Android Malware Detection Approaches in Combination with Static and Dynamic Features

Ming-Yang Su, Jer-Yuan Chang, and Kek-Tung Fung
(Corresponding author: Ming-Yang Su)

Department of Computer Science and Information Engineering, Ming Chuan University
5 De Ming Rd., Gui Shan District, Taoyuan City 333, Taiwan
(Email: minysu@mail.mcu.edu.tw)

(Received Sept. 5, 2018; Revised and Accepted Feb. 21, 2019; First Online July 30, 2019)

Abstract

This paper presents two approaches in which static features are combined with dynamic features and used to identify unknown Android malware after proper training by Weka, a well-known machine learning tool. Static features are derived by parsing the test APK after decompilation; they include permissions, sensitive function calls, native-permissions, and priority APK settings. Dynamic features are obtained by parsing an emulator log file after running the test app in the emulator, identifying important activities such as sending short messages (SM) without a user's consent, modifying system files, reading personal contact information, etc. Since the static features can be obtained quickly by parsing the decompiled APK, but dynamic features cannot be obtained in real-time, this study proposes two approaches, ModeA and ModeB, which make efficient use of both types of features. ModeA is a two-tier framework which uses static features for the first tier, and dynamic features for the second tier. Thus, the first tier can be run on a mobile device in real-time, and if the tested app is suspicious then the system can move to the second tier for dynamic feature analysis. ModeB is an off-line system in which static features are merged with dynamic features to measure a test app. ModeB can achieve an overall accuracy of 97.4% for the best case, with ten-fold cross validation in the experiments. The technique of n-fold cross validation, such as n = 2 or 10, is applied to demonstrate the performance of a system for detecting unknown malware.

Keywords: Android Emulator; Android Phones; Dynamic Features; Static Features; Ten-Fold Cross Validation; Weka

1 Introduction

According to the Symantec's annual security report published in March 2018 [25], there were 27K new variants of mobile malware detected in 2017, which was an increase of 54 percent from 17K in 2016. Similarly, a report on malicious mobile software evolution released by Kaspersky in February 2017 [27] showed that about 8.5 million malicious apps were found in 2016, which was 3 times as many as that in 2015. Android smartphones occupy the majority of the market, and are more prone to attacks due to their open source nature. This paper therefore focuses on Android platforms. Android app developers can use either Java, or C/C++ via Android NDK [4] to develop their applications. This convenience has unfortunately resulted in a rapid increase in malware. App developers upload their apps to the official Google Play store, or some unofficial markets, such as Apkpure [5]. Neither, however, has established an effective method of preventing the spread of malware. In addition, numerous free apps contain embedded ad modules for advertising, which makes smartphone security issues even worse. By April 2017, the global accumulative number of Android malware programs reached 19.5 million according to the AV-TEST security report [8]. However, antivirus software normally uses their virus definitions for matching, so when a new malware occurs, the antivirus software may fail to detect it in time. According to the research conducted by Apvrille and Strazzere [7], it takes, on average, three months to spot a new malware in the wild.

Antivirus company Trend Micro selected known malicious apps, classified them, and found that most of them fell into the malware families [26]. This study also showed that the most common behavior of malware is the SMSREG family; all malicious apps belonging to this type carry out malicious activities via Short Message Service (SMS). They steal a user's personal data and send it to a server via short messages, or deliver malware download links via SMS. The second most common malware type was found to be the FAKEINST family, which sends unauthorized short messages to a specific number to register unsolicited, unwanted expensive services for the user, without their consent. Therefore, sending short messages by an app automatically could be regarded as a possible contribution to being classified as malware. In this study,

sending short messages is considered a dynamic feature when analyzing the emulator logs. This study combines static and dynamic app features, and uses Weka to design two malware detection approaches, called ModeA and ModeB, respectively.

Static features are extracted from the app, which are permissions, native-permissions, sensitive function calls and priority-setting values. Dynamic features are obtained by executing the app in a sandbox emulator, and then extracting the activities from the log file of the emulator, such as sending short messages. For ModeA, a two-tier mechanism is proposed in which the first tier employs static features, and the second tier employs dynamic features. For ModeB, all static and dynamic features are merged together to provide a design with higher detection accuracy. Furthermore, a feature weighting strategy is also applied to improve the overall accuracy of ModeB.

The remainder of this paper is arranged as follows. Section 2 offers a literature review, Section 3 describes the framework of ModeA and its related experiment results, Section 4 describes ModeB and its related experiment results, and Section 5 offers some conclusions.

2 Related Researches

Some studies of static, dynamic, or hybrid detections of Android malware are reviewed in this Section. Normally, static analysis relies on features extracted from the app without executing code, while dynamic analysis extracts features based on execution on an emulator.

2.1 Static Detection

Static analysis is a traditional malware detection method for computers, mostly applicable to smart phones. Samra *et al.* [21] used clustering with information retrieval (IR) to identify Android malware. The authors extracted the features of the apps from their XML-files, which declare permissions requested by apps, and then used the Weka K-Mean algorithm for classification. In the paper, the dataset of 18,174 Android apps consisted of 4,612 instances of business and 13,535 instances of tools; the experimental results show that the recall and precision were both 0.71. Liu *et al.* [15] proposed a two-layered permission-based detection scheme for detecting malicious Android apps. The authors considered apps requesting permission pairs as an additional condition, and also considered used permissions to improve detection accuracy.

