AndroSAT: Security Analysis Tool for Android Applications

Saurabh Oberoi*, Weilong Song†, Amr M. Youssef‡
Concordia Institute for Information Systems Engineering
Concordia University
Montreal, Quebec

***Abstract*—With about 1.5 million Android device activations per day and billions of application installation from Google Play, Android is becoming one of the most widely used operating systems for smartphones and tablets. In this paper, we present *AndroSAT*, a Security Analysis Tool for Android applications. The developed framework allows us to efficiently experiment with different security aspects of Android Apps through the integration of (i) a static analysis module that scans Android Apps for malicious patterns. The static analysis process involves several steps such as n-gram analysis of dex files, de-compilation of the App, pattern search, and analysis of the AndroidManifest file; (ii) a dynamic analysis sandbox that executes Android Apps in a controlled virtual environment, which logs low-level interactions with the operating system. The effectiveness of the developed framework is confirmed by testing it on popular Apps collected from F-Droid, and malware samples obtained from a third party and the Android Malware Genome Project dataset. As a case study, we show how the analysis reports obtained from *AndroSAT* can be used for studying the frequency of use of different Android permissions and dynamic operations, detection of Android malware, and for generating cyber intelligence about domain names involved in mobile malware activities.**

Keywords–Android Security; Static Analysis; Dynamic Analysis.

I. INTRODUCTION

According to a recent report from Juniper Networks [1], smartphone sales have increased by 50% year-on-year. In the third quarter of 2013, more than 250 million smartphones were sold worldwide. This rapid increase of smartphone usage has moved the focus of many attackers and malware writers from desktop computers to smartphones. Today, mobile malware is far more widespread, and far more dangerous, especially in Bring Your Own Device (BYOD) arrangements where mobile devices, which are often owned by users who act as defacto administrators, are being used for critical business and are also being integrated into enterprises, government organizations and military networks [2][3].

Android, being one of the utmost market share holders, not only for smartphones and tablets, but also in other fields such as automotive integration, wearables, smart TVs and video gaming systems, is likely to be facing the highest threat from malware writers. As an open-source platform, Android is arguably more vulnerable to malicious attacks than many other platforms. According to the report from Juniper Networks [1], mobile malware grew 614% for a total of 276,250 malicious Apps from March 2012 to March 2013. Another recent report from Kaspersky [4] shows that 99% of all mobile-malware in the wild is attacking the Android platform. Kaspersky also mentioned that mobile malware is no longer an act of an individual hacker; some rogue companies are investing time and money to perform malicious acts such as stealing credit card details and launching phishing attacks, to gain profit. According to the Kaspersky report, the number of unique banking trojans raised from 67 to 1321 from the start to the end of 2013. Thousands of users were convinced to pay millions of dollars due to the gradual dissemination of infected Apps. In extreme cases, an application with malicious intent can do more than just sending premium text messages–they can turn a phone into a spying tool. These spying tools can track the current location of a smartphone, make phone calls, send and receive text messages and send stolen private information to remote servers without raising any alarm.

In this paper, we present a Security Analysis Tool for Android applications, named *AndroSAT*. The developed framework allows us to experiment with different security aspects of Android Apps. In particular, *AndroSAT* comprises of:

- A static analysis module that scans Android Apps for malicious patterns (e.g., potentially malicious API calls and URLs). This process involves several steps such as n-gram analysis of dex files, de-compilation of the App, pattern search, and extracting security relevant information from the AndroidManifest files.
- A dynamic analysis sandbox that executes Android Apps in a controlled virtual environment, which logs low-level interactions with the operating system.
- Analysis tools and Add-ons for investigating the output of the static and dynamic analysis modules.

In order to demonstrate the effectiveness of our framework, we tested it on popular Apps collected from F-Droid [5], which is a Free and Open Source Software (FOSS) repository for Android applications, and a malware dataset obtained from a third party as well as from the Android Malware Genome Project. The reports produced by our analysis were used to perform three case studies that aim to investigate the frequency of use of different Android permissions and dynamic operations, detection of malicious Apps and generating cyber intelligence about domain names involved in mobile malware activities. The results obtained by the first case study can be utilized to narrow down the list of features that can be used to determine malicious patterns. In the classification experiment, using the features extracted from our analysis reports, we applied feature space reduction, and then performed classification on the resultant dataset. As will be explained in Section V, the obtained classification results are very promising. Finally, in our cyber-intelligence gathering experiment, we used the IP addresses recorded during the static and dynamic analysis of malware Apps to produce a graphical representation of the geographical locations of possible malicious servers (and their ISPs) that communicate with malicious Apps. These three experiments show the versatility as well as the wide variety of possible usages for the information obtained by *AndroSAT*.

The rest of the paper is organized as follows. In the next section, we discuss some related work. A brief review of

 ISBN: 978-1-61208-376-6

Android and its security model is provided in Section III. Section IV details the design of our framework and explains the static and dynamic analysis modules. Our experimental results are presented in Section V. Finally, our conclusion is given in Section VI.

