ANDROINSPECTOR: A SYSTEM FOR COMPREHENSIVE ANALYSIS OF ANDROID APPLICATIONS

Babu Rajesh V, Phaninder Reddy, Himanshu P and Mahesh U Patil

Centre for Development of Advanced Computing

ABSTRACT

Android is an extensively used mobile platform and with evolution it has also witnessed an increased influx of malicious applications in its market place. The availability of multiple sources for downloading applications has also contributed to users falling prey to malicious applications. A major hindrance in blocking the entry of malicious applications into the Android market place is scarcity of effective mechanisms to identify malicious applications. This paper presents AndroInspector, a system for comprehensive analysis of an Android application using both static and dynamic analysis techniques. AndroInspector derives, extracts and analyses crucial features of Android applications using static analysis and subsequently classifies the application using machine learning techniques. Dynamic analysis includes automated execution of Android application to identify a set of pre-defined malicious actions performed by application at run-time.

KEYWORDS

Mobile Security, Malware, Static Analysis, Dynamic Analysis, Android

1. INTRODUCTION

Android is a widely used mobile platform and due to its dominance in consumer space, Android becomes a lucrative target for malware developers who are exploiting the popularity and openness of Android platform for various benefits. Malware developers use Android marketplaces as entry points for hosting their malicious applications into the android user space. According to Risk-IQ [1] report, malicious applications in Play store have grown by 388 percent from 2011 to 2013, while the number of such applications removed annually by Google has dropped from 60 percent in 2011 to 23 percent in 2013. As a large number of applications are uploaded and updated regularly on these market places, Manual analysis of all the applications is difficult task. A major hindrance for these market places is a scarcity of effective mechanisms to evaluate the security threats possessed by the mobile applications being uploaded. Though static analysis of Android applications gives a good idea of what an application is capable of, it is the behavioural analysis of the application during it's execution which depicts the exact behaviour of the application and detects if any malicious actions have been performed. Analysis of an application by manually executing it is a cumbersome and error prone process.

In this regard we present 'AndroInspector', a system for comprehensive analysis of an Android application using both static and dynamic analysis techniques. Dynamic analysis component of AndroInspector identifies malicious actions performed during application execution by analysing traces generated at run time. Application execution is carried out by automating the process of test case generation and execution. Static analysis component comprises of extracting various

DOI : 10.5121/ijnsa.2015.7501

crucial features from an Android application, assigning weights to these features and subsequently classifying the application as either malware or benign using a classifier model. The classifier model is trained using the malware data set of 1260 malware samples acquired from Genome Malware Project [2] and popular benign applications obtained from Google Play Store. The model was then tested against 500 malware samples obtained from Virustotal malware intelligence service [3].

2. RELATED WORK

Androguard [4] statically extracts features from APK, but this tool shows high false positive rate. DroidMat [5] combines static and dynamic analysis approaches. It extracts features like permissions and intents using static analysis and API calls using dynamic analysis. Adrieene et al. [6] proposed an approach to identify over privileged applications by comparing API calls invoked with permissions declared in the Manifest. William Enck et al. [7] proposed an approach where a certificate is generated during an application's installation. This certificate gives complete information about the application by rating them using Kirin security rules which are based on the combinations of permissions extracted from Manifest file. DroidAnalytics [8] is a signature based system for detecting repackaged applications. The drawback of this technique is it requires large and balanced data set of malware and benign samples. Shabtai et al. [9] applied machine learning classifier techniques like decision tree, Naive Bayes (NB), Bayesian Networks (BN) etc. to classify Android applications as games and utilities citing the non availability of malware applications. They collected around 22,000 features initially and later reduced to 50 features for the purpose of classification. Classification using AndroInspector's classifier model requires extraction of 24 features from the Android application.

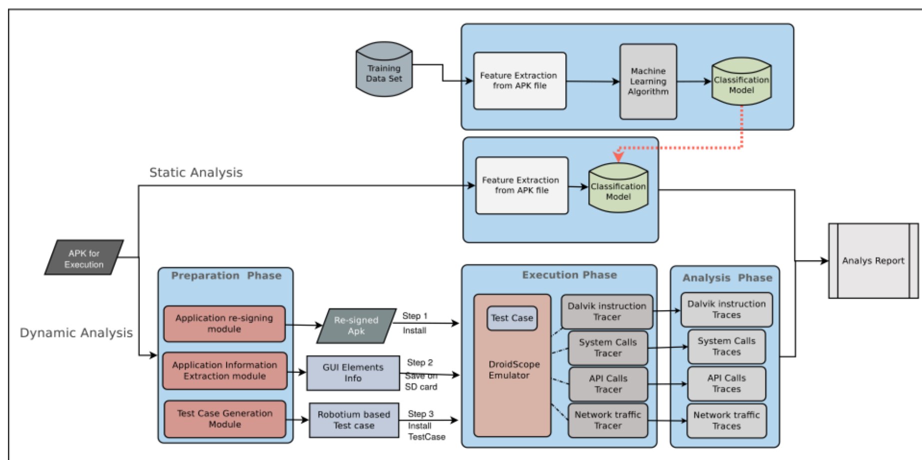
Recently, a lot of work has also been done in the areas of automated Android application execution and dynamic analysis of Android application. Automated android application execution tools and frameworks are primarily used for the purpose of automated application testing. Tools currently available for the purpose of automatic application execution can be broadly divided into two categories. The first type of tools like Sikuli [10], Selendroid [11] require the developer to generate a test case specific to the application. Test case developers for these tools need to have information like ID, text, alignment etc about UI elements of the application. The second category of tools are of 'Record and play' type. Here the user needs to record a sequence of events first and then replay them. Ranorex [12] and Reran [13] are tools which fall into the second category. In both the categories of tools mentioned above, either manual intervention is required or it is essential to run the application at least once for test case generation. Another test automation framework, GUIRipper [14] tests Android applications via their GUI by automatically exploring the application with the aim of exercising the application GUI in a structured manner. PUMA [15] is a programmable framework containing a generic UI automation and analysis. It uses Monkey [16] for triggering events on the GUI. The monkey tool triggers a set of pseudo random events on the GUI. Hence the execution path is random and not structured. Robotium [17] is an open-source test framework for writing automatic grey box test cases for Android applications. Robotium can be used for developing test cases for function, system and acceptance test scenarios, spanning multiple Android activities. TaintDroid [18] provides a system-wide dynamic taint tracking across multiple sources of sensitive data. DroidScope [19] is an Android analysis platform based on virtual machine introspection. DroidScope reconstructs both the OS-Level and Java-level semantics simultaneously. Also to facilitate custom analysis across three levels of an Android device, that is hardware, OS and Dalvik Virtual Machine, DroidScope provides possibility to develop plug ins which monitor activities across all three levels. Neither TaintDroid nor DroidScope provide any means of automatic application execution. CopperDroid [20], a dynamic analysis tool, provides system call-centric analysis of the application. For application execution, CopperDroid installs and UN-installs the application thrice and analysis is done on

traces collected only during installation and uninstallation. Due to limited execution, most of the application behaviours can't be observed. In Construction of AndroInspector, we used Robotium framework to develop application specific test cases and DroidScope for monitoring application by collecting traces during application execution. Aubrey-Derrick Schmidt et al. [21] extracted function calls of an installed application using readelf command. These function calls were later compared with function calls of the malware executables present on a Remote Detection Server. In contrast to this, our approach does not analyse applications on an Android device because of limited resources like power, memory and data usage. DroidRanger [23] detects malicious applications of known malware families in popular Android marketplaces using permission-based behavioural foot printing. To detect malware from unknown families, DroidRanger uses heuristic-based filtering scheme. The drawback of DroidRanger is the requirement of manual operations while analysing and collecting behaviour of applications.

