# AndroSAT: SECURITY ANALYSIS TOOL FOR ANDROID APPLICATIONS

SAURABH OBEROI

A THESIS

IN

THE CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE IN INFORMATION SYSTEMS

SECURITY

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

JUNE 2014

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: Saurabh Oberoi

Entitled: AndroSAT: Security Analysis Tool for Android Applications

and submitted in partial fulfillment of the requirements for the degree of

MASTER OF APPLIED SCIENCE IN INFORMATION SYSTEMS SECURITY

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

| | |
|---|---|
| Dr. A. Ben Hamza | Chair |
| Dr. L. Wang | Examiner |
| Dr. A. Bagchi | Examiner |
| Dr. A. Youssef | Supervisor |

Approved by ______________________________
Chair of Department or Graduate Program Director

______________________________
Dean of Faculty

Date June 26, 2014

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Saurabh Oberoi**

Entitled: **AndroSAT: Security Analysis Tool for Android Applications**

and submitted in partial fulfillment of the requirements for the degree of

**Master of Applied Science in Information Systems Security**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair

______________________________ Examiner

______________________________ Examiner

______________________________ Examiner

______________________________ Supervisor

______________________________ Co-supervisor

Approved ______________________________

Chair of Department or Graduate Program Director

__________ 20 ____ ______________________________

Dr. Christopher Trueman, Dean

Faculty of Engineering and Computer Science

# ABSTRACT

AndroSAT: Security Analysis Tool for Android Applications

Saurabh Oberoi

With about 1.5 million Android device activations per day and billions of applications installation from Google Play, Android is becoming one of the most widely used operating systems for smartphones and tablets.

Besides typical personal usages, Android mobile devices are also being integrated into enterprises, government organizations, and military networks. Consequently, these devices hold valuable sensitive information which makes them face the same level of malicious attacks that have targeted the desktop environments over the past three decades.

In this thesis, 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, and detection of Android malware.

# Acknowledgments

First of all, I would like to express my deepest gratitude to my supervisor, Dr. Amr Youssef, for his constant support, heartily guidance and enduring patience during my graduate study. This thesis would not have been possible without his help. His attitude and enthusiasm for scientific and academic research will always be my role model.

I also wish to express my appreciation to all the faculty and people at Concordia Institute for Information Systems Engineering for having such a warm and cosy working environment. To each of my professors, I owe a great debt of gratitude for their wonderful teaching, which has helped me in reaching this stage.

I thank my grandmother, parents and sister for teaching me valuable lessons of life. I also thank my fiancée who always kept me going and helped me in my hard times.

Last, but not least, I would like to thank my colleagues at Concordia University, specifically, Song Weilong, Eman Alzahrani, Rohit Upadhyay, Maryam Asgariazad and Dhruv Jariwala for making my time at the University and in Montreal pleasurable and memorable.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Android is one of the most popular operating system these days. It powers more than a billion smart devices that include phones, tablets, wearables and consoles [5]. Around one million android devices are activated every day and more than one billion applications are downloaded every month, worldwide [6]. Android powered devices have replaced everyday use of laptops and desktops. Most of the smartphones these days have a better hardware configuration as compared to the laptop and desktop computers used in the 90's.

Android is the biggest open source mobile operating system. Its open source nature makes it flexible and customizable while maintaining the easy to use quality [5]. On the other hand, being an open source platform, Android opens itself to numerous amount of sophisticated attacks. At the same time, it also gives an opportunity to the developers to quickly learn its possible vulnerabilities and keep it vaccinated against attacks. Overall, being an open source helps Android to remain up to date against attacks as compared to many other mobile platforms and, in turn, Android can build strong prevention and detection techniques.

Over the past years, Android applications with malicious intent were observed not only at the third party app stores but also at the Google Play store. These malicious applications include virus, worms, Trojans, remote access tools, botnets and coin-miners. Although

Google has exerted a large effort to get rid of malicious applications from the Google Play store, there were thousands of victims before Google realized the malicious existence of such malware. All these malicious applications were proved to be dangerous to the smartphone users as they aimed for stealing the personal information of users including contacts, emails, pictures, text messages, call logs and banking information. One of the most dangerous types of malware is remote access tools, also known as RATs, which can remotely control a smartphone without the user's knowledge and installation of such malware does not even need a rooted device.