II. Related Work

Due to sudden increase in the number of Android malware, researchers too have moved their focus and resources towards securing the Android platform from this rising threat. Blasing *et al.* [6] developed a system named AASandbox that utilizes a loadable kernel module to monitor system and library calls for the purpose of analyzing Android applications. Wu *et al.* [7] developed a system named DroidMat that extracts the information from an AndroidManifest and, based on the collected information, it drills down to trace the application programming interface(API) calls related to the used permissions. It then uses different clustering techniques to identify the intentions of the Android malware. Reina *et al.* [8] developed a system named CopperDroid, which performs the dynamic analysis of an Android application based upon the invoked system calls. They claimed that their system can detect the behavior of an application whether it was initiated through Java, Java Native Interface or native code execution. Burguera *et al.* [9] focused on identifying system calls made by Android applications and developed a tool named Crowdroid to extract the system calls and then categorize these system calls into either malicious or benign by using K-means clustering. DroidRanger, a system proposed in [10], consists of a permission-based behavioral footprinting scheme that detects new samples of known Android malware families and a heuristics-based filtering scheme that identifies certain inherent behaviors of unknown malicious families.

Spreitzenbarth *et al.* [11] developed a system named Mobile-Sandbox, which is designed to perform integrated analysis and some specific techniques to log calls to non-Java APIs. Alazab *et al.* [12] used DroidBox, a dynamic analysis tool that generates logs, behavior graph and treemap graphs to explain the behavior of an Android App. They collected 33 malicious applications grouped into different families and scanned them with different antivirus. They combined the graphs of the applications within the same family to verify if the graphs eventually reflect the family and then compared it with results from different antivirus companies. Another dynamic analysis tool named TaintDroid is presented in [13], which is capable of simultaneously tracking multiple sources of sensitive data accessed by Android application. In the work reported in [14][15], n-gram features are extracted from benign and malware executables in Windows PE format. The extracted features are then used to generate model with classifiers supported by WEKA.

Compared to other related work, one key feature in our system, *AndroSAT*, is that the developed sandbox allows not only for observing and recording of relevant activities performed by the apps (e.g., data sent or received over the network, data read from or written to files, and sent text messages) but also manipulating, as well as instrumenting the Android emulator. These modifications were made to the Android emulator in order to evade simple detection techniques used by malware writers.

III. Android Overview

Android is an emerging platform with about 19 different versions till date [16]. Table I shows different Android versions with their corresponding release date. As shown in Figure 1, the Android framework is built over Linux kernel [17] that controls and governs all the hardware drivers such as audio, camera and display drivers. It contains open-source libraries such as SQLite, which is used for database purposes, and SSL library that is essential to use the Secure Sockets Layer protocol. The Android architecture contains *Dalvik Virtual Machine* (DVM), which works similar to the Java Virtual Machine (JVM). However, DVM executes *.dex* files whereas JVM executes *.class* files.

TABLE I. ANDROID VERSION HISTORY

Android Version	OS Name	Release Date
1.0	Alpha	09/2008
1.1	Beta	02/2009
1.5	Cupcake	04/2009
1.6	Donut	09/2009
2.0-2.1	Eclair	10/2009
2.2	Froyo	05/2010
2.3.x	Gingerbread	12/2011
3.1-3.2	Honeycomb	02/2011
4.0.3-4.0.4	Ice Cream Sandwich	10/2011
4.1.x-4.3	Jelly Bean	08/2012
4.4	KitKat	09/2013

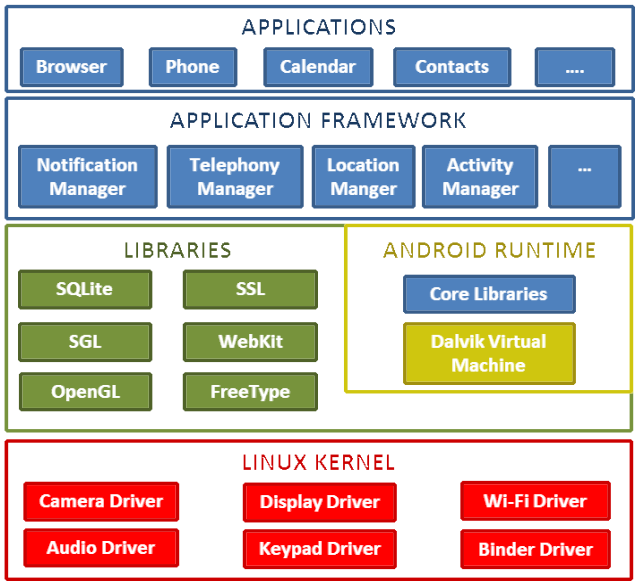

Figure 1. Android architecture [17]

Every application runs in its own Dalvik virtual environment or sandbox in order to avoid possible interference between applications and every virtual environment running an application is assigned a unique *User-ID* (UID). The application layer consists of the software applications with which users interact. This layer communicates with the application framework to perform different activities. This application framework consists of different managers, which are used by an Android application. For example, if an application needs access to an