3. APPROACH

AndroInspector performs both static and dynamic analysis on a given Android application and uses information gained from both to provide a comprehensive view of application behaviour.

Illustration 1: AndroInspector Architecture



The static analysis component gives out a verdict as to weather the application is malicious or benign. The dynamic analysis component lists out the suspicious actions performed by the application during execution. Figure 1 depicts AndroInspector architecture.

3.1. Dynamic Analysis

Dynamic Analysis of an Android application refers to analysing the application during its execution. AndroInspector performs dynamic analysis by first executing the application on an Android emulator and collecting various levels of traces simultaneously. The traces generated are then analysed to identify malicious actions. This process is divided into 3 phases namely preparation phase, execution phase and analysis phase. The test case for application execution is generated during the preparation phase. Execution phase comprises of test case execution and collecting run time traces. During the analysis phase, traces collected in execution phase are analysed to detect suspicious behaviour.

3.1.1. Preparation Phase

Traversal through an application during application execution comprises of traversing through the application's activities as well as triggering events on all the UI widgets present in each activity. The event triggering may lead to another activity or may trigger some functionality. When provided only with an APK file we do not have required information to generate test case for application execution. To extract the necessary information, we disassemble the APK. The application is disassembled using apktool [24]. The information thus acquired is used to generate an Robotium based test case specific to the application. Information extracted for test case generation is explained below:

Package name: Application's package name is required while installing the application. An application's package name can be extracted from its corresponding 'AndroidManifest.xml' file.

Launch activity: The launch activity/Main activity of an application is where the application execution starts. The launch activity name is available in the 'AndroidManifest.xml' file.

List of activities: All activities present in an application are listed in it's 'AndroidManifest.xml' file.

List of intent filters: List of all the intents and intent-filters are used to invoke the broadcast receivers and services which may be waiting for some specific action to occur on which intent would be triggered. Intent filters are extracted from the Android Manifest file.

The test case generated is structured in a way that all the activities comprising the application are traversed in a depth first search fashion. DFS for application execution means first main activity is traversed and all other activities are traversed sequentially in the order of their reachability from main activity.

3.1.2. Execution Phase

By the end of preparation phase we have a robotium based test case specific to the application to be executed. The Android emulator used during dynamic analysis is DroidScope. (Reasons for using droidscope are stated in the next section). If the application and test case have different signatures, then test case does not have access to the application and its elements. To overcome this, we re-sign the application under analysis and test case with “Android Debug Mode”. The test case is then compiled and built using Apache Ant [25] tool. The Android application is then executed on the emulator using test case on the device. This test case is limited to testing the UI elements and testing the Activities in an application.

Initially the test case starts the application execution by launching the Launch/Main activity. The test case then triggers events on all the elements present in the activity. Triggering events on UI elements is performed by using the Robotium based API's provided for different types of UI elements. UI elements like buttons, image buttons, list views etc. are clicked, where as edit texts, date time pickers are set with some per-determined values. Both types of actions (clicking and setting values) are carried out by using API's provided by Robotium which use the IDs of elements to identify elements and perform a specified action on them. If on triggering an event on any UI element causes the launch of another activity, then the activity launched is identified and actions are performed upon elements in the newly launched activity. This is repeated till the control reaches an activity(let's say 'activity Last') from where another activity cannot be instantiated, when actions on all elements in that activity are performed, control moves back to the previous activity(that is the activity from which 'activity Last' had originated) and checks

whether actions upon all the elements in the activity have been performed. If yes, we go back to previous activity or else perform action upon the remaining elements.

Once all the activities are parsed and all the elements in those activities are executed, we exit the test case. Figure 2 shows the flow of execution in test case.

After execution of all the UI elements of the application, the broadcast receivers in the application are invoked one at a time by triggering intents specific to the broadcast listeners. Triggering intents is done by using 'Activity Manager' (am) in 'adb shell'. By this point, activities and broadcast receivers have been executed.

Even though we did not explicitly start the Services, the tests executed above would have started the following types of services :

Services which are started when application is installed
Services which are started when application is launched
Services which start when any activity is launched
Services which start when some action is performed on an UI element
Services which are start on receiving specific intents.

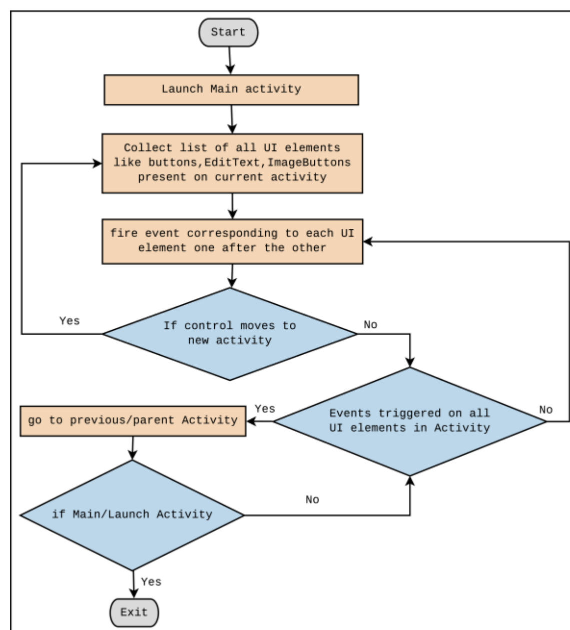


Illustration 2: Flowchart for automatic application execution

3.1.3. Analysis Phase

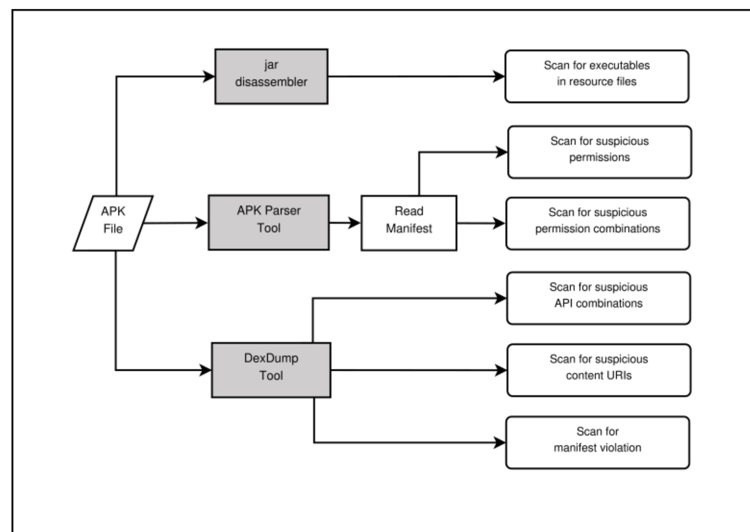
The traces collected during execution phase are used for analysis. During the execution phase, the application is executed on DroidScope emulator. DroidScope emulator is an Android analysis platform for virtualization-based malware analysis. DroidScope provides the possibility to develop plug ins to access both the OS-level and Java-level semantics simultaneously and seamlessly. DroidScope monitors the three levels of an Android device: hardware, OS and Dalvik Virtual Machine. Using DroidScope, We developed plug ins to monitor and record the a) dalvik instruction traces b) system calls and c) API-level activity. The network activity performed by the application during application execution is captured using tcpdump. All the information gathered from dalvik instruction traces, system calls traces, API calls and network activity traces are then parsed to identify a set of per-defined patterns which indicate the occurrence of malicious