Zhao and Qian [29] suggested that most malware variants were created by automatic tools, and thus there are special fingerprint features for each malware family. The authors decompiled the Android APK, and mapped the three different kinds of features, Opcodes, API packages and high level risky API functions, to three integrated channels of an RGB image, respectively. They then adopted neural networks to identify each family's features. The experimental results showed that the proposed method successfully identified all 14 malware datasets with an accuracy of 90.67% on average. Fereidooni *et al.* [11] proposed a system called ANASTASIA, which detected a malicious Android app by statically analyzing its behaviors. The authors utilized a large number of statically extracted features from various security behavioral characteristics of an app. A detection framework was built based on machine learning with a high performance detection rate and an acceptable false positive rate. The authors then evaluated the performance on a large-scale malware data-set, including 18,677 malware and 11,187 benign apps, and the results showed a true positive rate of 97.3%, and a false negative rate of 2.7%.

Maier *et al.* [17] also described using obfuscation techniques to bypass static analysis via modifying partial program codes to avoid being similar to known malware samples. They tested and evaluated several antivirus utilities which were able to efficiently identify known malware, but had little success detecting malware after obfuscation. This means that malware detection cannot only rely on static analysis. Some researchers have therefore begun to develop dynamic detection techniques, or combinations of dynamic and static detection techniques.

2.2 Dynamic Detection

Dynamic analysis is based on the behaviors of an application, i.e., the application must be installed and executed in an emulator, and then the log file is analyzed to determine if the application is suspicious. Sun *et al.* [24] indicated that sandbox environments play an important role in the field of information security. A sandbox can execute malware in an isolated environment, minimizing its destructive power, and can test the malware for different ways to find its main intentions. Bhatia and Kaushal [9] presented an approach to perform dynamic analysis of Android apps to classify them as malicious or non-malicious. The authors developed a system which collects and extracts the system call traces of all apps during their runtime interactions with the phone platform. Subsequently all the collected system call data is aggregated and analyzed to detect and classify the behavior of Android applications.

Singh and Hofmann [22] extracted the system call behavior of 216 malicious apps and 278 normal apps to construct a feature vector for training a classifier. The authors applied several classification algorithms to the dataset, including decision tree, random forest, gradient boosting trees, k-NN, Artificial Neural Network, Support Vector Machine and deep learning. Furthermore, three feature ranking techniques, i.e., information gain, Chi-square statistic, and correlation analysis, were used to select appropriate features from the set of 337 system calls. Experiments showed that Support Vector Machines (SVM), after selecting features through correlation analysis, outperformed other techniques, where an accuracy of 97.16% was achieved. Maier *et al.* [18] demonstrate that

Android malware can bypass current automated analysis systems, including AV solutions, mobile sandboxes, and the Google Bouncer. The authors found that malware can either behave benignly or load malicious code dynamically at runtime. They also investigated the frequency of dynamic code loading among benign and malicious apps, and found that malicious apps make use of this technique more often. About one third of 14,885 malware samples were found to dynamically load and execute code, which means traditional antivirus tools can't detect these kinds of malware.

2.3 Hybrid Detection and Other Methods

Qian *et al.* [19] proposed an approach with two steps using static and dynamic analysis separately in each step. The first step used static analysis, and a permission combination matrix was used to determine the risk of the app. For suspicious apps, based on reverse engineering, the authors planted Smali code to monitor sensitive APIs such as sending SMS, accessing user location, device ID, phone number, etc. Their experiment results showed that almost 26% of apps in the Android market have privacy leakage risks. Kapratwar *et al.* [14] suggested that static analysis is more efficient, while dynamic analysis can be more informative, particularly in cases where the code is obfuscated. In this research, the authors applied machine learning techniques to analyze the relative effectiveness of particular static and dynamic features for detecting Android malware.

They also carefully analyzed the robustness of the scoring techniques under consideration. Liu *et al.* [16] decompiled an app to obtain static features by searching the permissions used by the app from the AndroidManifest file, and the APIs from the Smali file. The app was also installed in an emulator to obtain dynamic features from its behaviors. Finally, the static and dynamic vectors were merged into a machine learning system for classification.

Kang *et al.* [13] proposed a method to improve the performance of Android malware detection by incorporating the creator's information as a feature, and classifying malicious applications into similar groups. The proposed system enables fast detection of malware by using creator information such as certificate serial numbers. Additionally, it analyzes malicious behaviors and permissions to increase detection accuracy. The system can also classify malware based on similarity scoring. Its detection rate and accuracy are 98% and 90%, respectively. The MobileSandbox system proposed by Spreitzenbarth *et al.* [23] combines static and dynamic analysis, i.e., the results of static analysis are used to guide dynamic analysis and extend the coverage of executed code. It also uses specific techniques to log calls to native (i.e., "non-Java") APIs, and finally combines these results with machine-learning techniques to classify the analyzed samples as either benign or malicious. Rodriguez-Mota *et al.* [20] proposed a hybrid test framework in which dynamic analysis was implemented after the static analysis of an app. They also analyzed Trojans, and found common features for instances. These features can be used for static analysis to increase classification accuracy.

This study proposes two approaches using static and dynamic features to identify malware on Android platforms. The first, called ModeA, is a two-tier system. ModeA uses static features for the first tier, which can be run in real-time, and uses dynamic features for the second tier. ModeB, is an offline system in which static features are merged with dynamic features to measure a test app. ModeB was able to achieve an overall accuracy of 97.4% for unknown malware in the experiments. The two approaches are introduced in Sections 3 and 4, respectively.