 ISBN: 978-1-61208-376-6

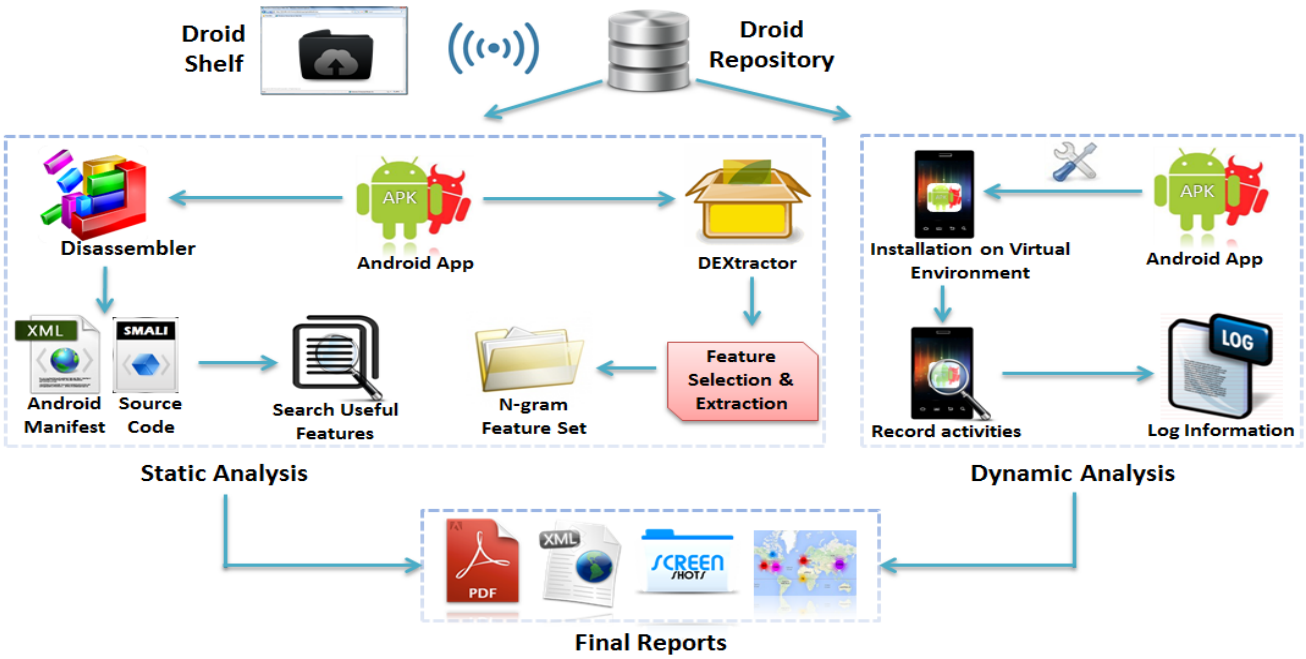

Figure 2. Overview of *AndroSAT*

incoming/outgoing phone call, it needs to access *Telephony-Manager*. Similarly, if an application needs to pop-up some notifications, it should interact with *NotificationManager*.

An Android application, also known as an APK package, consists of AndroidManifest.xml, res, META-INF, assets and classes.dex files. The AndroidManifest.xml file contains information about supported versions, required-permissions, services-used, receivers-used, and features-used [17]. META-INF contains the certificate of the application developer, resource directory (res) contains the graphics used by an applications such as background, icon and layout [17]. Assets directory contains the files used by an Android application, such as SQLite database and images. The classes.dex file is an executable file in a format that is optimized for resource constrained systems.

IV. System Overview

In this section, we provide an overview of *AndroSAT*. A local web-server is setup where we can upload the Android applications into our Droid-Repository (MySQL database of Android applications to be analyzed) through a PHP webpage (DroidShelf). As depicted in Figure 2, *AndroSAT* includes two main modules, namely a static analysis module and a dynamic analysis module, which are used together to produce analysis reports in both XML and pdf formats. The produced XML reports can then processed using several add-ons and analysis tools.

A. Static Analysis

Static analysis techniques aim to analyze Android Apps without executing them. The objective of these techniques is to understand an application and predict what kind of operations and functionalities might be performed by it without executing it. Different forms of static analysis have proved to be very

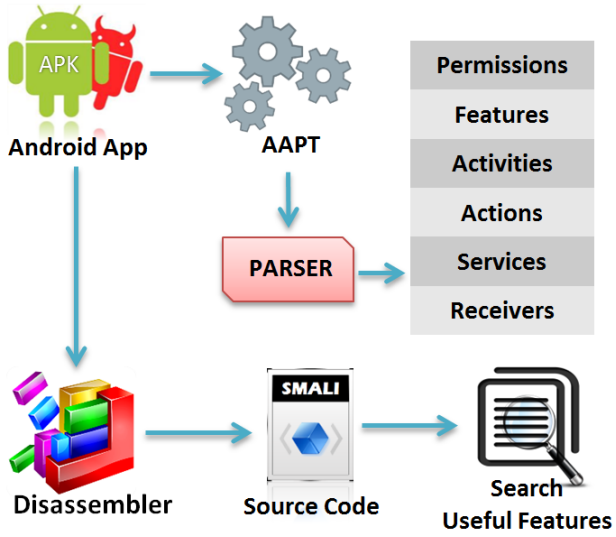

Figure 3. Overview of the static analysis module

useful in detecting malicious Apps. As shown in Figure 2 and Figure 3, the process of static analysis involves several steps such as extracting n-gram statistics of .dex files, disassembling the application, performing pattern search for malicious API calls and URLs, and extracting relevant information (such as used permissions, activities, intents and actions, services, and receivers) from the AndroidManifest file. In order to perform the static analysis process, the analyzed application is first fetched from the Droid-Repository. Then, data from *AndroidManifest* file is extracted from the APK package using the Android Asset Packaging Tool (AAPT). AAPT is used for compiling the resource files during the process of Android App

 ISBN: 978-1-61208-376-6

development, and is included in the Android SDK package [18]. After the n-gram statistics is evaluated from the .dex file, the application is fed to the disassembler, which disassembles the application APK package to obtain the SMALI and Java code of the application. The disassembly process is performed using APKTool [19] and Apk2java [20], which are open source reverse engineering tools. Once the source code is obtained from the Android application undergoing analysis, we search for malicious functions/API calls, URLs and IP addresses. In what follows we provide some further details on the n-gram analysis process and the different features extracted from both AndroidManifest and source code.