activity. The malicious actions observed are then reported to the user. For deciding upon the patterns which would act as an indicator of malicious action, We executed 1260 malware samples on DroidScope and manually studied the traces extracted. As the behaviour corresponding to the malware samples from training set were known, the system call traces and API traces were observed when the malicious action was performed to deduce patterns which would help in identifying the occurrence of malicious activity. For example, to identify if the application is trying to send an SMS without user's consent, we look for API corresponding to sending SMS and also observe if the Messaging application was opened or not. If the Messaging application was not opened and an SMS was sent from the application under analysis, it is considered as a malicious action. Another example is 'dev/urandom_Access'. We parse through the system call traces to identify read or write system calls upon path 'dev/urandom'. Each pattern thus identified substantiates the occurrence of a specific malicious action. Any malicious action found is reported to the user. Malicious actions which were considered for finding patterns are stated in the Table 1

3.2. Static Analysis

Android applications are installed by using an Android application package (APK) file. APK file is an archive filewhich contains Java classes, resources and Manifest file. Static analysis constitutes of unpacking the android application and analysing the contents of application. Static analysis component of AndroInspector unpacks the application, extracts necessary information and uses the information extracted to classify the application as either malicious or benign using machine learning techniques. The information extracted for analysis is in form of various features of an Android application. Figure 3 shows how various features are extracted from an Android application.

Illustration 3: Feature extraction in AndroInspector



Following sub-sections describe feature selection for feature set, weight assignment to the features and selection of feature vector.

3.2.1. Features

3.2.1.1. Suspicious Permissions and Permission Combinations

A permission is a restriction limiting the access of an application to the device to protect critical data and code that could be misused to distort or damage the user experience. We considered the

patterns of suspicious permissions in malware samples as discovered by Y.Zhou et.al. [26]. For extracting permissions used by an application we use APKParser tool [22]. The permissions extracted were analysed and cross verified for high occurrence across malware samples available in our training dataset. Out of all the permissions specified as suspicious by Y.Zhou et.al, we discarded those permissions which were present in large numbers in benign samples as these would not significantly contribute during classification process. The presence or absence of the remaining suspicious permissions was then considered as a feature. Our findings are shown in Figure 4.

I.Rassameeroj [27] states that certain permission combinations enable an application to perform dangerous actions posing threat to user's data and privacy. We considered these combinations as features for our feature set. Table 2 depicts the permissions and permission combinations considered as features.

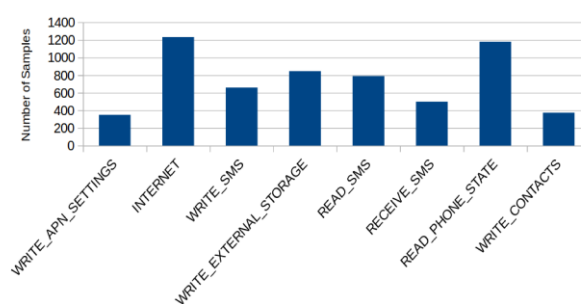


Illustration 4: Frequency of suspicious permissions among malware samples

3.2.1.2. Suspicious API Combinations

APIs used by an application determines the actual functionality and capability of the application. Static analysis of APIs used in an application hence becomes important to understand what the application actually intends to do. In the similar direction of selecting permissions as features, our approach contributes by evaluating APIs extensively used by malware applications. APIs were broadly classified according to their usage by the application. From the list of APIs which are found in large number of malware samples, combinations were derived which could pose a threat to the user. Two main types of threats considered are financial losses and leakage of user's personal information. For example APIs for accessing user's personal information (network details, device ID, line number, etc.) in combination with APIs for sending SMS enables an application to transmit user's personal information to a predefined source. This leads to both breach of privacy as well as monetary loss. The monetary loss here is due to cost incurred when the SMS is sent.\par APIs for evaluation are extracted by disassembling classes.dex file using dexdump tool present in Android SDK [28]. Figure 5 depicts the a snapshot of classes.dex when disassembled using dexdump tool. Table 3 lists the API combinations considered as a feature for our feature set.

Illustration 5: Disassembled dex file

```
B582 018370: 0c09                 |0010: move-result-object v9
B583 018372: 7100 d800 0000       |0011: invoke-static {}, Landroid/telephony/SmsManager;.getDefault:()Landroid/telephony/
     SmsManager; // method@00d8
B584 018378: 0c0c                 |0014: move-result-object v12
B585 01837a: 7210 5b00 0900       |0015: invoke-interface {v9}, Landroid/database/Cursor;.getCount:()I // method@005b
B586 018380: 0a01                 |0018: move-result v1
B587 018382: 3d01 0800            |0019: if-lez v1, 0021 // +0008
B588 018386: 7210 5d00 0900       |001b: invoke-interface {v9}, Landroid/database/Cursor;.moveToNext:()Z // method@005d
B589 01838c: 0a01                 |001e: move-result v1
B590 01838e: 3901 0300            |001f: if-nez v1, 0022 // +0003
B591 018392: 0e00                 |0021: return-void
B592 018394: 1a01 b007            |0022: const-string v1, "_id" // string@07b0
B593 018398: 7220 5a00 1900       |0024: invoke-interface {v9, v1}, Landroid/database/Cursor;.getColumnIndex:(Ljava/lang/
     String;)I // method@005a
B594 01839e: 0a01                 |0027: move-result v1
B595 0183a0: 7220 5c00 1900       |0028: invoke-interface {v9, v1}, Landroid/database/Cursor;.getString:(I)Ljava/lang/
     String; // method@005c
B596 0183a6: 0c0a                 |002b: move-result-object v10
B597 0183a8: 1a01 c80b            |002c: const-string v1, "has_phone_number" // string@0bc8
B598 0183ac: 7220 5a00 1900       |002e: invoke-interface {v9, v1}, Landroid/database/Cursor;.getColumnIndex:(Ljava/lang/
     String;)I // method@005a
```

3.2.1.3. Manifest Violation

All the permissions required by an application should be declared in the AndroidManifest.xml. These permissions determine what are all the capabilities the application has. During application installation, all the permissions declared by the application are not cross verified by the package manager. Thus, at the run time if the application needs to perform a certain action and it does not have corresponding permission, run time exceptions occur. Malware developers take advantage of this flaw to perform collusion attacks [29]. The collusion attack requires at least 2 applications to work in collaboration. In this type of attack, an over privileged application provides an under privileged application with necessary permissions at runtime. Soundcomber [30] is one such application which aims at collecting user's information by capturing audio from device's microphone and then sends it over the network with help of another application having necessary permissions. Figure 6 depicts a scenario where two applications combine their permissions to read contacts and send them over the network.

