to monitor Android API function calls as well as system calls, in order to gather information (features) required to detect malicious behavior. Currently available systems are tied to Android OS versions (some of them to older versions, such as 2.x) or to the SDK-provided emulator, whereas our approach is independent of the emulator and much more portable as it does not modify Android OS;
- From that, we developed a system that classifies applications as benign or malicious and tested it with thousands of apps, correctly classifying 96.66 % of them. To accomplish better training and accuracy, we extract novel features showing that those based on API function calls greatly increase the detection rate.

The remainder of this paper is organized as follows. Section 2 provides a background about Android malware and presents related work. The developed system is introduced in Sect. 3, whereas evaluation results and discussion are presented in Sect. 4. Section 5 discusses some of the limitations and, in Sect. 6, we conclude this paper and discuss some follow-up work.

2 Background and related work

Based on reports from antivirus companies, the authors of [6] describe the behavior of 46 malware samples collected between January 2009 and June 2011. The malicious behaviors identified were the following: user information stealing; premium calls and SMS messages[1]; SPAM SMS messages; novelty and amusement[2]; user credential stealing; search engine optimization; and ransom.

A similar study is presented in [23], but in this case the authors analyzed the samples manually. They used a dataset of 1,260 Android malware samples, which were collected between August 2010 and October 2011 and were separated in 49 malware families. The authors describe the behavior of these samples and show information regarding their time of discovery in the official market and in alternative ones. For each family, they specify how the malware is installed, how the malicious behavior is activated and what the malicious payload is. Also, the authors indicate the events monitored by the malware and the exploits used by them for privilege escalation.

Android malware detection is a critical task towards protecting users of application markets and improving these markets' vetting processes. However, detection is intimately bound to analysis, since features must be extracted so as to generate signatures or behavioral profiles. There are systems proposed in the literature that aim to analyze apps from markets in order to detect the presence of malware among them, as well as others solely with the purpose of providing useful information about static and dynamic characteristics of an unknown application. We discuss some of these systems in the following sections.

[1] These actions generate costs to the user.

[2] Some samples performed actions that seemed to be only useful for the amusement of the author.

2.1 Android malware analysis

Enck et al. [5] propose TaintDroid, a dynamic taint analysis system, which tracks sensitive data flow to detect when it is sent over the network. TaintDroid instruments the Android virtual machine interpreter and some APIs to accomplish system-wide taint tracking, but it does not handle native code. Their results show that seemingly unsupicious applications often disclose sensitive data, such as location, UUID and phone number. Although based on permissions granted by users, the data exposure monitoring process requires the application to be dynamically analyzed.

DroidBox [3] is a dynamic analysis system that builds upon TaintDroid and provides API calls, network data and data leaks, besides other important information.

Andrubis [10], which has a publicly available submission interface, is a system whose goal is to analyze Android applications using static and dynamic techniques. In the static analysis step, Andrubis collects information about required permissions, components to communicate with the operating system, intent-filters and URLs found in the bytecode. Dynamically-based information collection is accomplished through instrumentation of the Dalvik VM, taint tracking and network traffic capture. Andrubis is based on TaintDroid, DroidBox and other related projects. Yan and Yin propose DroidScope [20], a virtual machine introspection-based analysis system that bridges the semantic gap reconstructing OS-level and Java-level semantic views from outside. They also developed additional analysis tools to provide taint tracking and several levels of instruction tracing.

Spreitzenbarth et al. [16] present Mobile-Sandbox, a system that combines static and dynamic analysis techniques to obtain Android applications' behavior. Mobile-Sandbox's static analysis includes parsing the manifest file and the extracted bytecode, and aims to guide the dynamic analysis process, which is based on TaintDroid and DroidBox. In addition, Mobile-Sandbox monitors native code using the `ltrace` tool and analyzes network traffic captured during the application's execution. Another system that uses both static and dynamic analysis is AASandbox [2]. During static analysis, the system decompiles the application to Java code and look for suspicious patterns, such as the use of `Runtime.exec()` and functions related to reflection. During the dynamic step, AASandbox runs the application on a controlled environment and monitors system calls using a kernel module.