1) N-gram Analysis: Different forms of n-gram analysis have been previously used for malware detection in the Windows and Linux/Unix environments. Different from Portable Executables (PE) but similar to MSI packages in Windows, Android OS has Android application package file (APK) as the file format used to install application software. The APK file can be looked at as a zip compression package containing all of the application bytecode including *classes.dex* file, compiled code libraries, application resource (such as images, and configuration files), and an XML file, called AndroidManifest. The *classes.dex* file holds all of application bytecode and implementing any modification in the application behavior will lead to a change in this file. The process of n-gram analysis is performed by extracting application bytecode files (i.e., *classes.dex*), calculating byte n-gram, and then performing a dimensionality reduction step for these calculated n-gram features. The byte n-grams are generated from the overlapping substrings collected using a sliding window where a window of fixed size slides one byte every time. The n-gram feature extraction captures the frequency of substrings of length n byte. Since the total number of extracted features is very large, we apply feature reduction to select appropriate features for malware detection. We chose Classwise Document Frequency [14] as our feature selection criterion. *AndroSAT* applies feature reduction on bigram and trigram sorted by CDF value and top k features are selected. The obtained feature vectors are saved in the analysis reports and can then be used as inputs for classifier to generate models for malicious Apps. Surprisingly, as will be shown in the experimental results Section V, applying this simple analysis method to .dex files even without any pre-processing or normalization for the byte code yields very promising results and allows us to differentiate between malicious and benign applications with relatively very good accuracy.

2) Features extracted from the AndroidManifest file: Throughout the analysis process, the following features are extracted from the AndroidManifest file of analyzed applications:

- Requested Permissions: An Android application does not need any permission unless it is trying to use a private or system related resource/functionality of the underlying Android device. There are numerous permissions that developers can add into an Android application to provide better experience to users. Example of these permissions include CAMERA, VIBRATE, WRITE_EXTERNAL_STORAGE, RECEIVE_SMS, and SEND_SMS [5]. Permissions requested by an application inform user about what they can expect from the application and a smart user can easily realize if an application is asking for more than it should supposedly do. For example, an application claiming to show weather reports should raise suspicion if it requests a SEND_SMS permission.
- Features Used: An Android application can use hardware or software features. The features available (e.g., bluetooth, and camera) vary with different Android devices. Therefore, many applications use feature as a preference, i.e., they can still function even if the feature is not granted. Features come with a required attribute that helps a developer to specify whether the application can work normally with or without using the specific feature.
- Services-Used: Services-Used lists all the services recorded in the application AndroidManifest file. In an Android application, a service can be used to perform operations that need to run at the background for a long time and without any user interaction. The service keeps on running even if the user switches to another application. An attacker can make use of a service to perform malevolent activities without raising an alarm.
- Receivers Used: In Android, events send out a broadcast to all the applications to notify their occurrence. A broadcast is triggered once the event registered with the corresponding broadcast receiver [21] occurs. The main purpose of using a broadcast receiver is to receive a notification once an event occur. For example, if there is a new incoming message, a broadcast about the new incoming message is sent out and applications that use the corresponding receiver, i.e., *SMS_RECEIVED* receiver will get the incoming message. Malicious applications can use the broadcast receiver in numerous ways such as receive the incoming messages and monitor the phone state.
- Intents and Actions: An intent or action specifies the exact action performed by the broadcast receiver used in an application. Some of the most widely used broadcast receivers include SMS_RECEIVED, and BOOT_COMPLETED.
- Activities Used: Activities-used is a list of all the activities used in an Android application. In Android, every screen that is a part of an application and with which users can interact is known as an activity. An application can have more than one activity.

3) Feature Extraction from Source Code: In this section, we list the features extracted from the decompiled SMALI and Java source code.

- getLastKnownLocation: This function is used to get the last know location from a particular location provider. Getting this information does not require starting the location provider, which makes it more dangerous and invisible. Even if this information is stale, it is always useful in some contexts for malicious App developers.
- sendTextMessage: This function is most widely used by many malware developers to earn money or to send bulk messages while hiding their identity. The biggest advantage that attackers have when utilizing this function is that it sends text messages in the background and does not require any user intervention.

 ISBN: 978-1-61208-376-6

- getDeviceId: This function is used to obtain the *International Mobile Station Equipment Identity* (IMEI) number of the Android device. Every device has its own unique IMEI that can be used by the service provider to allow the device to access the network or block its access to the network. IMEIs of stolen phones are blacklisted so that they never get access to the network. An attacker with malevolent intentions can use this unique code to make a clone and then performs illegal activities or blacklists the IMEI so that the user can never put it back onto the cellular network.

All the features extracted by the static analysis module are then fed to a parser module in order to remove redundant data. The extracted relevant information is then saved in both XML and PDF formats.