One way to detect the possibility of collusion attack is to look for application which has declared more permissions than what it requires (over privileged applications), but the drawback with this approach is the high false positive rate. The reason for high false positive rate is that many developers declare majority of the permissions available irrespective of their usage by the application.

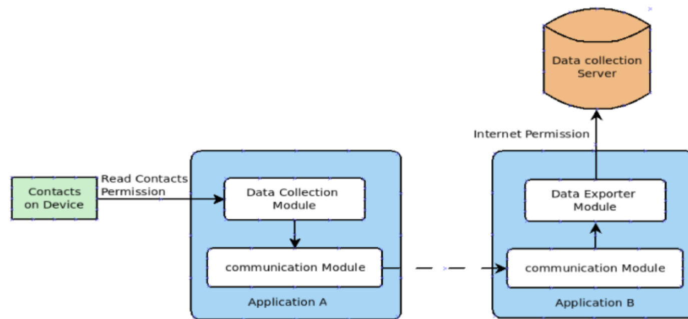


Illustration 6: A collusion attack scenario

We devised a different approach for detecting possible collusion attack. Rather than looking for over privileged applications we detect under privileged applications, that is the application declaring less permissions than what it actually required. The under privileged application then gets required privileges at runtime with the help of another application. To detect under privileged applications applications, we look for the permissions that will be used by the application at run time but are not present in application's manifest file. To derive permissions required by application at run time, permission required for executing each API present in application's dex

file is extracted. If any permission required for execution of an API is not found in the application's manifest file, it is considered as a manifest violation.

We derive the permissions required by an API with the help of Android's developer guide and Pscout [31].

Each occurrence of manifest violation is assigned a weight of 7. A summation of these permission's weights was considered as the weight of the feature (Manifest violation).

3.2.1.4. Suspicious Content URI

A content URI (used for data access) can be called suspicious if by using that URI an application can leak user's personal data or can access another application's data. For example, an application can get access to contacts by using URI: content://com.Android.contacts. Such suspicious URIs were identified and their presence was checked among various malware and benign samples available in the training set. Suspicious content URIs which were detected in most of the malware samples and few benign samples were considered as a feature for feature set. Figure 7 shows the content URIs extensively used by malware applications.

To collect the content URIs used by the application, we parse the dalvik byte code of disassembled classes.dex. The presence of content URIs that provide access to MMS, Browser and telephony data were seen among majority of malware applications.

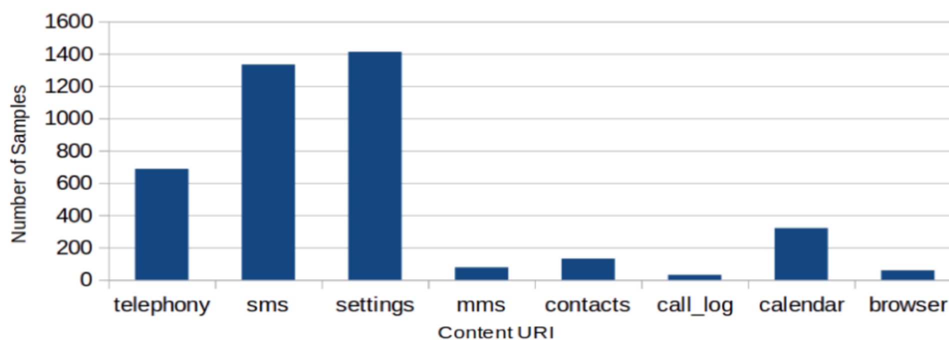


Illustration 7: Frequency of suspicious content URIs among malware samples

Each Suspicious Content URI was assigned a weight of 6. Summation of the weights for frequency of such suspicious content URIs is considered as the weight of the feature.

3.2.1.5. Detection of Executable code

Embedding malicious code into documents has been successful technique for distributing malware. Desktop malware like Pidief, ZBOT, SillyD have been distributed as malicious PDF, JPEG, mp3 files. Based on Shafiq [32] and Stolfo's [33] findings which stated that detection of embedded malware requires parsing the byte code of the documents, We employed a mechanism to find embedded executables by parsing the byte code of all the files present in the resources directory of an APK. Many malware samples show the presence of executables and shell scripts embedded within image and music files. Presence of image files embedded with executable code can be found in samples from malware families like DroidKungFu1 and RougePush. Malware samples from DroidKungFu3 and GingerMaster families show presence of music files embedded with executable code.\par As this behavior was detected only in malware samples, presence of embedded executables was assigned a maximum weight of 10. Summation of the weights for frequency of such files is considered as weight of the feature.

3.2.2. Assigning Weight to Features

The weight assigned to a feature represents the impact that presence or absence feature makes on an application's classification. Weights are assigned to each feature on a scale of 1 to 10 using heuristics based approach such that higher the weight of a feature, more the feature contributes during classification. The highest weight of 10 was assigned to presence of executables embedded in image or music files. Presence of embedded executables is the strongest indicator in our feature set of an application being malicious as only malware samples are found to have resource files injected with executable code. All other features were assigned weights relative to the weight of 'presence of embedded executables' feature. Manifest violations are assigned a weight of 7. This is because unlike a malicious application, a benign application declares all the permissions being used. When compared to 'suspicious Permission combinations' or 'suspicious API combinations', 'manifest violation' has more impact during classification but it is not as influential as 'presence of embedded executables'. Thus it is assigned a weight lower than 'presence of embedded executables' and higher than 'suspicious Permission combinations' and 'suspicious API combinations'. Presence of suspicious content URI in an application is assigned a weight of 6. The presence of these content URI was seen in both malicious and benign samples, but number of malicious samples containing these URIs was much greater than number of benign samples. Weights for suspicious content URIs, manifest violations, presence of executable code are frequency based. Thus the total weight for these features in the feature set is multiple of the frequency of the feature occurrence and the weight assigned to the feature. \par Permission combinations and API combinations are assigned a moderate weight of 5 as the presence of these leads to suspicious behaviours, but their presence cannot conclude an application of being a malware or benign. We assigned suspicious permissions the lowest weight of 3 as these permissions can be found in large number in both benign and malware samples. Table 4 depicts the assignment of weights to the features selected.

3.2.3. Feature Vector Selection

After deciding upon the application's attributes to be considered as features, we considered and evaluated three categories of feature vectors with a set of machine learning algorithms. All the three categories of feature vectors constituted of similar features, but represented in different way. The first and second categories of feature vectors were weighted feature vector where as the third category was a non weighted feature vector. The first category of feature vector contained weights for each feature along with the Euclidean distance as an additional feature. The second category of feature vector was derived by excluding Euclidean distance from the first feature vector. For the third category of feature vector, rather than considering the frequency and weight of a feature, we check only presence of a feature. Representation in feature vector is done as either 1 or 0 to depict the presence or absence of a specific feature in the sample.

3.2.3.1. Evaluation of model for Feature Vector Selection

K-fold cross validation was carried out in order to evaluate the efficiency of the classification model. The default implementation of cross validation provided by WEKA was used for this purpose. The efficiency of the classifier models generated using all three categories of feature vectors were compared based on cross validation. One round of cross-validation of a two class classifier model involves seggregating a sample of the training data set into two complementary subsets, subset for performing the analysis (the training set) and subset for validating the analysis (the validation set). Inconsistency is reduced by multiple rounds of cross-validation using different seggregations. Finally the average of all validation results is presented as true positive rate and false positive rate. We used WEKA [34] implementation for both model generation and cross validation. The true positive rate and false positive rate are deduced as follows :

$$\text{TPR}=\frac{TP}{TP+FN}$$

$$\text{FPR}=\frac{FP}{FP+TN}$$

Figure 8 (a) and Figure 8 (b) show variations in true positive rates and variations in false positive rates respectively for models generated using three categories of feature vectors.