## 1.1 Motivation

According to a recent report from Juniper Research [30], smartphone sales have increased by 49% year-on-year since the third quarter of 2012. 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 [32, 35].

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 biggest level of threat from malware writers. As an open-source platform, Android is arguably more vulnerable to malicious attacks than many other platforms. According to a report from Juniper Networks [29], mobile malware grew 614% for a total of 276,250 malicious apps from March 2012 to March 2013. Another recent report from Kaspersky [17] 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 has risen from 67 to 1321 from the start to the end of 2013. Thousands of users were forced 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.

## 1.2 Objectives

Throughout this work, we focus on building a prototype to analyze Android applications and record their interaction with the Android emulator. The idea is to make the whole process automated so as to keep the Android apps security analysis easier and faster by analyzing numerous Android applications at once without manual intervention.

The developed prototype aims at recording security relevant information that can be obtained by observing the actions performed by Android apps when they run on Android devices. This information can then be used to perform different case studies or infer the intentions of an Android app.

The objective of this work can be summarized as follows:

- Design and implement a static analysis tool for Android apps
- Design and implement a dynamic analysis tool for Android apps
- Integrate static and dynamic analysis tools in order to automate the whole analysis procedure

- Perform case studies on information collected during static and dynamic analysis of Android apps in order to confirm the effectiveness of the developed framework

## 1.3 Contributions

In this thesis, 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. This process involves several steps such as *n*-gram analysis of dex file (compiling an Android application generates a dex file, also known as Dalvik executable, as a part of the APK package), 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. 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. Many modifications were made to the Android emulator in order to evade simple detection techniques used by malware writers.

- 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 [22] 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, learning about the deciding factors in distinguishing between benign and malicious apps, interpreting data collected during the static and dynamic analysis of Android 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. The obtained classification results are very promising. All these experiments show the versatility as well as the wide variety of possible usages for the information obtained by *AndroSAT*.

## 1.4 Thesis Organization

The rest of this thesis is organized as follows. Android security, malware on Android and related work are reviewed in the next chapter. In chapter 3, we present *AndroSAT*, a Security Analysis Tool for Android applications. In chapter 4, we present the case studies on security relevant information obtained during static and dynamic analysis of Android applications in order to confirm the effectiveness of the developed framework. Finally, in chapter 5, we present a summary of our research and some possible future research directions.

# Chapter 2

# Background

## 2.1 Android Overview

Android is an emerging platform with about 19 different versions till date [47]. Table 1. shows different Android versions and their corresponding release date. As shown in Figure 1, the Android framework is built over a Linux kernel [15] which controls and governs all the hardware drivers such as audio, camera and display drivers. Android contains open-source libraries such as SQLite, which is used for database purposes, and SSL library which 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. 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 Android apps. For example, if an app needs access to incoming/outgoing phone calls, it needs to access

| 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 |

Table 1: Android version history [47]