B. Dynamic Analysis

The main advantage of the static analysis described above is that it can be performed relatively very efficiently without the need to execute the Apps and hence avoid any risk associated with executing malicious Apps. On the other hand, some malware writers use different obfuscation and cryptographic techniques that make it almost impossible for static analysis techniques to obtain useful information, which makes it essential to use dynamic analysis. Dynamic analysis is most widely used to analyze the behavior and interactions of an application with the operating system. Typically, dynamic analysis is performed using a virtual machine controlled environment in order to avoid any possible harm that can result from running the malware on actual mobile devices. Furthermore, using the virtual environment makes it easier to prepare a fresh image and install the new application in question on it for analysis.

The main disadvantage of dynamic analysis, however, is that the usefulness of the analysis is somewhat correlated to the length of the analysis interval and some malicious activities may not be invoked by the App during the, usually short, analysis interval either because the conditions to trigger these events do not happen during the dynamic analysis process or because the malicious App is able to detect that it is being monitored. Anti-debugging and virtual machine detection techniques have long been used by Windows malware writers. To make the virtual environment look like a genuine smartphone, we made some changes to the Android emulator, e.g., we modified IMEI, IMSI, SIM serial number, product, brand and other information related to the phone hardware.

During dynamic analysis, the application is installed onto the system and its activities or interactions are logged to analyze its actions. We use a sandbox to execute the Android application in question in a controlled virtual environment. Figure 4 shows an overview of our dynamic analysis module. The main part of this module is based on an open source dynamic analysis tool for Android applications named DroidBox [22]. However, as mentioned above, we performed some modifications in order to improve the resistance of the emulator against detection. *AndroSAT* launches the emulator using DroidBox that uses its own modified image making it possible to log the system and API level interactions of an Android application with the emulator. Once the emulator is up and running, the App is installed using *Android Debug Bridge* (ADB) for further analysis. Immediately after the successful

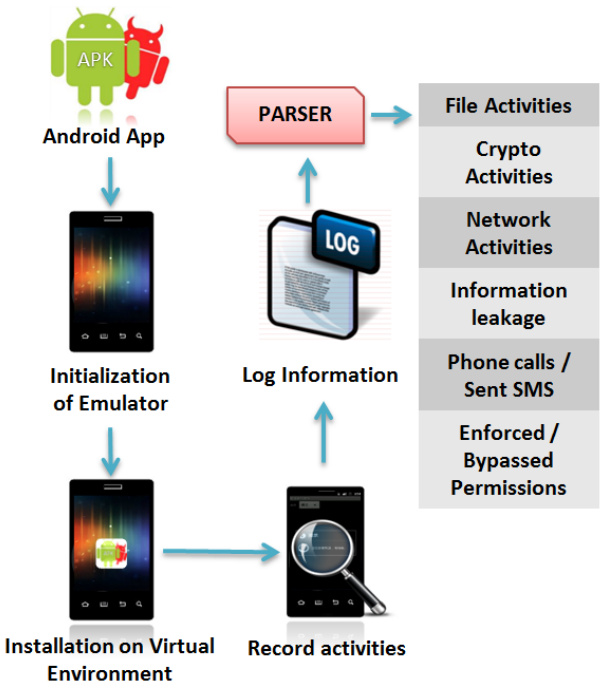

Figure 4. Overview of the dynamic analysis module

installation of the application, the module starts DroidBox to further analyze the APP for a configurable interval (the default is two minutes). Meanwhile, it launches the main activity of the installed application onto the emulator automatically, performs random gestures on it and takes screen shots of the application using the MonkeyRunner tool [23], while DroidBox consistently logs any system or API level interactions of the application with the operating system. The following features are collected during our dynamic analysis:

- File Activities: File activities consist of information regarding any file, which is read and/or written by the application. This information includes timestamps for these file activities, absolute path of accessed files and the data, which was written to/read from these files.
- Crypto Activities: Crypto Activities consist of information regarding any cryptographic techniques used by the application. It includes information regarding the type of operation (e.g., key generation, encryption, and decryption) performed by the application, algorithms used (e.g., AES, DES), key used and the data involved.
- Network Activities: It unveils the connections opened by the application, packets sent and received. It also provides detailed information about all these activities including the timestamp, source, destination and ports.
- Dex Class Initialized: In Android, an application can initialize a dex file, which is not a part of its own package. In the most malicious way, an application can download a dex file to the Android device and then executes it using the *DexClassLoader*. This way, an application under analysis will come out clean, which makes it almost impossible for any malware analyzer or sandbox to detect the malicious activities

 ISBN: 978-1-61208-376-6

performed by the application. DroidBox logs relevant details whenever an application initializes any dex class.

- Broadcast Receiver: As explained earlier, the use of broadcast receiver helps improve the user experience of an application. However, an attacker can use this functionality to easily gain access to private/critical data without raising an alarm in the users' mind. We log information regarding any broadcast receiver used by the application and record the name of the broadcast and the corresponding action.
- Started Services: Services play a very critical role in Android applications. They are used to execute the code in the background without raising an alarm. Started services provide the information about any service, which is started or initialized during the runtime of the application.
- Bypassed permissions: Lists the permission names, which are bypassed by the application. This aims to detect scenarios where an application can perform the task that needs a specific permission without explicitly using that permission. For example, an Android application can direct the browser to open a webpage without even using the Internet permission [24].
- Information Leakage: Information leakage can occur through files, SMS and network. Leakage may occur through a file if the application tries to write or read any confidential information (e.g., IMEI, IMSI, and phone number) of an Android device to or from a file. Leakage occurs through SMS if the information is sent through an SMS. Timestamp, phone number to which the information is sent, information type, and data involved are also logged. Leakage occurs through network if the application sends critical data over the Internet. Timestamp, destination, port used, information type and data involved is recorded. Detailed information about the absolute path of the file, timestamp, operation (read or write), information type (IMEI, IMSI or phone number) and data are logged.
- Sent SMS: If an Android application tries to send a text message, timestamp, phone number and the contents of the text message are logged.
- Phone call: If an Android application tries to make a phone call, timestamp and phone number are recorded.