2.2 Android malware detection

Zhou et al. [24] propose DroidRanger, a two-scheme system based on signatures and heuristics that intends to detect Android malware. On the one hand, the signature-based scheme relies on common permissions and behavioral footprints to identify samples from known families. On the other hand, the heuristics-based filtering scheme identifies suspicious behaviors (e.g., downloading and executing code from Web and dynamic loading of native code). Applications identified as suspicious are manually analyzed and if they are indeed malicious, the information necessary to detect samples from the same family in the signature-based step are manually extracted.

Zheng et al. [22] propose DroidAnalytics, a system to automatically collect, analyze and detect Android malware that makes use of repackaging, code obfuscation or dynamic payloads. Collection is accomplished by an extensible application crawler that receives marketplaces (official and alternatives) or URLs as input. Collected applications are then disassembled so as to obtain Android API calls. These API calls are used within a three-level signature generation process, which extracts malware features at the opcode level to identify variants. The dynamic analysis step consists of running samples that present network behavior, inside an emulator, in order to download additional pieces of code.

Sanz et al. [14] introduce PUMA, an Android malware detection method based on machine learning that uses information obtained from application's permissions. To evaluate their method, they collected 1,811 supposedly benign applications of several categories from Android Market and 249 unique malicious samples from the VirusTotal database. The features used to represent each sample are based on the set of permissions and the device's features required by the application. Using this information, the authors evaluated eight algorithms available in the WEKA framework and concluded that RandomForest provided the best results.

Elish et al. [4] propose a tool to determine whether unknown applications are malicious or not based on static data dependence analysis, aiming to identify software execution patterns related to the correlation of user inputs with critical function calls. They construct a data dependence graph for each analyzed application, which can then be used in comparisons to identify stealthy Android malware. Although their results show that the analyzed malware samples are distinguishable from the legitimate applications, since the former performed sensitive function calls without any user input.

Wu et al. [19] propose DroidMat, a detection system based on clustering techniques applied to statically extracted features from the application's manifest file (permission, component and intent information) and permission-related Android API call traces from the application's bytecode. The process for evaluating the system applied four combinations of clustering and classification algorithms to analyze a dataset of 1,500 benign applications (downloaded from GooglePlay) and 238 malicious ones, and resulted in 97.87 % accuracy.

Another system that performs Android malware detection using features obtained statically is DREBIN [1]. This system uses machine learning and features extracted from the

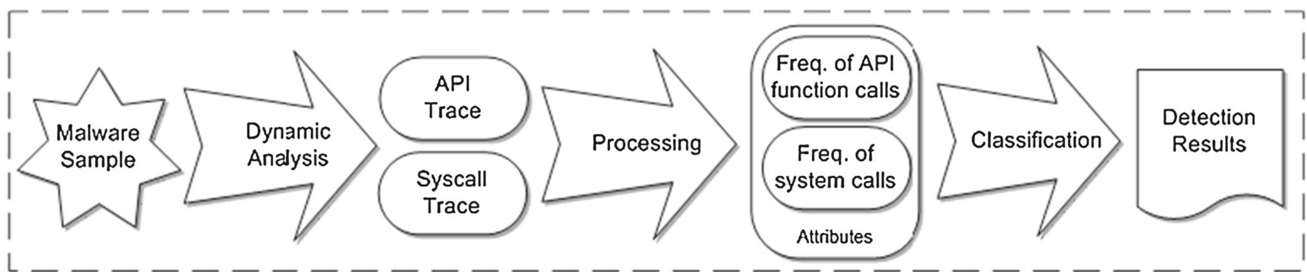

Fig. 1 System overview

manifest and the dex code of applications. The authors performed experiments with 123,453 benign samples and 5,560 malicious samples and the system obtained 93 % of accuracy.

Su et al. [17] present a smartphone dual defense protection framework to perform detection of malicious applications, using machine learning, as they are submitted for release on Android markets. Their approach consists of dynamically analyzing a new application to collect two sets of features: one related to system call tracing and the other related to network traffic statistics. A system call monitoring process makes use of the Linux's `strace` tool and restricts itself to 15 (of almost 300) of them that are related to process, memory and I/O activities. The `tcpdump` tool is used to capture network traffic, from which TCP/IP flows are extracted. The training of the system calls classifier involved 200 benign and 180 malicious applications, whereas the training of the network classifier involved 60 benign and 49 malicious applications. Both classifiers are based on WEKA's implementation of J.48 and RandomForest algorithms. The authors selected 70 benign and 50 malicious applications to evaluate their classifiers and obtained an accuracy rate of 94.2 and 99.2 % for J.48 and RandomForest, respectively.