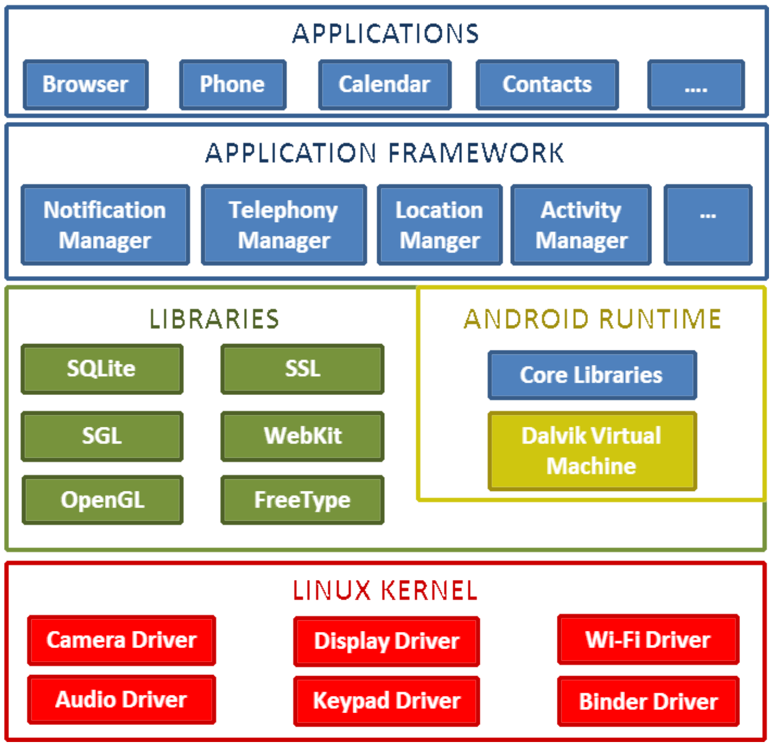


Figure 1: Android Architecture [15]

*TelephonyManager*. Similarly, if an app 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 [15]. 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 [15]. The assets directory contains the files used by an Android application, for example: SQLite database, and images. The classes.dex file is an executable file in a format that is optimized for resource constrained systems.

## 2.2 Android Security

The multi-layered security architecture of Android gains it a significant advantage and allows it to become secure compared to its counterparts. This multi-layer security architecture keeps it secure from apps with malicious intent. Figure 2 depicts the multi-layered security architecture of the Android platform. Android permissions is one of the layers in the secure operating system multi-layered architecture. However, the open source nature of Android makes it somewhat dependent upon the users and developers to do the right thing. To perform any given task, an application needs control over the users data and the device resources. At the time of the application installation, the users are asked to grant these permissions to the application which makes the operating system secure provided that users always manage to make the right decision at this step. Users can deny or allow all the permissions requested by an app. In other words, users have the privilege to grant a lot of permissions as requested by an application and may access the depths of Android operating system if the device is rooted.

To retain flexibility as an operating system, Android allows the users to access system

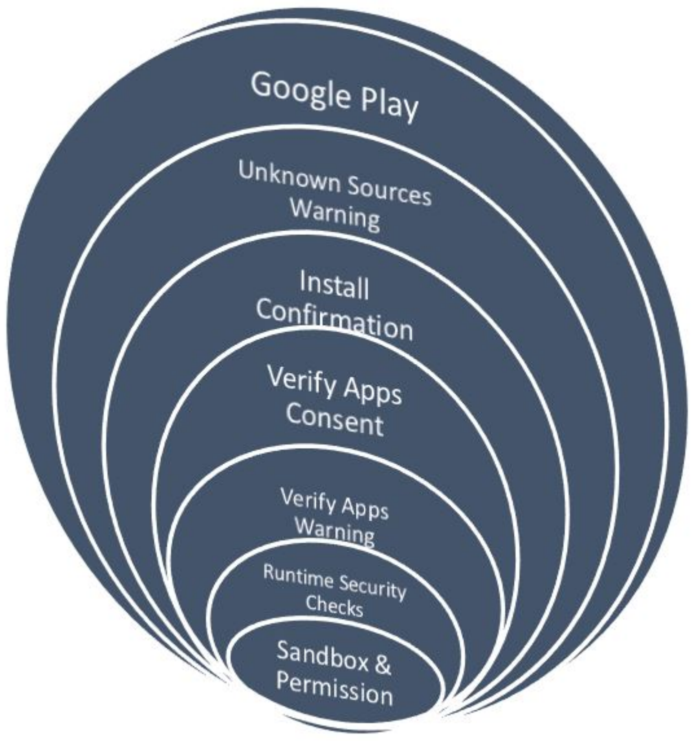


Figure 2: Multi-layered Security Architecture of Android Platform [33]

sensitive data or functionality by granting corresponding permission to an app. However, as a secure operating system Android also tries to keep sensitive data and functionalities safe from the mistakes of the users by warning them of possible malicious outcomes. For example, when a new application is installed, Android advices the user for possible issues which might occur if a particular permission is granted. Despite of such warnings, Android devices usually get open to a lot of malicious applications not because Android is insecure but because users make it insecure. However, security-savvy developers and users can keep the Android operating system more secure and flexible to use in day to day life operations. Android security can be divided into three different levels:

- Secure Android Operating System Development
- System Level Security
- Application Level Security

### 2.2.1 Secure Android Operating System Development

According to Google, for making and keeping an operating system secure, it is always necessary to think about security from the very first development steps of the operating system. The key steps in secure Android operating system development process are listed below [11]:

- **Design Review :** The security process starts from the very first step in the operating system life cycle. Design of the operating system is then reviewed by chief security professionals at Android [11]. The development process of the Android operating system goes to the next steps only if the design passes all the security aspects of an operating system.

- **Penetration Testing and Code Review :** During the development process of the Android operating system, it goes through plenty of pen-testing processes. The code is reviewed and penetrated by the Android Security Team, Google Information Security Engineering team, and independent security consultants [11]. The main aim of these tests and reviews is to check the code for possible vulnerabilities and known weaknesses to fix them before moving to the next steps.

- **Open Source and Community Review :** Google allows the Android operating system to be reviewed by external security professionals, security developers, code reviewers and secure code reviewing communities [11]. After all, security by obscurity does not work and it is better to get scrutiny done by experts.

- **Incident Response :** Even after shipping the operating system, security issues might occur in an operating system. New vulnerabilities might get discovered while the product is shipped and being used by millions of users. Android has a special response team for all vulnerabilities that are discovered once the operating system is live. The goal of this team is to eradicate any vulnerability that is discovered, as quickly as possible. The usual tasks of this team include fixing the vulnerability, releasing fixed updates to the users and removing the malicious applications from the Google Play store [11].

### 2.2.2 System Level Security

The Android platform provides an extra layer of security at the operating system level. Some of the security aspects that come under system level security of Android platform are mentioned below:

- **Linux Security :** The Android platform is based upon a Linux kernel and consequently, it inherits all the security aspects of this Linux Kernel [11]. The Linux platform is also an open source platform and is used by millions of users and security experts. Being an open source platform, Linux was built, designed, researched, tested and fixed by security experts all over the planet which makes it a reliable, robust and trusted platform.

- **SE Linux :** Android applies the functionalities of Security Enhanced Linux to make the operating system more secure and reliable [11]. SE Linux is used most widely in the areas which need very high security like government servers, military systems and other security oriented departments. In Android 4.3 and later versions, discretionary access control environment is replaced by mandatory access control [13]. Mandatory access control limits an app with root privileges to write to raw block device specified in the associated policy. Hence, overcoming the issues with discretionary access control environments where an app can write to any raw block device.

- **Root Access :** Generally, no third party application can gain root access to an Android device. Only the kernel and few core applications have root access and can execute with root privileges [11]. An application with root privileges can access everything on an Android platform whether it is another application's data, user's private data or private banking information. With root privileges, any application can take over the whole operating system and learn about all the information stored on that device. In this case, even encryption does not work because the decryption key itself is stored onto the device which is controlled by the rogue application.

For any application to gain root access, it needs the operating system to be rooted which can be easily done by installing a custom OS image onto the device. With a rooted device, users can access extra functionalities of the Android device - for example, using a custom firewall, Tor browser, packet sniffers, ad blockers and CPU tuners.

- **Application Sandbox :** Every Android application installed on the operating system is assigned a unique User ID (UID) and each user operates under a separate process. This creates a sandbox for every application which means that each application runs in its own Dalvik virtual environment or sandbox in order to avoid possible interference between applications [11].

  This technique is a bit different than that of the usual Linux operating system and can only be bypassed if the Linux kernel is compromised. The application sandbox keeps the application secure from other applications and keeps the application specific data/information secure and private. It would be a disaster if a third party application can have control over banking applications or if a malicious application takes control of the e-mail clients. The idea of keeping the applications in their own sandbox keeps the operating system secure yet simple.