3 System Approach: Mode A - A Two-tier Design

This approach is a two-tier framework, with static and dynamic analysis, operated with an on-line analysis tool, VirusTotal [28], as assistance. The whole structure of this approach is illustrated in Figure 1. It first implements a static analysis in the user's phone, and if the test application is identified as a suspicious app, it is uploaded to the sandbox server for further dynamic analysis. In terms of static analysis, the proposed system extracts the permissions, native-permissions, intent-priority setting, and function calls from the test app. After the app (APK format) is decompiled by Apktool [6], two important files, AndroidManifest.xml and classes.dex, can be obtained. The AndroidManifest.xml contains three kinds of features: permissions, native-permissions and priority settings; the classes.dex contains function calls. Dynamic analysis requires a sandbox server, so that the test app can be uploaded to the server for further analysis. The dynamic analysis uses an Android emulator to run the suspicious app, and the behavior pattern is then extracted to check whether the application contains malicious activities.

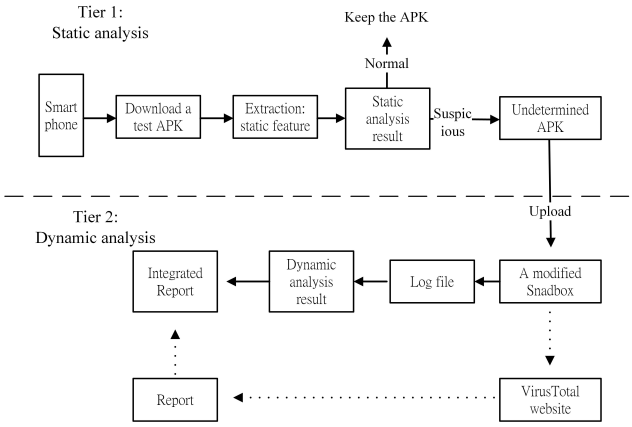


Figure 1: System structure of Mode A: A two-tier design

3.1 Static Analysis

First, the test app is decompiled to obtain static features, which include permissions, native-permissions, intent-priority and sensitive functions. The permission [1] is a security design on Android platforms. If an app wants to execute some specific functions, corresponding permissions must be declared in the AndroidManifest.xml file, and shown to the user before installation so that users are aware of the activities of an app by permission declaration. In this study, the 135 permissions provided by Android 5.0 were considered.

Native-permission is provided by Google to enhance Android programming by supporting other languages, like C/C++, other than JAVA to develop applications. While this flexibility extends Android's supportability, malware developers use this facility to insert malicious codes into applications, or to disguise malicious programs as normal applications, hiding their intentions from the user.

Intent-priority is also declared in Android Manifest.xml, representing the intent-priority of program activity. For example, if the intent-priority value of Application A is larger than that of Application B, related messages are sent to A before they are sent to B. The intent-priority value is preset as 0, and its numerical range is -1000 to 1000. This study found that malware normally sets its intent-priority value higher than normal programs on purpose, so as to make sure the malware receives information first. The value of normal programs usually does not exceed 100.

The final feature of static analysis is sensitive function calls. This study analyzes how many times an application uses sensitive functions as a feature of static analysis. Figure 2 shows partial program codes after a malicious app is decompiled. The malware uses the sendTextMessage( ) function, and sends short messages to a specific number when it is started, and uses the setComponent( ) function to start up another malware. Table 1 lists some of the 59 sensitive functions concerned in this study.

3.2 Dynamic Analysis

In order to carry out dynamic analysis, a sandbox server is built, so that a user can upload an app via mobile phone interface to this server for runtime testing, and then the log file of the emulator can be analyzed to check whether malicious activities are present. This system also uploads the test application to VirusTotal [28] for testing. VirusTotal is a free on-line scanning website. Finally, the system returns the dynamic analysis result and VirusTotal on-line test result to the user. The sandbox environment in this study is an Android virtual machine (Android emulator) [2] provided by Google, which analyzes the activity log in the active stage of applications to identify suspicious activities. MonkeyRunner [3] is an automated testing tool provided by Google, developed based on the Python language. MonkeyRunner is provided via API for developers to write script, after which it sends commands to the Android device to "simulate trigger events". This study prerecords several scripts to simulate user behavior patterns, and MonkeyRunner starts the scripts when an application is running in the Android emulator.

The information exported from an Android system to the log file of the emulator in the preset environment is not specific enough, however. For example, in sending a short message, only the activation of the function is recorded, but the receiving number and the message content are not. In order to detail the information exported to the log file, the emulator's files needed to be modified first in this study. The files comprising the emulator are extracted from the image file "system.img". All functional files are in the .jar format. After decompression, a jar file can generate a classes.dex file, which contains some functional files of the emulator. Then, the de-compilation tool Dex Manager [12] is required, which decompiles a classes.dex file to a smali program code file. After modification, the modified program code (in smali) can then be packaged into classes.dex. Finally, the modified classes.dex is repackaged by compression/decompression tool to the .jar format in order to run the emulator. Take sending a short message (SM) as an example. The function for sending an SM is stored in the SmsManager of telephony-common.jar. The default Android emulator only records the event that the function has been activated, but the content of the message and the recipient of the message are omitted. Modification (see the red frame in the upper part of Figure 3) enables the emulator to record the receiving number and message content in the SendTextMessage file. The yellow frame in the middle of Figure 3 shows that in the log the receiving number and short message content are actually recorded when the SendTextMessage is started. Similarly, Figure 4(a) shows that the short message sending function has been activated, the recipient phone number is 81168, and the content of the message is "SP99". Another example in the log is given in Figure 4(b), where a rename function has been executed, with the names before and after the change. In this study, twelve functions given in Table 2 in the default Android emulator were modified first in order to record more information when they were activated, as if more information about the activities is recorded, then more sophisticated determinations can be performed.