3 System overview

Figure 1 presents the system overview. To identify malicious applications, the developed system obtains information about the application's behavior using dynamic analysis. This process is explained in Sect. 3.1. The obtained information is comprised by Android API function calls and system calls, and is fed to a processor, which extracts features from the information. These features are composed by the frequency of use of API functions and system calls, and are used by a classifier to categorize the application as malicious or benign. The feature extraction and classification processes are explained in Sect. 3.2.

3.1 Data extraction

To obtain its behavior, the application is first instrumented by APIMonitor[3], a tool that modifies the application so that calls to certain functions are registered, along with the parameters passed and the return value. We modified the `default_api_collection`[4] file, used by APIMonitor, to include methods related to network access, process execution, string manipulation, file manipulation and information reading. The instrumented version of the application is executed for five minutes in the standard Android emulator—distributed with the Android SDK.

The analysis of Android API function calls is important because it allows the extraction of high-level information about the behavior of applications. However, some applications use native code instead of Android API functions. Thus, through the `strace` tool, we also monitor the system calls executed by the application. Section 3.1.1 presents examples of registered API function calls and system calls.

The advantages of our monitoring process are not needing to modify the Android code and also being independent of the virtualization platform. Analysis systems that use a modified version of Android, such as TaintDroid [5], Andrubis [10] and Mobile-Sandbox [16], need to be constantly updated to the newest version of the Android system, a task that is quite time-consuming, so that they are able to analyze samples that target that particular version of the operating system. Moreover, systems that use virtual machine introspection, such as Droidscope [20], are dependent on the virtualization platform (e.g., Qemu) and cannot be used on a different virtualization platform or on a bare-metal one.

On the negative side, the disadvantages of our monitoring system are the use of a monitoring tool inside the analysis environment and the modification of the analyzed sample. These actions make the system more detectable by malware, which can stop the execution or execute benign actions when it becomes aware of the analysis. The previously mentioned systems that use virtual machine introspection or a modified version of Android do not suffer from that. Although in these cases the monitoring tool cannot be detected, the malware sample can still detect the virtual or emulated environment, if it is not a bare-metal platform.

[3] https://code.google.com/p/droidbox/wiki/APIMonitor.

[4] This file defines the functions that are monitored.

3.1.1 Log examples

Listings 1 and 2 present examples of API function calls registered by the instrumented application. Listing 1 presents a call to a function that sends an SMS message. In this case the destination number is "7132" and the message is "846978". Listing 2 presents a call to a function the executes a process. The executed process is `/data/data/org.zenthoughtflashrec/cache/asroot` and the parameters are `/data/data/org.zenthought.flashrec/cache/explXXXXXX`, `/data/data/org.zenthought.flashrec/cache/dump_image`, `recovery` and `/mnt/sdcard/recovery-backup.img`.

Listing 1 Call to send an SMS message

```
Landroid/telephony/SmsManager;−>sendTextMessage(Ljava/lang/
    String;=7132
| Ljava/lang/String;=null
| Ljava/lang/String;=846978
| Landroid/app/PendingIntent;=null
| Landroid/app/PendingIntent;=null)V
```

Listing 2 Call to execute a process

```
Ljava/lang/Runtime;−>exec([Ljava/lang/String;={
/data/data/org.zenthought.flashrec/cache/asroot,
/data/data/org.zenthought.flashrec/cache/explXXXXXX,
/data/data/org.zenthought.flashrec/cache/dump_image,
recovery,
/mnt/sdcard/recovery−backup.img})Ljava/lang/Process;=Process[
    id=541]
```

Listing 3 presents two calls to the `execve` system call. They were both used to obtain information about the device, one focusing on CPU information and the other on memory information.