Dynamic analysis module logs all these features into a text file, which is then sent to the parser module to remove any redundant data. The extracted relevant information is then saved in XML and PDF formats.

V. EVALUATION

The reports generated by our framework contain useful information regarding the analyzed Android applications, which in turn can be used in many Android security related applications. To confirm the effectiveness of our proposed framework, we analyzed a total of 1932 Android applications, out of which 970 are benign and 962 are malicious. We collected the malicious samples from the Android Malware Genome Project [25] and from another third party. The benign samples were obtained from F-Droid [18], which is a Free and Open Source Software (FOSS) repository for Android applications. We also verified the applications collected from F-Droid are benign using VirusTotal [26].

Results from the dynamic analysis show that 254 out of 962 (i.e., 26%) malicious applications and none of the 970 benign applications lead to private data leakage through network. Many malware writers use cryptographic techniques to hide the malicious payload in order to make it impossible for a signature based malware analyzer to understand the malicious intentions of an application. Among the analyzed Apps 41 out of 962 malicious applications and 2 out of 970 benign applications use cryptography at runtime. The experimental results also suggest that an Android application with different versions might have different package contents and hence the checksum of the packages might differ. However, the checksum of *classes.dex* file of some different versions came out to be the same. This tells us that malware writers might add junk data in the APK package to make an application look different while the content of *classes.dex* file remains the same.

We incorporated the reports generated by our framework and used them to perform three case studies, namely performing frequency analysis for the different permissions and operations used by Android Apps, cyber-intelligence and classification.

A. Frequency analysis of Android permissions and dynamic operations

Figure 5(a) shows the top 15 permissions used by the analyzed malicious applications and their frequency as compared to the benign ones. It is interesting to find out that some permissions are used by most of the malicious applications. As depicted in Figure 5(a) READ_PHONE_STATE permission is used by $\approx$86% of the malicious Apps as compared to $\approx$12.% of the benign Apps. This permission is most widely used to obtain system specific data such as IMEI, IMSI, phone number and SIM serial. Similarly, the frequency of use of INTERNET, ACCESS_WIFI_STATE, ACCESS_NETWORK_STATE, READ_SMS, and WRITE_SMS show noticeable differences. Figure 5(b) shows the top 15 permissions used by the analyzed benign applications and their frequency as compared to the analyzed malicious applications. In total, 962 malicious applications used 10,203 permissions, which come to an average of 10.6 permissions per application. On the other hand, 970 benign applications used 3,838 permissions, which come to an average of around 3.95 permissions per application. These results confirm that, on average, the number of permissions requested by a benign application is less than the number of permissions requested by a malicious application.

Figure 5(c) shows the top 15 dynamic operations performed by the analyzed malicious applications. As shown in the figure, there are many operations that are dominantly performed by the malicious applications. These include BroadcastReceiver(BOOT_COMPLETED), OpenNetworkConnection:80, and DataLeak_Network. Applications with malicious intents use BOOT_COMPLETED broadcast receiver to receive a notification whenever an Android device boots up so that they can perform the intended malicious activity or launch malicious services that keep running in the background. Another deciding factor is data leakage through network, which has a high occurrence in our malicious dataset, i.e., 254 as compared to 0 in the analyzed benign ones. Figure