3.3 Dataset and Experimental Results

In this study, normal apps were downloaded, for the most part, from Google Play, and checked by anti-virus software, while malicious apps were mostly obtained from the Contagio Malware [10] website, which periodically updates and shares malware. In total, 900 normal programs and 300 malware programs were applied in the experiments. Thus 1,200 feature vectors representing 1200 apps were obtained and fed into Weka, a machine learning tool, for classification. An instance of one static feature vector is shown in Figure 5. The permissions (black), function calls (brown), and native-permissions (purple) and are set

```
f (paramIntent.getAction().equals("android.intent.action.PACKAGE_ADDED"))

SmsManager.getDefault().sendTextMessage("18670259904", null, " TroJan Ok", null, null);
if (paramIntent.getDataString().substring(8).equals("com.example.com.android.trogoogle"))
{
  Intent localIntent = new Intent();
  localIntent.setComponent(new ComponentName("com.example.com.android.trogoogle",
                                             "com.example.com.android.trogoogle.MainActivity"));
  localIntent.setAction("android.intent.action.VIEW");
  startActivity(localIntent);
```

Figure 2: Illustrations of sensitive functions in applications

Table 1: Some of the 59 Sensitive functions

ContentResolver;->query ;	PackageManager;->installPackage ;
Camera;->open ;	IActivityManager$Stub$Proxy;->shutdown ;
MediaRecorder;->setAudioSource ;	Downloads$ByUri;->startDownloadByUri ;
PowerManager;->reboot ;	Telephony$Mms;->query ;
SmsManager;->sendTextMessage ;	ContentService;->dump;
BluettoothSocket;->connect;	ActivityManagerNative;->restartPackage ;

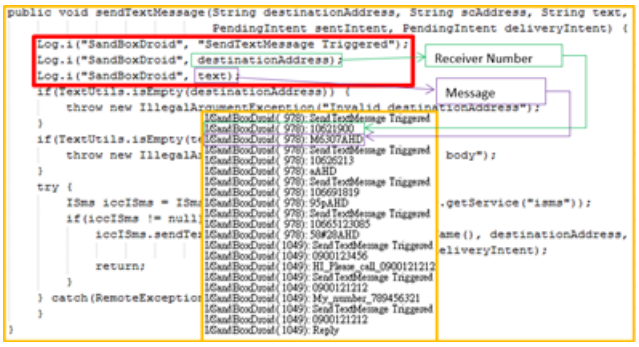
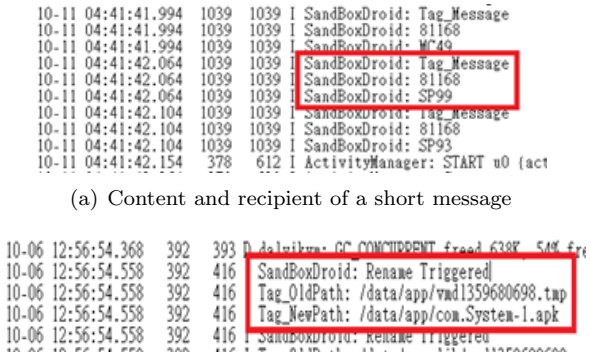


Figure 3: System function modifications



(a) Content and recipient of a short message

(b) The names of a file before and after rename

Figure 4: More information contained in the log of the emulator

Table 2: Modified functions in the Android emulator

1. Turn off background apps	2. Send short messages
3. Execute commands	4. Get GPS data
5. Get device ID	6. Get phone number
7. Turn on camera	8. Delete files
9. Rename files	10. Open files
11. Copy files	12. Retrieve app information

as 1 if they are used, and set as 0 if they are not used. In terms of priority (green), a priority greater than 0 and smaller than 1000 (relatively normal) is set as 0, and one that is less than 0 or greater than 1000 (relatively abnormal) is set as 1. The final feature tells Weka whether the app is malware; "yes" is normal and "no" is malicious.

Table 3 shows the experiment results by Weka. "Non-split" means that all apps are used for training, and tested as well. The n-fold cross-validation (n = 2 or 10) means that all data are divided into n equal parts; n - 1 parts are used for training, and the remainder are used for testing, repeated n times with a different part used each time. Finally, the average value is shown. In terms of accuracy, the best data obtained by non-split, two-fold and ten-fold all occur when the SVM algorithm is used; the values are 93.4%, 89.8% and 91.1%, respectively. Non-split has the maximum value, because the training and test are the same (complete) dataset. The ten-fold experiment yields better data than the two-fold experiment for larger

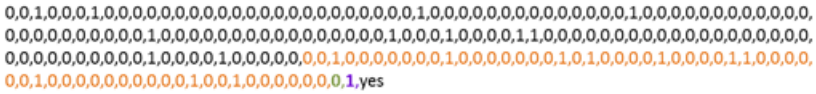


Figure 5: Example of one static feature vector

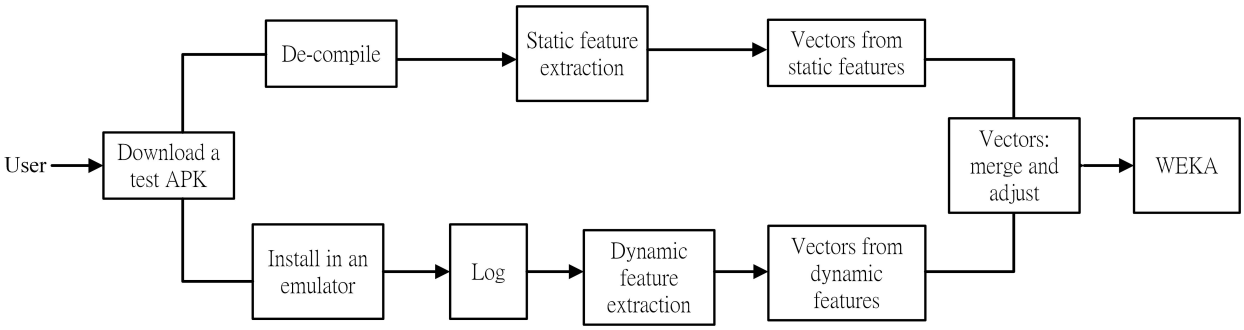


Figure 6: System structure of Mode B: A mixed features design

training sets. Normally, non-split is used to measure the known-attack detection performance of a defense mechanism, and n-fold is used to evaluate its unknown attack detection performance. Like most cases in the fields, both n=2 and n=10 are considered in this paper.