High true positive and low false positive rates are observed for the second category of feature vector, that is a feature vector with weights and excluding Euclidean distance. Thus the second

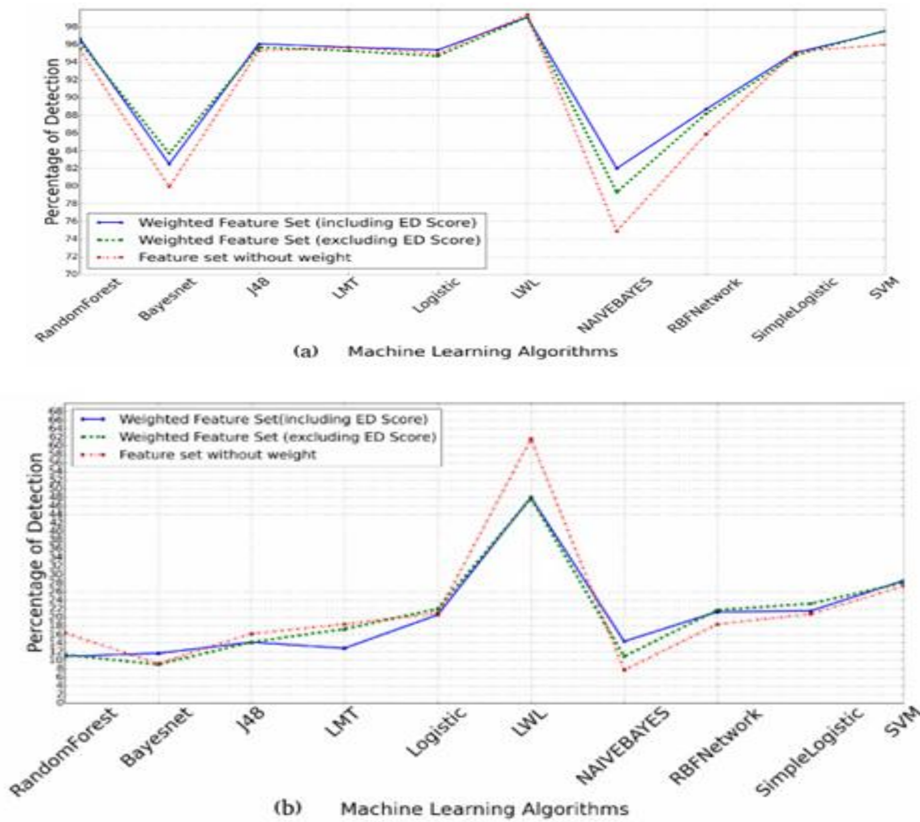


Illustration 8: Variation inTPR (a) and FPR (b) for various models

category of feature vector was considered for providing features to the machine learning algorithms. The reason for omitting Euclidean distance from the feature set was its last rank among the features on applying Chi-Square attribute ranking mechanism. This illustrated that excluding it as a feature would not affect the detection rates. Figure 9 shows variation in Euclidean distance across all the samples present in our dataset.

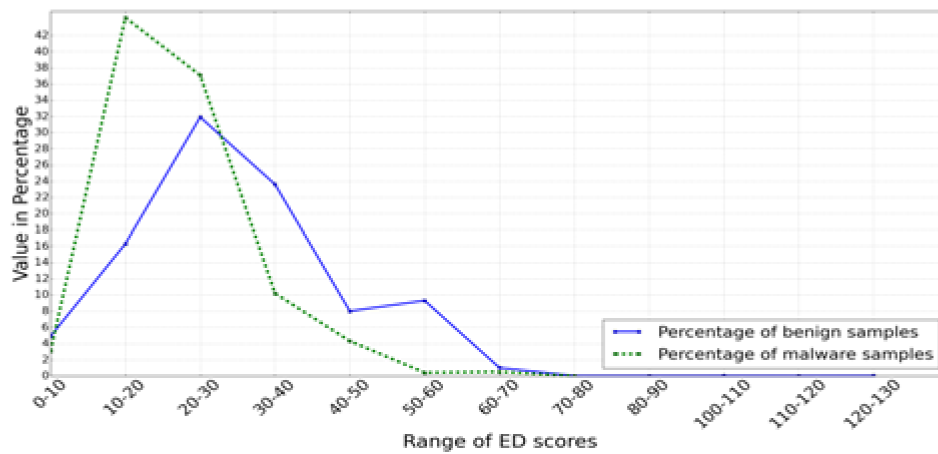


Illustration 9: Variation in ED scores among benign and malware samples

Figure 10 shows the receiver operating characteristic (ROC) graph for the classification model built using second category of feature set. This graph illustrates the performance of a binary classifier system built using various machine learning algorithms and the weighted feature set. Random Forest algorithm depicts the maximum ROC space in the ROC curve which proves that for the given training set, classifier model built using Random Forest is more efficient than models generated using other machine learning algorithms. We used model built using Random Forest algorithm as the classifier in AndroInspector implementation.

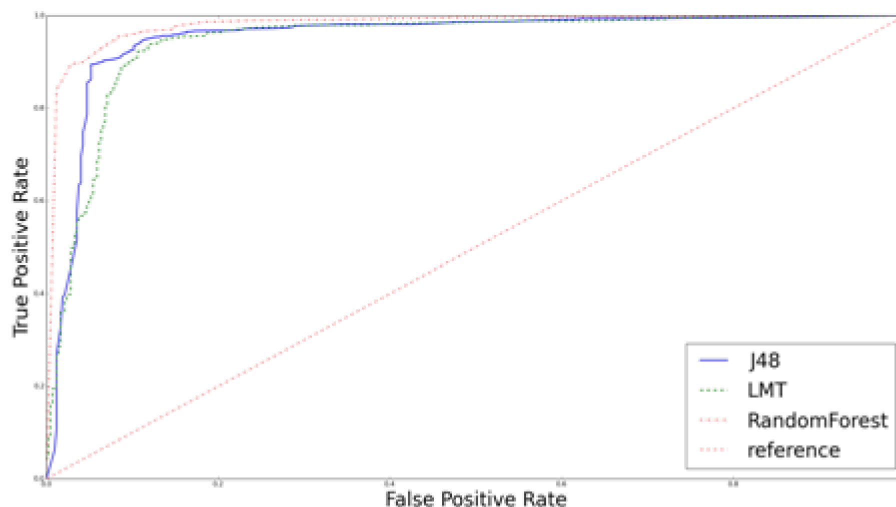


Illustration 10: ROC Curve for classifier models based on various algorithms

3.2.3. Classification Using AndroInspector

Classification of an Android application by AndroInspector as either malicious or benign is based solely on information obtained during the static analysis of the application. Static analysis is carried out in two phases. First phase is the knowledge building phase. In this phase, AndroInspector extracts specific features and builds feature set of all the samples from the training set. These feature sets are then provided to the machine learning algorithm using WEKA