Listing 3 Examples of registered system calls

```
execve(''/system/bin/cat'', [''/system/bin/cat'', ''/proc/cpuinfo
    ''],
[''ANDROID_SOCKET_zygote=9'', ''
    ANDROID_BOOTLOGO=1'',
  ''EXTERNAL_STORAGE=/mnt/sdcard'', ''
    ANDROID_ASSETS=/system/app'',
  ''PATH=/sbin:/vendor/bin:/system/s''...,
  ''ASEC_MOUNTPOINT=/mnt/asec'', ''
    LOOP_MOUNTPOINT=/mnt/obb'',
  ''BOOTCLASSPATH=/system/framework/''..., ''
    ANDROID_DATA=/data'',
  ''LD_LIBRARY_PATH=/vendor/lib:/sys''..., ''
    ANDROID_ROOT=/system'',
  ''ANDROID_PROPERTY_WORKSPACE=8,327''...]) = 0

execve(''/system/bin/cat'', [''/system/bin/cat'', ''/proc/meminfo
    ''],
[''ANDROID_SOCKET_zygote=9'', ''
    ANDROID_BOOTLOGO=1'',
  ''EXTERNAL_STORAGE=/mnt/sdcard'', ''
    ANDROID_ASSETS=/system/app'',
  ''PATH=/sbin:/vendor/bin:/system/s''...,
  ''ASEC_MOUNTPOINT=/mnt/asec'', ''
    LOOP_MOUNTPOINT=/mnt/obb'',
  ''BOOTCLASSPATH=/system/framework/''..., ''
    ANDROID_DATA=/data'',
  ''LD_LIBRARY_PATH=/vendor/lib:/sys''..., ''
    ANDROID_ROOT=/system'',
  ''ANDROID_PROPERTY_WORKSPACE=8,327''...]) = 0
```

3.1.2 Analysis stimulation

Some actions of the malware are only carried out if certain events are observed or if certain interactions with the graphic interface are performed. To stimulate these actions we automatically generate random events with the MonkeyRunner tool, which is distributed with the Android SDK, and create some events related to phone calls, SMS messages, geographic location and battery state, using the emulator.

As the events that interact with the graphic interface are generated randomly, they may not lead the application to execute the malicious code. One way to solve that is manually interacting with the applications during the analyzes, but when analyzing a large number of applications, it becomes too time-consuming. Another way to do this is by statically identifying which interactions are necessary to reach the relevant portions of the code and provide these interactions during the analysis. This approach is used by [21], but their system requires a modified version of the Android OS, which may be a problem, as discussed earlier. Another way to do this would be to identify the necessary interactions as done by [21], but generate them without needing a modified version of the OS. We leave this as a future work.

To make the analysis system more similar to the system of a real user, making it harder for malware to identify it is being analyzed, we changed the IMEI and phone number of the device [18]. Moreover, we added some contact information.

3.2 Malware identification

The attributes used to classify the applications as malicious or benign are extracted from the data obtained during dynamic analysis. More precisely, we extract the amount of calls to each one of the 74 monitored Android API functions and the amount of calls to each one of 90 system calls[5].

For example, after an analysis, if the API function calls log produced the results illustrated in Listings 1 and 2, and the system call trace produced Listing 3, the attributes of the evaluated sample would be the following array: `1,1,0,...,0,2,0,...,0`, in which the first two "1" refer to the `android/telephony/SmsManager;->sendTextMessage` and the `java/lang/Runtime;->exec` API function calls, and the following "zeroes" refer to the frequency of the other API function calls, whereas the

[5] The lists of API functions and system calls used are presented in http://pastebin.com/T7Yfbksq and http://pastebin.com/5Xyjh8GS.

Table 1 The amount of malicious and benign samples in the training and testing datasets

	Training	Testing	Total
Malicious	2,295	2,257	4,552
Benign	1,485	1,483	2,968
Total	3,780	3,740	7,520

"2" refer to the `execve` system call, followed by a sequence of "zeroes" related to the remaining system calls' frequencies.

To create the classifier we first evaluated several algorithms, using the Weka [9] framework, and the one that performed best was RandomForest (with 100 trees). This experiment is detailed in Sect. 4.2.

4 Evaluation

This section describes the datasets used in the evaluations, the comparison of algorithms performed to select which one would compose the classifier and the test carried out to evaluate the classification system, including a comparison with other systems.