- **Safe Mode :** All devices powered by Android can be booted in a safe mode which starts the operating system in an environment which is free from third party applications where users can only access pre-installed applications which come with the operating system [11]. As this mode is free from third party applications, it is extremely secure and is vaccinated against malicious attacks.

- **Cryptography :** Android offers cryptographic techniques which can be used by applications to keep the sensitive data, such as credentials and banking information, secure from any external or network intrusions. These techniques use cryptographic primitives such AES, SHA, RSA and DSA [11].

- **Password Protection :** Android protects the data from unauthorized access through different device locking techniques. There are six different ways of locking an Android device which include swipe, face unlock, face and voice, pattern, PIN and password.

### 2.2.3 Application Level Security

- **Applications from Unknown Sources :** By default, the Android platform does not allow any application from third party stores to be installed on the device unless the user specifically tells it to. Android gives an option to allow installation of applications from sources other than the Google Play store. Since it is considered insecure by Android, this functionality is not activated by default and can only be activated manually with the consent of the user.

- **Application Signing :** Every application must be signed by its developer before it is uploaded to the Google Play store [11]. Signing helps to uniquely identify the author of the application and makes it easier for the Google Play store to keep track of the application's developer for security purposes.

  While the application is installed onto the Android device, its certificate is verified with the signature to make sure that the integrity of the application is maintained. Application signing is used to assign the applications to the application sandbox. Thus, two applications with similar application signature will share the same application sandbox.

- **Android Permission Model :** On an Android device, no application can access user or system sensitive data of an Android device [11]. An application, by default, uses no permissions until and unless it needs access over user or system specific resources. The user can either allow or deny all the permissions asked by an application. As of now, there is no feature like denying or allowing only a specific subset of the permissions requested by an application. Once the permissions requested by an application are granted, there is no way to take them off from that application.

- **Cost-Sensitive APIs :** Android distinguishes the APIs which might incur cost to the users in some sense and names them Cost-Sensitive APIs [11]. To use these APIs, the application should ask for their corresponding Android permissions. Some of the Cost-Sensitive APIs are mentioned below:

  - **SMS/MMS :** Android applications use this API to send or receive SMS/MMS. An application with malicious intent can subscribe to premium messaging services using this API.
  - **Network :** Using this API, some malicious applications may use cellular data to download huge amount of data using the victim's Android device and hence, incurring unwanted data usage cost.
  - **Telephony :** The application with malicious intent can use this API to make calls to premium numbers without the user's knowledge.
  - **In-App Purchase :** Many users set-up their credit cards on their Android devices to make the process of In-App purchase or buying an application easier and faster. Malicious applications using this API can cost the user a lot of money by making irrelevant purchases without the user's knowledge.

## 2.3 Evolution of Android Security over time

The main aim of the Android developers was to keep the Android platform secure yet simple. These developers have paid a lot of attention in keeping the platform secure from all the attacks and vulnerabilities that were discovered. They have maintained this security by not only maintaining the operating system level security but also application level security.

Android security has evolved over recent years with the new outcoming versions of the Android operating system. Some of the recently added security features in the Android platform to keep the platform secure from the known vulnerabilities/issues are:

- **Memory Management Enhancements :** The enhancements made to the memory management in the context of security are mentioned below [11]:
  - **Android 1.5 :**
    - ◇ Extensions to OpenBSD dlmalloc to prevent double free() vulnerabilities and to prevent exploits against heap corruption
    - ◇ OpenBSD calloc to prevent integer overflows during memory allocation
    - ◇ ProPolice to prevent stack buffer overruns
    - ◇ safe_iop to reduce integer overflows
  - **Android 2.3 :**
    - ◇ Hardware-based No eXecute (NX) to prevent code execution on the stack and heap
    - ◇ Linux mmap_min_addr to mitigate null pointer dereference privilege escalation
    - ◇ Format string vulnerability protections
  - **Android 4.0 :**

    - ◇ Address Space Layout Randomization to randomize key locations in memory

  - **Android 4.1 :**
    - ◇ Avoid leaking kernel addresses
    - ◇ Position Independent Executable support