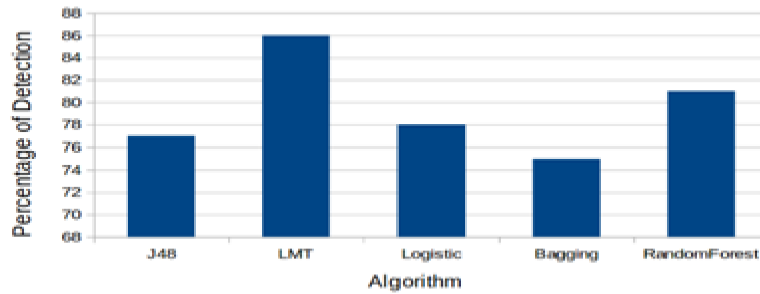


Illustration 11: Detection rate of AndroInspector for malware samples

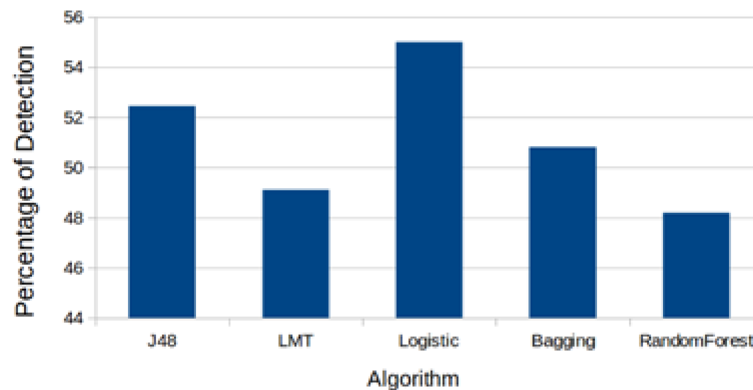


Illustration 12: Detection rate of AndroInspector for benign samples

implementation of machine learning algorithms. A two class classifier model is thus generated. Classifier model generated during this phase can be used for classification of samples without updating the model every time a new sample is provided for analysis.

Second phase is the classification phase. In this phase, features are extracted from test application which needs to be classified and a corresponding feature set is built. Now this feature set is provided to the classification model generated during phase 1. The classification model then classifies the sample as either malicious or benign

4. RESULTS

After application analysis, AndroInspector generates an output json file. This output report generated contains details regarding the presence or absence of all the features under consideration and a verdict on weather the application is either malicious or benign. The report also specifies all the suspicious content URIs and embedded executables present in the application.

The efficiency of AndroInspector's classification model was tested by analysing 500 malware samples obtained from Virustotal malware intelligence service [3] and 800 benign samples from ApkDrawer [35]. Collectively these samples constituted of our test-set. It was verified beforehand that the test-set does not contain any samples in common with the training-set by comparing the hash code of each sample in test set against hash codes of samples from training set. Figure 11 and Figure 12 depict the detection rates of malware samples and benign samples respectively by using AndroInspector.

Results from Dynamic analysis of malware samples obtained from GENOME project [2] indicates that 90 percent of the malware samples analysed performed cryptographic operations and 85 percent of them accessed files related to the device. Figure 13 depicts malicious actions observed on executing malware samples.

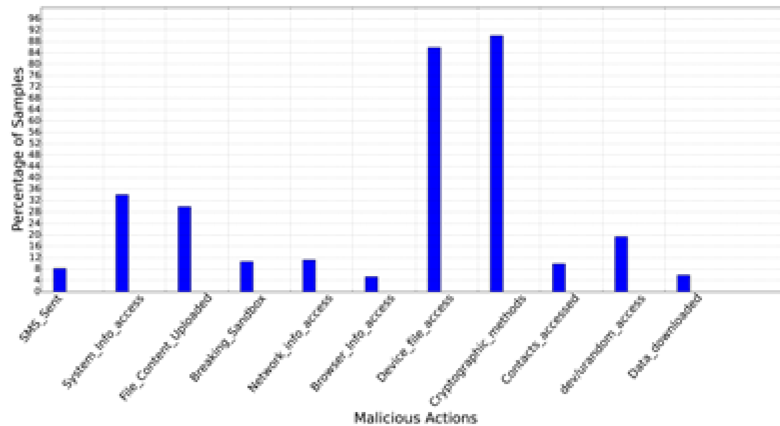


Illustration 13: Malicious actions detected during execution of malware samples

On analysis of network activity based traces it was observed that most malware samples communicate with IP addresses based in Beijing and Guangzhou cities of China. Figure 14 depicts network connections made by malware samples with IP addresses from different cities.

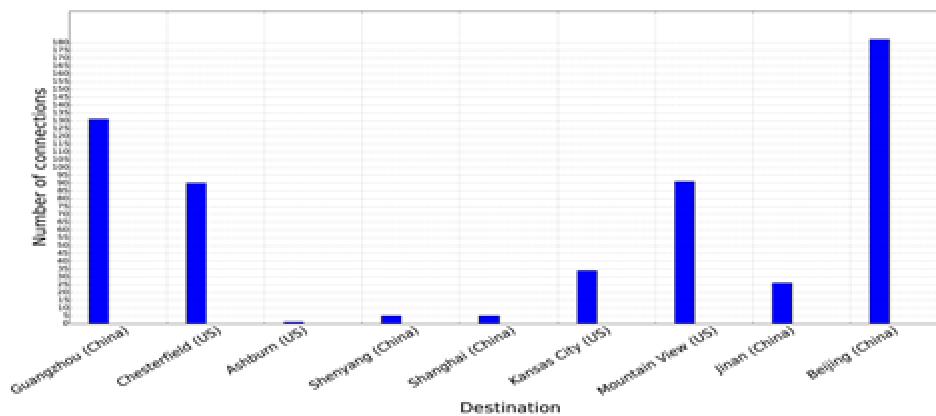


Illustration 14: Network connections made by malware samples

The detection rate of AndroInspector was compared with the detection rates of four other anti virus solutions for the same set of malware samples. Figure 15 shows the detection rate of AndroInspector in comparison with Kaspersky (version 12.0.0.1225) [36], McAfee (version 6.0.5.614) [37], Avast (version 8.0.1489.320) [38] and TrendMicro (version 9.740.0.1012) [39].