Table 3: Experimental results obtained using static features only

Algorithm \ Result		TP rate	FP rate	Accuracy
Bayes Net	non-split	0.794	0.170	0.803
	two-fold	0.791	0.183	0.798
	ten-fold	0.793	0.167	0.803
Naïve Bayes	non-split	0.781	0.143	0.800
	two-fold	0.787	0.153	0.802
	ten-fold	0.783	0.150	0.800
K-NN	non-split	0.998	0.293	0.925
	two-fold	0.991	0.420	0.888
	ten-fold	0.988	0.380	0.896
J48	non-split	0.997	0.287	0.926
	two-fold	0.983	0.393	0.889
	ten-fold	0.989	0.360	0.902
SVM	non-split	1.000	0.263	0.934
	two-fold	0.970	0.320	0.898
	ten-fold	0.982	0.303	0.911

In terms of dynamic analysis, 30 normal programs and 70 malware programs were selected randomly to form the test set. These programs were installed in the emulator in turn and executed, and MonkeyRunner was started to simulate user behavior. The log was imported into the analysis tool developed in this study, so as to identify suspicious activities. As mentioned in Section 1, sending short messages (SM) without user's awareness is an important feature regarded as malware. In the experiments, the test app was installed in the modified emulator without executing MonkeyRunner [3], *i.e.*, if any SM is sent out, it must be sent by the app without user consent, because no-one is operating the app. Thus far, this the only the criterion for ModeA that has been considered, i.e., if an app sends an SM without user consent, it is regarded as malware. Of the 70 malware programs, 41 applications included such malicious actions. Unreported malware did not trigger the short message sending function, and the 30 normal programs were all identified as having normal behavior. According to the analysis result, as long as an application has an automatic short message sending function, it will be successfully detected in this way. However, malware which executes other attack behaviors (aside from SM sending attacks) cannot be detected. In the following section, approach ModeB merges all static and dynamic features to provide a more precise malware detection design.

4 System Approach: Mode B - All Features Combined, and Feature Weighting

ModeB merges static and dynamic app features, and adjusts their weights appropriately. Similarly, the static features include permissions, native-permissions, functions and priority settings in the application; the dynamic features are obtained by executing the app in the emulator, and then extracting important activities from logs. The overall structure of ModeB is shown in Figure 6. The static features are obtained as in ModeA. Figure 7 shows the tool designed in this study for extracting dynamic features from the log.

After merging the 12 dynamic features with the static features, a feature vector of the app is shown in Figure 8(a), in which the red part, also underlined, represents dynamic features. Table 4 shows the experiment results obtained using Weka. In terms of accuracy, the best experimental results for non-split, two-fold, and ten-fold occurred when the SVM algorithm was used; their values are 95.3%, 92.3% and 93.9%, respectively. Again, non-split obtains the maximum value due to having the same (complete) dataset for training and testing. The ten-fold experiment has better results than the two-fold experiment because of its larger training sets.

Table 4: Experimental results of merging static and dynamic features

Algorithm \ Result		TP rate	FP rate	Accuracy
Bayes Net	non-split	0.787	0.150	0.803
	two-fold	0.790	0.157	0.803
	ten-fold	0.782	0.160	0.797
Naïve Bayes	non-split	0.772	0.143	0.793
	two-fold	0.773	0.143	0.794
	ten-fold	0.776	0.147	0.795
K-NN	non-split	0.998	0.210	0.946
	two-fold	0.992	0.390	0.897
	ten-fold	0.989	0.293	0.918
J48	non-split	0.997	0.227	0.941
	two-fold	0.983	0.317	0.908
	ten-fold	0.989	0.277	0.923
SVM	non-split	1.000	0.190	0.953
	two-fold	0.978	0.243	0.923
	ten-fold	0.991	0.217	0.939

Since there is a large gap between the numbers of static and dynamic features, this study used a weighting method to mitigate the effect resulting from that disparity. The most frequently used feature was given the highest weight, while the other features were compared with the highest feature, and given their corresponding weights. For example, if Feature A has the highest frequency of use, its value is 1000, and the maximum weight is 20 in this paper, so the frequency of use of A is divided by 50 to obtain the weight 20. The other features can be deduced by analogy, divided by 50, and rounded off to obtain their respective weights. The number of dynamic features is quite small in relation to that of the static features, so the maximum weight, i.e. 20, is given to any feature once it occurs. Figure 8(b) shows a feature vector of the app after weight adjustment. The experiment results from Weka are shown in Table 5. The SVM algorithm results in the best accuracy in the three experiments (non-split, two-fold and ten-fold), and their values are 99.5%, 96.3% and 97.4%, respectively. It is clear that the results are improved by adjusting the feature weights. Figure 9 shows the best outcome of the ten-fold experiments using SVM in Weka. The other outcomes of the ten-fold experiments using Bayes Net, Naïve Bayes, K-NN, and J-48 in Weka are given in Figures 10(a), 10(b), 10(c) and 10(d), respectively.