 ISBN: 978-1-61208-376-6

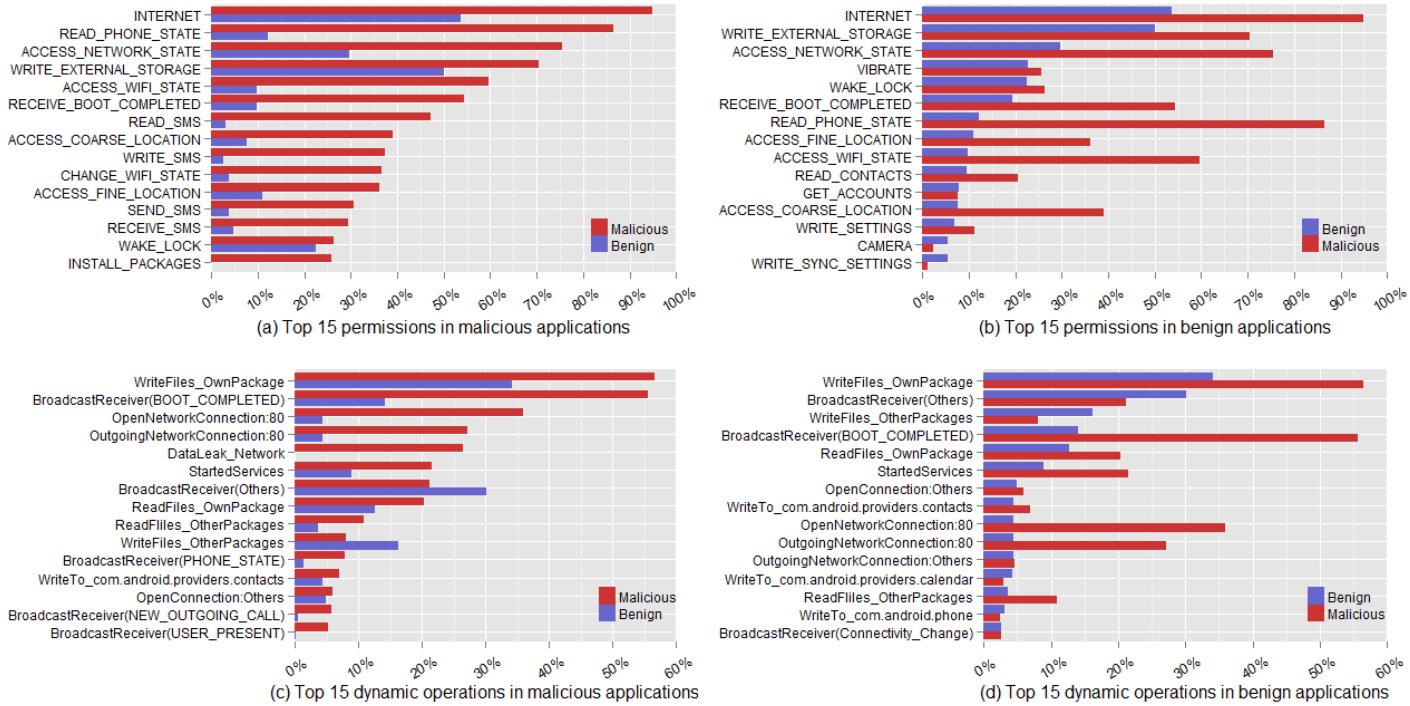

Figure 5. Most frequently used permissions and dynamic operations for the analyzed Apps

5(d) shows the top 15 dynamic operations performed by the benign applications in our dataset.

B. *Cyber-intelligence*

One of the main objectives of cyber-intelligence is to track sources of online threats. In this work, we used the URLs and IP addresses recorded during the static and dynamic analysis of malware Apps to produce a graphical representation of the geographical locations of possible malicious servers (and their ISPs) that communicate with these malicious Apps. Figure 6 shows a sample output of this analysis (IP addresses are not shown for privacy and liability concerns).

Figure 6. Geographical Presentation of the locations of suspected IPS

C. *Malware detection*

Throughout this experiment, we incorporated 134 static, 285 dynamic and 400 n-grams based features. We performed our classification task on 1932 Android applications using different combinations of these features, namely static analysis features, dynamic analysis features, n-grams based features, combination of static & dynamic features (S+D), combination of static & n-grams based features (S+N) and combination of dynamic & n-grams based features (D+N). We also combined features from all three analysis techniques, i.e., static, dynamic and *n*-grams (S+D+N) and performed feature space reduction using *classwise document frequency* to obtain a feature-set containing the top features for classification. We employed five different algorithms supported by WEKA [27] for classification with 10-fold cross-validation: SMO [28], IBK [28], J48 [28], AdaBoost1(J48 as base classifier) [28], and RandomForest [28]. Our experimental results show that AdaBoost1 and RandomForest models achieve a better accuracy compared to the other models. Figure 7 shows the results obtained for the five different feature sets in terms of accuracy, precision, and recall. From Figure 7(a), it is clear that *n*-gram features using AdaBoost1, D+N features using AdaBoost1 and S+D+N features using RandomForest provide the highest accuracy $\approx$98%. Figure 7(b) and Figure 7(c) shows the corresponding precision and recall, respectively. The low accuracy obtained when using dynamic analysis only can be explained by noting that, throughout our dynamic analysis process, we do not interact with the applications with carefully chosen gestures. Consequently, there is no guarantee that we check complete paths that can be traversed by the application or even a good portion of it. Furthermore, the short dynamic analysis interval might not be enough to trigger some of the bad events performed by malicious Apps. On the other hand, it should not be noted that the relatively high accuracy obtained with the combined features should also be interpreted with care since it might have resulted because of the limited variance in the

 ISBN: 978-1-61208-376-6

characteristics of the analyzed samples.