4.1 Datasets

The malicious application dataset is composed by samples from the "Malgenome Project" [23] and from a torrent file acquired from VirusShare (http://tracker.virusshare.com:6969/), totalling 4,552 samples. To compose the benign dataset we developed a crawler to collect applications from the AndroidPIT market (http://www.androidpit.com/). Through it, we gathered 3,831 applications to compose the benign dataset. These applications were submitted to VirusTotal, a system that uses more than 40 antivirus systems to scan the submitted file, and the ones that were detected by at least one antivirus were removed. Hence, the benign dataset contains 2,968 applications. In order to compose the training and testing datasets, we randomly split the malicious and benign datasets. Table 1 shows the amount of malicious and benign samples in the training and testing datasets.[6]

4.2 Evaluation of classification algorithms

In order to identify which algorithm to use in the classifier, we compared the results obtained using several machine learning algorithms (the same ones used in [14]). For this test we used the training dataset mentioned before. Table 2 presents the algorithms and configurations used in the comparison. Furthermore, Table 3 presents the accuracy yielded by the 10-fold validation performed using each algorithm. The accuracy was calculated as $Accuracy = \frac{(TP+TN)}{(TP+TN+FP+FN)}$, with FP being false-positive, FN being false-negative, TP being true-positive and TN being true-negative. The algorithm that achieved th best results was RandomForest with 100 trees. The RandomForest algorithm generates several decision trees and chooses the one with the best results.

Table 2 The algorithms and configurations used in the evaluation to select the algorithm to be used by our classifier

Algorithm	Configurations
RandomForest	Number of trees {10, 50, 100}
J.48	Default
SimpleLogistic	Default
NaiveBayes	Default
BayesNet	Search algorithm {K2, TAN}
SMO	Kernel {PolyKernel, NormalizedPolyKernel}
IBk	Value of k {1, 3, 5, 10}

Table 3 Comparison of the detection using several classification algorithms over the training dataset with 10-fold validation

Algorithm	Accuracy (%)
RandomForest 10	93.20
RandomForest 50	95.65
RandomForest 100	95.96
J.48	93.04
NaiveBayes	82.39
SimpleLogistic	67.92
BayesNet TAN	74.53
BayesNet K2	89.92
SMO PolyKernel	75.03
SMO NPolyKernel	85.45
IBk 1	89.92
IBk 3	87.60
IBk 5	86.85
IBk 10	83.70

4.3 Detection evaluation

As the RandomForest algorithm (with 100 trees) yielded the best results in the previous experiment, we used it to evaluate our detection system. We trained the classifier using the training dataset and used it to classify the testing dataset. Table 4 presents the confusion matrix with the results of this test. From the 2,257 malicious applications used for testing, 2,168 were correctly classified and 89 were false-negatives,

[6] The lists with the SHA-1 hash values of the samples used can be found at http://pastebin.com/0K9Xxj7U (training/malicious), http://pastebin.com/FCp9pCsK (training/benign), http://pastebin.com/ZwLnDPJd (testing/malicious) and http://pastebin.com/apV32ywX (testing/benign).

Table 4 Confusion matrix with the detection results using the RandomForest (100) algorithm

Results	Correct class		Total
	Malicious	Benign	
Malicious	2,168	36	2,204
Benign	89	1,447	1,536
Total	2,257	1,483	3,740

i.e., malicious applications classified as benign. From the 1,483 benign applications, 1,447 were classified as such, whereas 36 were considered malicious, comprising the false-positives. The values of false-positive, false-negative, true-positive, true-negative, accuracy (A), recall (R), precision (P), harmonic mean (F-measure) and the amount of correctly classified samples are shown in Table 5. The recall was calculated as $Recall = \frac{(TP)}{(TP+FN)}$, the precision was calculated as $Precision = \frac{(TP)}{(TP+FP)}$ and the harmonic mean was calculated as $F\text{-}measure = \frac{(2*R*P)}{(R+P)}$.

4.4 Discussion

Table 6 presents the comparison of the results obtained by our system with the results presented in [1,14,17,19]. The PUMA [14], DREBIN [1] and DroidMat [19] systems statically extract features, whereas our system and the one presented in [17] do it dynamically. Though the results obtained by DroidMat are a little better than ours, systems that rely on static analysis to obtain information from the code may fail when dealing with highly obfuscated samples and samples that download and execute code at runtime, as mentioned before. Moreover, our evaluation used a significantly larger number of malicious samples than PUMA and DroidMat.