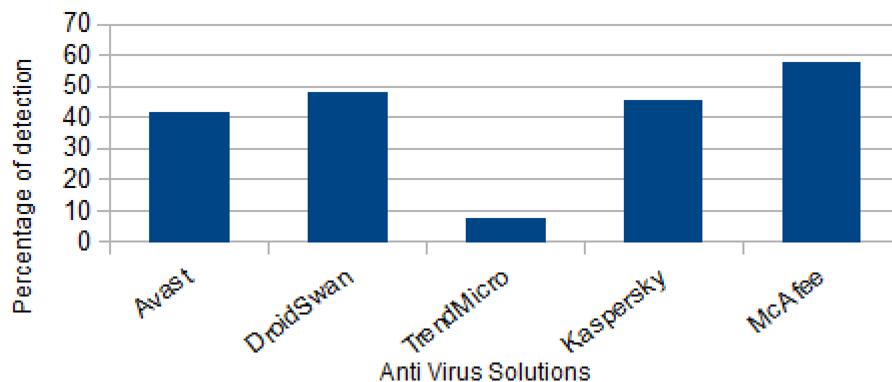


Illustration 15: Detection rates of AndroInspector in comparison with other AV solutions

ecall rate of AndroInspector with Random Forest based classifier for malwares from various malware families is shown in Figure 16

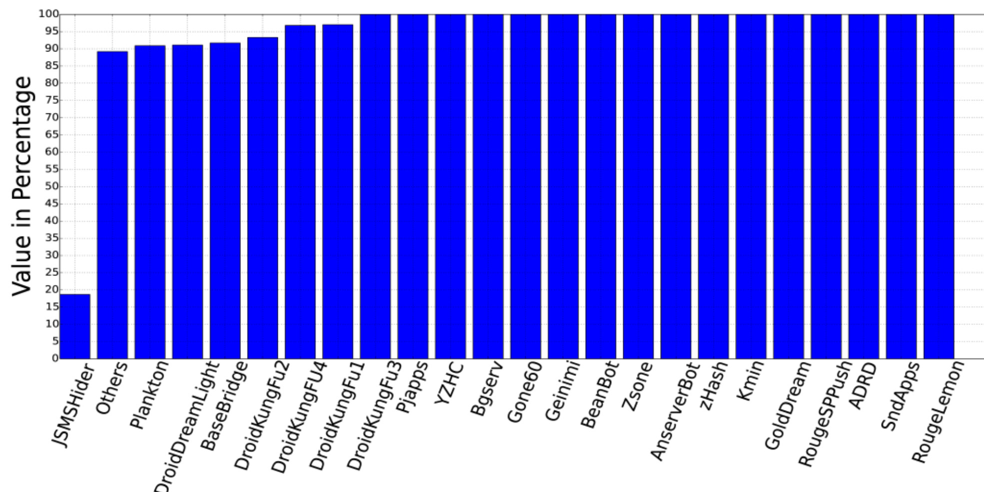


Illustration 16: Recall rate of AndroInspector for various malware families

5. Conclusion

We present AndroInspector, an approach for detecting malicious Android applications based on static analysis and dynamic analysis of their respective APK files. Static analysis is responsible for classifying the application as either malware or benign whereas dynamic analysis identifies the malicious actions performed by the application during execution. The process of classification comprises of extracting 24 features, assigning weights to the features and finally using the collection of feature weights as a feature set. The feature set along with Random Forest classifier model is then used to classify the given sample as either malware or benign. We observed that classifier model built using Random Forest shows higher TPR and lower FPR when compared to other machine learning algorithms. Observations from dynamic analysis revealed that a large number of malware samples (Training set and test set) accessed device related information. Analysis of application's network activity revealed that majority of malware samples connected to servers located in China.

Future scope of work involves developing a classifier model which considers information from both static analysis and dynamic analysis while classifying an Android application. We are also working towards modifying the dynamic analysis component such that it's functioning would be independent of application's Android version and would works on any generic Android emulator.

REFERENCES

[1] RiskIQ, Feb 19 2014, Research Also Shows Steady and Significant Drop in Number of Malicious Apps Being Removed in Past Three Years. Available: http://www.riskiq.com/company/press-releases/riskiqreports-malicious-mobile-apps-google-play-have-spiked-nearly-400

[2] Genome Project. Android malware samples. http://www.malgenomeproject.org.

[3] S. Hispasec Sistemas. Virustotal malware intelligence service, 2011.

[4] A. Desnos. Androguard. Available at https://code.google.com/p/androguard/.

[5] Wu, Dong-Jie, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and KuoPing Wu. ”Droidmat: Android malware detection through manifest and API calls tracing.”In Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on, pp. 62-69. IEEE, 2012.

[6] Felt, Adrienne Porter, et al. ”Android permissions demystified.”Proceedings of the 18th ACM conference on Computer and communications security. ACM, 2011.

[7] Enck William, Machigar Ongtang, and Patrick McDaniel. ”On lightweight mobile phone application certification.” Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009.

[8] Zheng, Min, Mingshen Sun, and John Lui. ”Droid Analytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware.” Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on IEEE, 2013.

[9] Shabtai, Asaf, Yuval Fledel, and Yuval Elovici. "Automated static code analysis for classifying Android applications using machine learning." Computational Intelligence and Security (CIS), 2010 International Conference on. IEEE, 2010.

[10] Yeh, Tom, Tsung-Hsiang Chang, and Robert C. Miller. "Sikuli: using GUI screenshots for search and automation." Proceedings of the 22nd annual ACM symposium on User interface software and technology. ACM, 2009.

[11] Selendroid, Ebay software foundation,'Test automation for native or hybrid Android apps and the mobile web with Selendroid.'. http://selendroid.io/

[12] Ranonex. Android Test Automation - Automateyour App Testing. http://www.ranorex.com/mobile-automation-testing/android-test-automation.html.

[13] Gomez, Lorenzo, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. "Reran: Timing-and touch-sensitive record and replay for android." In Software Engineering (ICSE), 2013 35th International Conference on, pp. 72-81. IEEE, 2013.

[14] Amalfitano, Domenico, et al. "Using GUI ripping for automated testing of Android applications." Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM, 2012.

[15] Hao, Shuai, et al. "PUMA: Programmable UI-Automation for Large Scale Dynamic Analysis of Mobile Apps." Proceedings of the 12th annual international conference on Mobile systems, applications, and services. ACM, 2014

[16] Google. UI/Application Exerciser Monkey, http://developer.android.com/guide/developing/tools/monkey.html

[17] Robotium. User scenario testing for Android. http://code.google.com/p/robotium/.

[18] Enck, William, et al. "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones." Communications of the ACM 57.3 (2014): 99-106.

[19] Yan, Lok-Kwong, and Heng Yin. "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis."USENIX Security Symposium. 2012.

[20] Tam, Kimberly, et al. "CopperDroid: Automatic Reconstruction of Android Malware Behaviors." (2015).

[21] Schmidt, A-D., Rainer Bye, H-G. Schmidt, Jan Clausen, Osman Kiraz, Kamer A. Yuksel, Seyit Ahmet Camtepe, and Sahin Albayrak. ”Static analysis of executables for collaborative malware detection on android.” In Communications, 2009. ICC’09. IEEE International Conference on, pp. 1-5. IEEE, 2009.