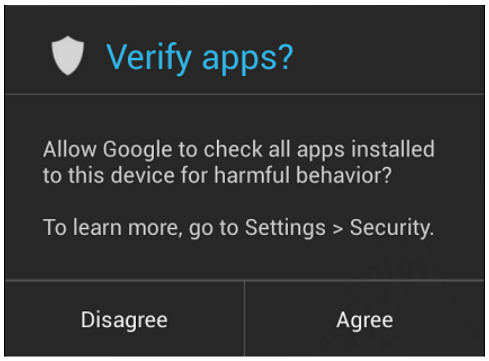
  - **Android 4.2 :**
    - ◇ FORTIFY_SOURCE for system code

  - **Android 4.3 :**
    - ◇ ADB Authentication
    - ◇ Android sandbox reinforced with SELinux

- **Device Encryption :** Full device encryption was introduced in Android 3.0 and is supported by all its following versions. The device can be encrypted using the dmcrypt implementation of AES128 with CBC and ESSIV:SHA256[1] [12]. The encryption key accepted by Android is only user entered password and not pattern as seen in lock-screen of an Android device. This password/encryption key is secured by AES 128 and, to keep it secure from password guessing attacks, a salt is added to it and repeated hash is generated using SHA1 [11].

- **Device Administration :** Device Administration API was introduced in Android 2.2 and is supported by all its following versions. Device Administration API allows an application to access the device as an administrator. An application with administrator privileges can evade its uninstallation and perform administrative tasks such as restoring the Android device to factory defaults. It is most widely used by the software installed on the laptop or desktop computers to communicate with the Android

[1]Encrypted salt-sector initialization vector technique is used to generate initialization vectors for block encryption in cipher-block chaining



Verify apps?

Allow Google to check all apps installed to this device for harmful behavior?

To learn more, go to Settings > Security.

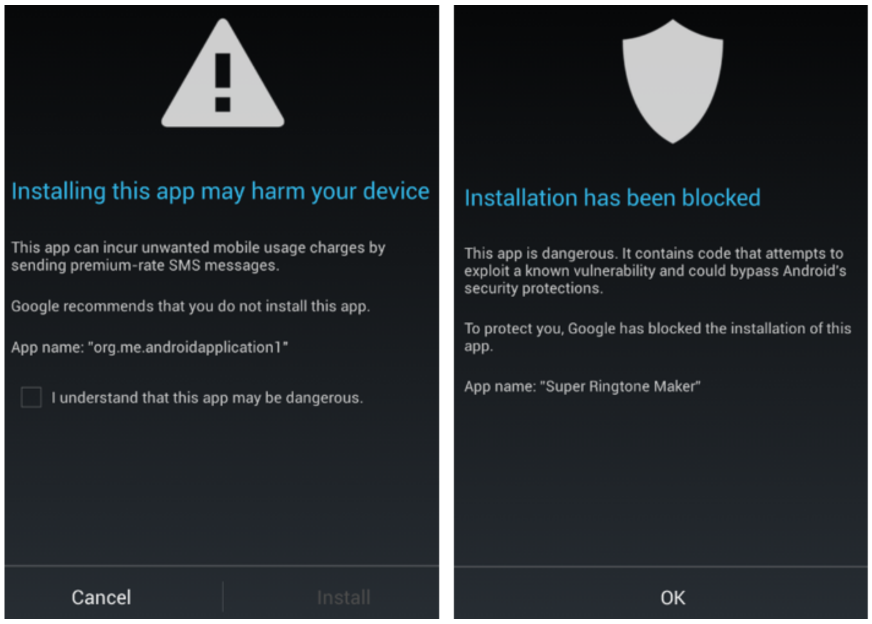
Figure 3: The GUI for the Verify Apps functionality [38]

device (for example, Samsung Kies and Sony PC Companion). Also, it is used to remotely wipe all the data from the device, restore the factory defaults, lock-screen and reset the lock code.