The features used by our system and the one presented by Su et al. [17] have some elements in common. Their system uses the frequency of use of 15 system calls and 9 features from network traffic, whereas our system uses the frequency of use of 90 system calls and also the frequency of use of 74 Android API functions. We argue that the API calls provide important information for the classification. To corroborate that assertion, we performed another experiment, using the same datasets presented before, for training and testing, but this time we used three additional sets of features: the frequency of the 15 syscalls used by Su et al.; the frequency of

Table 6 Comparison of the results obtained by our system with the results presented in related work, showing the number of malicious and benign samples used in the evaluation test, the accuracy obtained and whether the system extracts features statically or dynamically

System	Samples (Mal./Ben.)	Accuracy (%)	Type
DroidMat [19]	238/1,500	97.87	Static
PUMA [14]	249/1,811	86.41	Static
DREBIN [1]	5,560/123,453[a]	93	Static
Su et al. [17]	50/70	99.20	Dynamic
Our system	2,257/1,483	96.82	Dynamic

[a] This is the total dataset used by them, including testing and training. They randomly split the dataset into training (66 %) and testing (33 %), 10 times, and average the results

Table 7 Comparison of the features used by our system with the features used by Su et al. [17]

Feature set	FP	FN	Accuracy (%)
Freq. of API function calls + freq. of system calls	36	89	96.82
Freq. of API function calls + freq. of 15 system calls	39	93	96.62
Freq. of 15 system calls	180	191	89.70
Freq. of API function calls	73	190	93.33

the 15 syscalls used by Su et al. plus the frequency of API calls; the frequency of API calls. The results of this test along with the detection rate obtained by the previous test (the evaluation of our system) are presented in Table 7 and show that using the features related to API calls greatly improved the detection rate.

Besides the classification using 15 system calls, the system presented by Su et al. has also a classifier that uses features extracted from network traffic. This classifier is used to detect malicious samples that were not identified by the first classifier and that match a certain heuristic. From the 191 malicious samples incorrectly labeled by the classifier that used 15 system calls, 150 matched the heuristic used in their work. Considering the best scenario, in which these 150 samples are correctly identified using their classifier that uses network features, the accuracy would be 93.02 %, which is still considerably lower than the value of 96.82 % obtained by our system. This is another evidence of the benefits obtained using the features related to Android API function calls. A possible reason for the accuracy obtained by Su et al. being

Table 5 Values of false-positive (FP), false-negative (FN), true-positive (TP), true-negative (TN), accuracy (A), recall (R), precision (P), harmonic mean (F-measure) and correctly classified samples (CC) obtained in the system evaluation

FP	FN	TP	TN	A	R	P	F-measure	CC
2.43 %	3.94 %	96.06 %	97.57 %	96.82 %	96.06 %	97.53 %	96.79 %	96.66 %

greater in the evaluation test presented in their work is the use of too few samples.

5 Limitations

The main limitations of the developed system are related to shortcomings inherent to dynamic analysis approaches. The analysis system may fail to observe the malicious behavior of samples in some situations, due to problems when gathering resources, to the lack of the necessary stimulation or to the detection of the analysis environment. If, for example, the malware tries to obtain some piece of code from the Internet or tries to connect to a command and control server to get instructions, but the connection fails, the sample may stop executing without performing malicious actions. In addition, the malware sample may execute malicious actions only when certain interactions with the user interface are performed or when certain events, such as receiving an SMS message, occur. If the system fails to simulate these events, the malicious behavior will not be shown. Lastly, malware may detect the analysis environment and stop executing or execute innocuous actions, so the system will not obtain information about them. This detection can be carried out by the identification of virtualized or emulated environment, or the identification of monitoring tools.

6 Conclusions and future work

In this paper we presented a system that uses machine learning to classify Android applications as malicious or benign using information about the use of Android API functions and system calls. To gather the information needed by the detection system, we implemented a dynamic analysis system. To evaluate the capabilities of the detection system, we trained it with 3,780 applications and tested it using 3,740 samples, obtaining an accuracy of 96.82 %. This result was compared to other detection systems, which demonstrated the relevance of our approach.