[22] J. Erdfelt. Apkparser tool. https://code.google.com/p/xml-apk-parser.
[23] Zhou, Yajin, Zhi Wang, Wu Zhou, and Xuxian Jiang. ”Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets.” In NDSS. 2012.
[24] Winsniewski, R.: Android, “Apktool: a tool for reverse engineering Android apk files,” 2012,[Online] Available: http://code.google.com/p/android-apktool/
[25] Ant, Apache. "The Apache Ant Project." (2010).
[26] Zhou, Yajin, and Xuxian Jiang. "Dissecting android malware: Characterization and evolution." Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012.
[27] Rassameeroj, Ittipon, and Yuzuru Tanahashi. "Various approaches in analyzing Android applications with its permission-based security models." Electro/Information Technology (EIT), 2011 IEEE International Conference on. IEEE, 2011.
[28] Google Inc. Official Page for android developers. http://developer.android.com.
[29] Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. "Towards Taming Privilege-Escalation Attacks on Android." In NDSS. 2012.
[30] Schlegel, Roman and Zhang, Kehuan and Zhou, Xiao-yong and Intwala, Mehool and Kapadia, Apu and Wang, XiaoFeng. 'Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones.'NDSS, 2011
[31] Au, Kathy Wain Yee, Yi Fan Zhou, Zhen Huang, and David Lie. "Pscout: analyzing the android permission specification." In Proceedings of the 2012 ACM conference on Computer and communications security, pp. 217-228. ACM, 2012.
[32] Shafiq, M. Zubair, Syed Ali Khayam, and Muddassar Farooq. "Embedded malware detection using markov n-grams." In Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 88-107. Springer Berlin Heidelberg, 2008.
[33] Stolfo, Salvatore J., Ke Wang, and Wei-Jen Li. ”Towards stealthy malware detection.” Malware Detection. Springer US, 2007. 231-249.
[34] Hall, Mark, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. ”The WEKA data mining software: an update.” ACM SIGKDD explorations newsletter 11, no. 1 (2009): 10-18.
[35] Z. Jay. Apkdrawer.com. http://www.apkdrawer.com.
[36] Kaspersky mobile security. Available at http://www.kaspersky.co.in/downloads/android-security.
[37] Mcafee mobile security. Available at https://www.mcafeemobilesecurity.com/.
[38] Avast mobile security. Available at http://www.avast.com/en-in/free-mobile-security.
[39] Trendmicro mobile security. Available at http://www.trendmicro.com/us/enterprise/product-security/mobile-security/.

APPENDIX

Table 1. Malicious Actions Considered

Feature	Threat
SMS_Sent	Application sending SMS without user's interaction
Data_Download	Application is trying to download data over the network
dev/urandom_Access	Application performs read or write operations on /dev/urandom limits the expansion of entropy pool of /dev/random thus limiting the randomness and hence making it easier to crack cryptographic algorithms
BreakingSandbox	Application accessing other applications on the device or data related to other applications
NetworkInfo_Access	Application accessing device's network related information

BrowsingInfo_Accessed	Application accessing cookies and browsing history from the device
SystemInfo_Access	Application can know the current system state
CryptographicMethods	Application using cryptographic algorithms for data encryption
Contacts_Accessed	Application has accessed contacts on device without user's consent
File_Content_Uploaded	Application sending a file over the network
DeviceFiles_Access	Application accessing sensitive files related to the device

Table 2. Suspicious permissions and permission combinations

Suspicious permissions and permission combinations	Weight assigned
READ SMS	3
WRITE SMS	3
RECEIVE SMS	3
WRITE CONTACTS	3
WRITE APN SETTINGS	3
SEND SMS	3
ONLY INTERNET	3
ONLY WRITE EXTERNAL STORAGE	3
WRITE SMS and RECEIVE SMS	5
SEND SMS and WRITE SMS	5
INTERNET and WRITE EXTERNAL STORAGE	5
INTERNET,RECORD AUDIO, READ PHONE STATEand MODIFY PHONE STATE	5
ACCESS FINE LOCATION or ACCESS COARSE LOCATION, RECEIVE BOOT COMPLETED and INTERNET	5
INTERNET,RECORD AUDIO and PROCESS OUTGOING CALLS	5

Table 3. Suspicious API combinations

Suspicious API combinations	Weight assigned
"landroid/telephony/telephonymanager;.getdeviceid" "landroid/location/locationmanager;.getlastknownlocation" "landroid/location/location;.getlatitude" "landroid/location/location;.getlongitude" "landroid/telephony/smsmanager;.sendtextmessage" "landroid/net/uri;.parse","landroid/location/locationmanager;.getbestprovider"	5
"ljava/net/urlencoder;.encode" "ljava/net/uri;.getquery" "ljava/net/httpurlconnection;.connect" "ljava/net/httpurlconnection;.geturl" "ljava/net/httpurlconnection;.getheaderfield" "landroid/location/locationmanager;.getbestprovider" "landroid/location/location;.getlatitude" "landroid/location/location;.getlongitude" "landroid/telephony/gsm/smsmanager;.sendtextmessage"	5
"landroid/net/uri;.parse" "landroid/content/contentresolver;.query" "landroid/database/cursor;.movetonext" "landroid/database/cursor;.getcolumnindex" "landroid/database/cursor;.getstring" "landroid/database/cursor;.close" "landroid/database/cursor;.movetolast" "landroid/database/cursor;.movetoprevious"	5
"landroid/net/uri;.parse" "ljava/net/urlencoder;.encode" "ljava/net/url;.openstream" "landroid/telephony/telephonymanager;.getdeviceid" "landroid/telephony/telephonymanager;.getline1number" "landroid/telephony/telephonymanager;.getnetworkcountryiso" "landroid/telephony/telephonymanager;.getnetworkoperatorname" "ljava/io/bufferedreader;.readline" "landroid/content/pm/packagemanager;.hassystemfeature"	5
"ljava/net/inetaddress;.getlocalhost"	5

"ljava/net/inetaddress;.gethostname"

"ljava/net/url;.openstream"

"ljava/net/inetaddress;.getbyname"

"ljava/net/inetaddress;.equals"

"ljava/net/inetaddress;.hashcode"

"landroid/net/uri;.parse"

"landroid/telephony/smsmanager;.getdefault"

"landroid/telephony/smsmanager;.dividemessage"

"landroid/telephony/smsmanager;.sendtextmessage"

"landroid/telephony/telephonymanager;.getdeviceid"

"landroid/telephony/telephonymanager;.listen"

"ljava/net/urlencoder;.encode" 5

"ljava/net/uri;.<init>"

"landroid/location/location;.hasaccuracy"

"landroid/location/location;.distanceto"

"landroid/location/location;.gettime"

"landroid/location/location;.getaccuracy"

"landroid/location/location;.getlatitude"

"landroid/location/location;.getlongitude"

"landroid/location/location;.getprovider"

"landroid/location/locationmanager;.requestlocationupdates"

"landroid/location/location;.<init>"

"landroid/location/location;.setaccuracy"

"ljava/net/urlencoder;.encode" 5

"ljava/net/url;.<init>"

"ljava/net/url;.openconnection"

"landroid/telephony/telephonymanager;.getline1number"

"landroid/telephony/smsmanager;.getdefault"

"landroid/telephony/smsmanager;.sendtextmessage"

"landroid/telephony/smsmessage;.getdisplayoriginatingaddress"

"landroid/telephony/smsmessage;.getmessagebody"

"landroid/telephony/smsmessage;.createfrompdu"

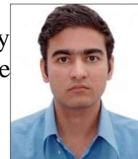
Table 4. Weight assignment to various features

Feature type	Weight assigned
Suspicious permissions	3
Suspicious permission combinations	5
Suspicious API combinations	5
Suspicious content URI	6
Manifest violation	7
Presence of executable	10

AUTHORS

Babu Rajesh V has been working for three years in the field of mobile security and malware analysis. His areas of interests include mobile security and embedded security

Phaninder Reddy has been working for two years in the field of mobile security and malware analysis. His areas of interests include machine learning and data analytics



Himanshu Pareek has around six years of experience in developing and design of security solutions related to small sized networks. He has research papers published on topics like malware detection based on behaviour and application modelling

Mahesh U Patil received master degree in electronics and communication. Presently he is working as Principal Technical Officer at Centre for Development of Advanced Computing. His research interests include Mobile Security and Embedded Systems.