Table 5: Experimental results for weighted features

Algorithm \ Result		TP rate	FP rate	Accuracy
Bayes Net	non-split	0.876	0.180	0.862
	two-folds	0.888	0.187	0.869
	ten-folds	0.878	0.187	0.862
Naïve Bayes	non-split	0.88	0.183	0.864
	two-folds	0.881	0.180	0.866
	ten-folds	0.882	0.183	0.866
K-NN	non-split	0.998	0.02	0.993
	two-folds	0.984	0.187	0.942
	ten-folds	0.981	0.103	0.96
J48	non-split	0.996	0.017	0.993
	two-folds	0.970	0.153	0.939
	ten-folds	0.971	0.110	0.951
SVM	non-split	1.000	0.020	0.995
	two-fold	0.976	0.077	0.963
	ten-fold	0.983	0.053	0.974

Finally, a comparison between the proposed approach and other researches is given below. First, it must be noted that different researches used different methods of showing their performances, with different datasets. There is thus no unanimously fair way to compare them from a specific aspect. The performances of the aforementioned researches in Section 2.3 are summarized because they also used a hybrid approach. The work of [19] was to monitor the information leakage of apps, and the main result is that by experiments, almost 26% applications in the Android market have privacy leakage risks. In [14], the authors used a small dataset with 103 malware and 97 benign apps. They evaluated the performances of static analysis and dynamic analysis separately, instead of merging both types of features together. The authors adopted the term ROC curve, not accuracy, to show the performances of static analysis and dynamic analysis, and

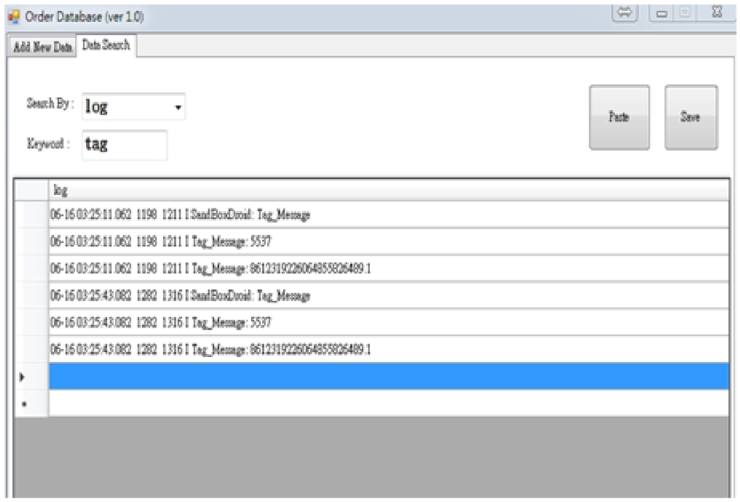


Figure 7: Parsing tool designed to extract dynamic features from the log

0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,1,0,0,0,0,
0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,yes

(a) An example feature vector

0,0,7,0,0,0,20,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,0,0,0,0,0,0,0,0,0,0,0,0,0,0,20,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,20,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6,0,0,0,13,0,0,0,0,4,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,17,0,0,0,0,0,0,0,5,0,0,0,0,0,0,0,0,20,0,0,0,0,0,0,0,16,0,8,0,0,0,0,7,0,0,0,0,
9,12,0,0,0,0,0,0,15,0,0,0,0,0,0,0,0,0,0,0,7,0,0,4,0,0,0,0,0,0,0,20,0,0,0,20,0,0,0,0,0,20,0,yes

(b) An example feature vector after adjustment

Figure 8: Feature vector for one app after merging static and dynamic features

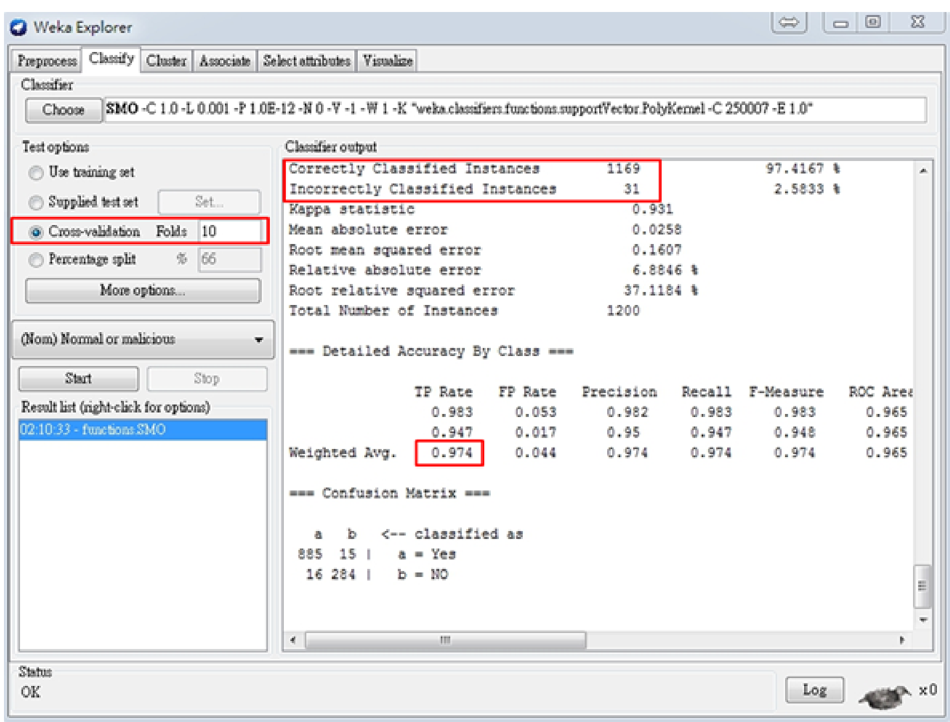
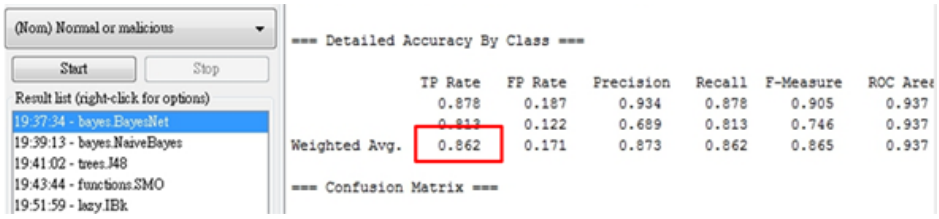
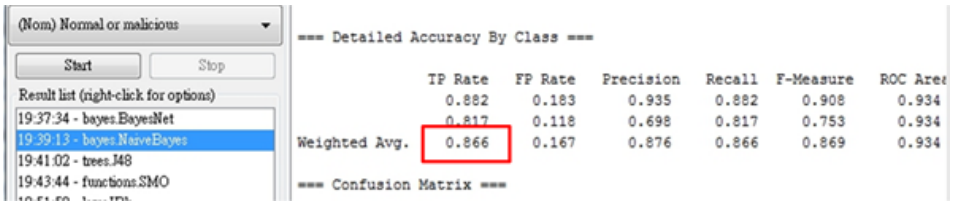