Future work includes the following: using attributes obtained from the network traffic and attributes obtained statically to enhance the detection capabilities of our system; detecting the evasion of sensitive information using signatures; making a public submission interface available to other researchers and common users, so they can check whether a given application is malicious; developing a non-random way to stimulate the malware using information obtained from the code without the need to modify the Android OS.

Acknowledgments Part of the results presented in this paper were obtained through the project "Evaluation and prevention of security vulnerabilities in smartphones and tablets", sponsored by Samsung Electronics da Amazônia Ltda., in the framework of law No. 8,248/91.

References

1. Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of android malware in your pocket (2014)
2. Blasing, T., Batyuk, L., Schmidt, A.D., Camtepe, S.A., Albayrak, S.: An android application sandbox system for suspicious software detection. In: 2010 5th International Conference on Malicious and Unwanted Software (MALWARE), pp. 55–62. IEEE (2010)
3. DroidBox: Android application sandbox. https://code.google.com/p/droidbox/ (2011)
4. Elish, K.O., Yao, D., Ryder, B.G.: User-centric dependence analysis for identifying malicious mobile apps. In: Workshop on Mobile Security Technologies (2012)
5. Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., Sheth, A.: Taintdroid: an information-flow tracking system for real-time privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, pp. 1–6. USENIX Association (2010)
6. Felt, A., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A survey of mobile malware in the wild. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 3–14. ACM (2011)
7. Gartner: Gartner says worldwide sales of mobile phones declined 3 percent in third quarter of 2012; smartphone sales increased 47 percent. http://www.gartner.com/newsroom/id/2237315 (2012)
8. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: scalable and accurate zero-day android malware detection. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, pp. 281–294. ACM (2012)
9. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.: The weka data mining software: an update. ACM SIGKDD Explor. Newsl. **11**(1), 10–18 (2009)
10. iSecLab: Andrubis: a tool for analyzing unknown android applications. http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/ (2012)
11. Juniper: Juniper networks mobile threat center third annual mobile threats report: March 2012 through March 2013. http://www.juniper.net/us/en/local/pdf/additional-resources/3rd-jnpr-mobile-threats-report-exec-summary.pdf (2013)
12. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Computer Security Applications Conference, 2007, ACSAC 2007, Twenty-Third Annual, pp. 421–430. IEEE (2007)
13. Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., Vigna, G.: Execute this! analyzing unsafe and malicious dynamic code loading in android applications. NDSS **14**, 23–26 (2014)
14. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P.G., Alvarez, G.: Puma: Permission usage to detect malware in android. In: CISIS/ICEUTE/SOCO Special Sessions, pp. 289–298 (2012)
15. Spreitzenbarth, M.: The Evil Inside a Droid—Android Malware: past, present and future. In: E.F.S. Institute (ed.) Proceedings of the 1st Baltic Conference on Network Security & Forensics, pp. 41–59 (2012)
16. Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., Hoffmann, J.: Mobile-sandbox: having a deeper look into android applications. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp. 1808–1815. ACM (2013)
17. Su, X., Chuah, M., Tan, G.: Smartphone dual defense protection framework: detecting malicious applications in android markets. In: 2012 Eighth International Conference on Mobile Ad-hoc and Sensor Networks (MSN), pp. 153–160. doi:10.1109/MSN.2012.43 (2012)

18. VRT: Changing the imei, provider, model, and phone number in the android emulator. http://vrt-blog.snort.org/2013/04/changing-imei-provider-model-and-phone.html (2013)
19. Wu, D.J., Mao, C.H., Wei, T.E., Lee, H.M., Wu, K.P.: Droidmat: Android malware detection through manifest and api calls tracing. In: Seventh Asia Joint Conference on Information Security (Asia JCIS). doi:10.1109/AsiaJCIS.2012.18 (2012)
20. Yan, L.K., Yin, H.: Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21st USENIX Conference on Security Symposium. USENIX Association (2012)
21. Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., Zou, W.: Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 93–104. ACM (2012)
22. Zheng, M., Sun, M., Lui, J.C.: Droidanalytics: a signature based analytic system to collect, extract, analyze and associate android malware. In: Proceedings of The 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 13) (2013)
23. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: Proceedings of the 33rd IEEE Symposium on Security and Privacy (2012)
24. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium (2012)