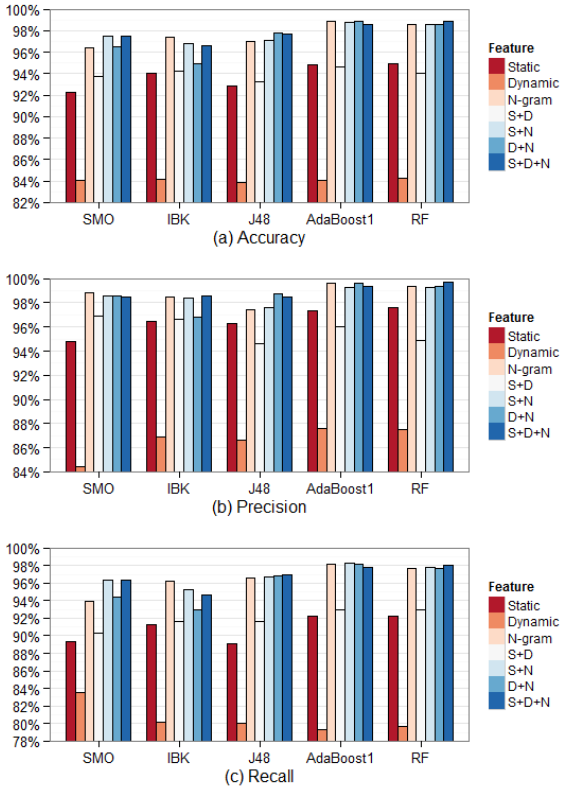

Figure 7. The classification results

VI. Conclusion

The increasing popularity of the Android operating system has led to sudden escalation in Android malware. In this work, we developed a framework to analyze Android applications using static and dynamic analysis techniques (*AndroSAT*). The effectiveness of *AndroSAT* was tested by analyzing a dataset of 1932 applications. The information obtained from the produced analysis reports proved to be very useful in many Android security related applications. In particular, we used the data in these reports to perform three case studies: analyzing the frequency of use of different Android permissions and dynamic operations for both malicious and benign Apps, producing cyber-intelligence information, and maleware detection. The implemented prototype can be further extended to allow for more useful add-ons that can be used to provide further investigation of the security of Android applications.

References

[1] "Third Annual Mobile Threats Report," 2013, URL: http://www.juniper.net/us/en/local/pdf/additional-resources/3rd-jnpr-mobile-threats-report-exec-summary.pdf [accessed: 2014-09-05].

[2] Q. Li and G. Clark, "Mobile security: a look ahead," Security & Privacy, IEEE, vol. 11, no. 1, 2013, pp. 78–81.

[3] C. Miller, "Mobile attacks and defense," Security & Privacy, IEEE, vol. 9, no. 4, 2011, pp. 68–70.

[4] "Kaspersky: forget lone hackers, mobile malware is serious business," Feb. 2014, URL: http://www.theguardian.com/technology/2014/feb/26/kaspersky-android-malware-banking-trojans [accessed: 2014-09-05].

[5] "Android Permissions," URL: http://developer.android.com/reference/android/Manifest.permission.html [accessed: 2014-09-05].

[6] T. Blasing, L. Batyuk, A. D. Schmidt, S. A. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in Proceedings of the 5th international conference on Malicious and unwanted software (MALWARE). IEEE, 2010, pp. 55–62.

[7] D. J. Wu, C. H. Mao, T. E. Wei, H. M. Lee, and K. P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in Proceedings of the Seventh Asia Joint Conference on Information Security (Asia JCIS). IEEE, 2012, pp. 62–69.

[8] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors," EuroSec, Apr. 2013.

[9] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. ACM, 2011, pp. 15–26.

[10] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in NDSS, 2012.

[11] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: having a deeper look into android applications," in Proceedings of the 28th Annual ACM Symposium on Applied Computing. ACM, 2013, pp. 1808–1815.

[12] M. Alazab, V. Monsamy, L. Batten, P. Lantz, and R. Tian, "Analysis of malicious and benign android applications," in Proceedings of the 32nd International Conference on Distributed Computing Systems Workshops (ICDCSW). IEEE, 2012, pp. 608–616.

[13] W. Enck, P. Gilbert, B. G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones," Communications of the ACM, vol. 57, no. 3, 2014, pp. 99–106.

[14] S. Jain and Y. K. Meena, "Byte level n–gram analysis for malware detection," in Computer Networks and Intelligent Computing. Springer, 2011, pp. 51–59.

[15] D. K. S. Reddy and A. K. Pujari, "N-gram analysis for computer virus detection," Journal in Computer Virology, vol. 2, no. 3, 2006, pp. 231–239.

[16] "Android version history," URL: http://en.wikipedia.org/wiki/Android_version_history [accessed: 2014-09-05].

[17] S. Brahler, "Analysis of the android architecture," Karlsruhe institute for technology, 2010.

[18] "The Android Asset Packaging Tool," URL: http://developer.android.com/tools/building/index.html [accessed: 2014-09-05].

[19] "Android APKTool: A tool for reverse engineering Android apk files," URL: https://code.google.com/p/android-apktool/ [accessed: 2014-09-05].

[20] "Apk2java: Batch file to automate apk decompilation process," URL: http://code.google.com/p/apk2java/ [accessed: 2014-09-05].

[21] "Android Broadcast Receiver," URL: http://developer.android.com/reference/android/content/BroadcastReceiver.html [accessed: 2014-09-05].

[22] "DroidBox: Android Application Sandbox," URL: https://code.google.com/p/droidbox/ [accessed: 2014-09-05].

[23] "MonkeyRunner," URL: http://developer.android.com/tools/help/monkeyrunner_concepts.html [accessed: 2014-09-05].

[24] A. Lineberry, D. L. Richardson, and T. Wyatt, "These aren't the Permissions you're Looking for," in DEFCON 18, Las Vegas, NV, 2010.

[25] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP). IEEE, 2012, pp. 95–109.

[26] "VirusTotal," URL: https://www.virustotal.com/ [accessed: 2014-09-05].

[27] "WEKA," URL: http://www.cs.waikato.ac.nz/ml/weka/ [accessed: 2014-09-05].

[28] I. H. Witten and E. Frank, Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann, 2005.

 ISBN: 978-1-61208-376-6