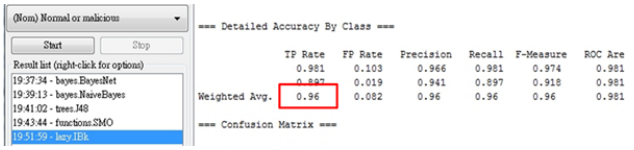
Figure 9: The best outcome of the ten-fold experiments using SVM in Weka



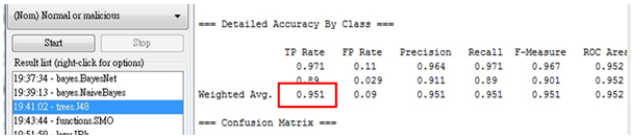
(a) Bayes Net



(b) Naïve Bayes



(c) K-NN



(d) J-48

Figure 10: Outcomes of the ten-fold experiments using different algorithms in Weka

their best results obtained by 10-fold cross-validation were 0.966 and 0.884, respectively, both using the RF100 algorithm. [16] is a short, 2 page paper, in which the authors took 3,414 static features from permissions and API features, and 345 dynamic features from emulator log files. Using a dataset containing 500 malicious apps and 500 benign apps, the authors achieved the best results, in terms of accuracy, for static detection and dynamic detection using SVM (99.28%) and Naïve Bayes (90.00%), respectively. However, the authors did not mention what kind of experimental techniques were adopted, like non-split, 2-fold, 10-fold, etc. In addition, they did not merge static and dynamic features together and resolved to do this in future work.

In [13], the authors proposed an Android malware detection and classification system based on static analysis using serial number information from the certificate as a feature. As a result, the detection system can achieve 98% accuracy, and the classifier module can classify the 20 kinds of malware families with 90% accuracy. The work of [23] did not focus on the classification of malicious and benign apps. Instead, the authors focused on the calls to native (i.e., "non-Java") APIs in apps, because of the potential risks of such calls. They evaluated the system on more than 69,000 apps from Asian third-party mobile markets, and found that about 21% of them actually use native calls in their code. In [20], the authors took 39 trojan Android malware as samples to illustrate the effectiveness of the proposed method. No result regarding classification of a dataset was given.

5 Conclusion

Security mechanisms for Android platforms are fast becoming an important and urgent issue. Current malware technology changes quickly, and malware using obfuscation cannot be identified using only static analysis. Therefore, the malware detection approaches proposed in this paper combine static and dynamic features. Since static features can be obtained by parsing the decompiled test SDK, while dynamic features need to be done with an emulator, two approaches, ModeA and ModeB, are proposed to apply the use of static and dynamic features. ModeA is a two-tier framework in which the first tier can be run on a mobile device to obtain the static features of a test SDK and make a decision in real-time; if necessary, the tested SDK can be uploaded to a server for the second tier check by dynamic features. In ModeB, static features and dynamic features were merged, and since the static features far outnumbered the dynamic features, weights of features were also adjusted to address the disparity. According to the Weka experiments, the overall accuracy values achieved by ModeB for detecting known (non-split) and unknown (ten-fold) malware were 0.995 and 0.974, respectively, both obtained by SVM algorithm.

Acknowledgments

This work was partially supported by the Ministry of Science and Technology, Taiwan, with contracts: MOST 105-2221-E-130-004, MOST 106-2221-E-130-002 and MOST 107-2221-E-130-003.

References

[1] Android, *Android Developer Permission*, June 27, 2019. (https://developer.android.com/reference/android/Manifest.permission.html)

[2] Android, *Android Virtual Device*, June 27, 2019. (https://developer.android.com/studio/run/emulator.html)

[3] Android, *MonkeyRunner*, June 27, 2019. (http://developer.android.com/tools/help/monkeyrunner_concepts.html)

[4] Android, *Android Native Code*, June 22, 2016. (https://developer.android.com/tools/sdk/ndk/index.html)

[5] Apkpure, *Download free Android Games and Android Apps*, June 27, 2019. (https://apkpure.com)

[6] Apktool, *A Tool for Reverse Engineering Android apk Files*, June 27, 2019. (http://ibotpeaches.github.io/Apktool/)

[7] A. Apvrille and T. Strazzere, "Reducing the window of opportunity for Android malware gotta catch'em all," *Journal in Computer Virology*, vol. 8, issue 1-2, pp. 61-71, 2012.

[8] AV-TEST, *Security Report 2016/17*, 2017. (https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2016-2017.pdf)

[9] T. Bhatia and R. Kaushal, "Malware detection in android based on dynamic analysis," in *International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pp. 1-6, 2017.

[10] Contagio mobile, *Android Fakebank Samples*, Mar. 20, 2018. (http://contagiominidump.blogspot.com/)

[11] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "ANASTASIA: android malware detection using static analysis of applications," in *8th IFIP International Conference on New Technologies, Mobility and Security (NTMS'16)*, pp. 1-5, 2016.

[12] Jasi2169_TeamURET, *Dex Manager v1.1 - Designed To Play With Classes.dex*, Dec. 30, 2014. (http://forum.xda-developers.com/android/software/tool-dex-manager-v1-0-designed-to-play-t2988532)

[13] H. Kang, J. W. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *International Journal of Distributed Sensor Networks*, online published in Jan. 2015.

[14] A. Kapratwar, F. D. Troia, and M. Stamp, "Static and dynamic analysis of android malware," in *3rd International Conference on Information Systems Security and Privacy (ICISSP'17)*, pp. 653-662, 2017.

[15] X. Liu and J. Liu, "A two-layered permission-based android malware detection scheme," in *IEEE International Conference on Mobile Cloud Computing, Services and Engineering*, pp. 142-148, 2014.

[16] Y. Liu, Y. Zhang, H. Li, and X. Chen, "A hybrid malware detecting scheme for mobile android applications," in *IEEE International Conference on Consumer Electronics (ICCE'16)*, pp. 155-156, 2016.

[17] D. Maier, T. Muller, and M. Protsenko, "Divide-and-conquer: why android malware cannot be stopped," in *9th International Conference on Availability, Reliability and Security (ARES'14)*, pp. 30-39, 2014.

[18] D. Maier, M. Protsenko, and T. Muller, "A game of Droid and Mouse: The threat of split-personality malware on Android," *Computers & Security*, vol. 54, pp. 2-15, Oct. 2015.

[19] Q. Qian, J. Cai, M. Xie, R. Zhang, "Malicious behavior analysis for android applications," *International Journal of Network Security*, vol. 18, no.1, pp.182-192, 2016.

[20] A. Rodriguez-Mota, P. J. Escamilla-Ambrosiot, S. Morales-Ortega, M. Salinas-Rosale, and E. Aguirre-Anaya, "Towards a 2-hybrid android malware detection test framework," in *IEEE International Conference on Electronics, Communications and Computers*, 2016.

[21] A. A. A. Samra, Y. Kangbin, and O. A. Ghanem, "Analysis of clustering technique in android malware detection," in *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS'13)*, pp. 729-733, 2013.

[22] L. Singh and M. Hofmann, "Dynamic behavior analysis of android applications for malware detection," in *International Conference on Intelligent Communication and Computational Techniques (ICCT'17)*, pp. 1-7, 2017.

[23] 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*, vol. 14, no. 2, pp 141–153, 2015.

[24] L. Sun, S. Huang, Y. Wang, and M. Huo, "Application policy security mechanisms of Android system," in *14th IEEE International Conference on High Performance Computing and Communications*, pp. 1722-1725, 2012.

[25] Symantec, *Internet Security Threat Report*, 2018. (https://www.symantec.com/security-center/threat-report)

[26] Tread, *Masque, FakeID, and Other Notable Mobile Threats of 2H 2014*, Jan. 7, 2015. (https://www.trendmicro.com/vinfo/us/security/news/mobile-safety/masque-fakeid-and-other-notable-mobile-threats-of-2h-2014?linkId=11679913)

[27] R. Unuchek, *Mobile Malware Evolution 2016*, Feb. 28, 2017. (https://securelist.com/analysis/kaspersky-security-bulletin/77681/mobile-malware-evolution-2016/)

[28] VirusTotal, *Analyze Suspicious Files and URLs to Detect Types of Malware Including Viruses, Worms, and Trojans*, June 27, 2019. (https://www.virustotal.com)

[29] Y. L Zhao1 and Q. Qian, "Android malware identification through visual exploration of disassembly files," *International Journal of Network Security*, vol.20, no.6, pp.1061-1073, 2018.

Biography

Ming-Yang Su received his B.S. degree from the Department of Computer Science and Information Engineering of Tunghai University, Taiwan in 1989, and received his M.S. and Ph.D. degrees from the same department of the National Central University and National Taiwan University in 1991 and 1997, respectively. He is an IEEE member, and currently a professor of the Department of Computer Science and Information Engineering at the Ming Chuan University, Taoyuan, Taiwan. His research interests include network security, intrusion detection/prevention, malware detection, mobile ad hoc networks, Mobile security and wireless sensor networks.

Jer-Yuan Chang received the M.S. degree in 2016, from the Department of Computer Science and Information Engineering of Ming Chuan University, Taoyuan, Taiwan. His research interests are in the areas of MANET security, Mobile security, intrusion detection and prevention.

Kek-Tung Fung received the M.S. degree in 2014, from the Department of Computer Science and Information Engineering of Ming Chuan University, Taoyuan, Taiwan. His research interests are in the areas of Mobile security and SIP security.