- **Application Verification :** The application verification (See Figure 3) was introduced in Android 4.2 and is supported by all its following versions. By selecting the option named "Verify Apps" under security settings in Android devices, users can verify the third party apps to identify known malware. Application verifier verifies the application with the Google's malicious applications database and if the verifier suspects it to be harmful for the device, it may stop the installation process and display a message as shown in Figure 4.

- **Premium SMS Warning :** This functionality was introduced in Android 4.2. The premium SMS warning warns the user if any application tries to send a premium SMS and if that SMS will incur premium charges as shown in Figure 5. In such cases, the user can either go ahead and send the message by selecting the Send button on the notification or cancel it by hitting Cancel.

- **Keychain :** Keychain was introduced in Android 4.0 and is available in all its following versions. Keychain is used to store private keys and their corresponding certificates in credential storage. These keys and certificates can then be accessed by


Installing this app may harm your device

This app can incur unwanted mobile usage charges by sending premium-rate SMS messages.

Google recommends that you do not install this app.

App name: "org.me.androidapplication1"

☐ I understand that this app may be dangerous.

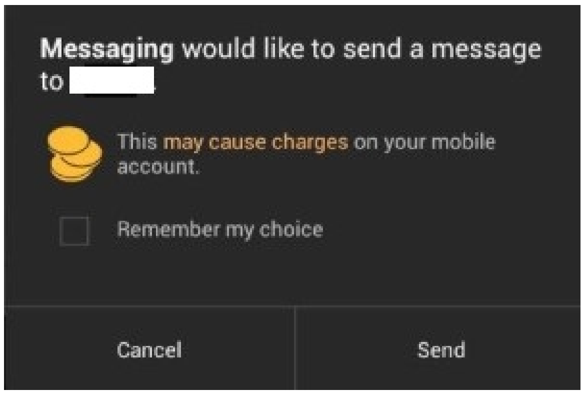
Installation has been blocked

This app is dangerous. It contains code that attempts to exploit a known vulnerability and could bypass Android's security protections.

To protect you, Google has blocked the installation of this app.

App name: "Super Ringtone Maker"

Figure 4: An Example for Verify Apps functionality detecting malicious application installation [38]


**Messaging** would like to send a message to [redacted]

This may cause charges on your mobile account.

☐ Remember my choice

Figure 5: Premium SMS Warning

their corresponding applications. This functionality provides user convenience by not prompting the users to enter the keys again and again.

- **READ_LOGS permission :** The *READ_LOGS* permission is used to allow any application to read all the system level log files which might contain privacy-sensitive data and activities performed on the Android device. These log files store all the actions performed by the user and the data involved in those actions. Every activity on an Android device including SMS sent, received with the data inside the message, call received, websites browsed, applications opened and credentials entered are written in the system log files. Recently, Android prevented third party applications from accessing the *READ_LOGS* permission due to security issues.

## 2.4 Past, Present and Future of Android Malware

With the increasing number of activated Android devices, attackers are trying to utilize every chance to gain control over these millions of devices either to earn money, mine electronic coins, perform a DDoS attack or maintain a sophisticated botnet. Being an open-source platform, Android faces many security challenges every day and Android developers are trying their best to keep the platform secure from any known or unknown malicious activity that might result into user inconvenience. Most Android users are unaware of the vulnerabilities of the platform and do not see any malicious activity coming their way. This does not mean that Android is perfectly secure. Android malware have made it quite clear that Android is not as safe as users consider it to be.

Millions of users were forced to pay billions of dollars due to infected Android devices. According to the Android developers, they have made their platform very powerful and gave all the power to its users. It is the user's responsibility to keep it safe and away from the malicious intentions. As seen in case of windows and many other platform, we individuals are lazy and tend to ignore warnings issued by the operating system. The ignorance of users has made them pay a hefty amount to attackers. Most of the malicious applications in the wild depend on social engineering techniques to get installed on the users' devices. We believe that with reasonable attention and understanding of the Android capabilities these threats can be avoided.

In this section, we discuss the past, present and possible future of Android malware.

### 2.4.1 Past

In the past, Android malware was proved to be a challenge for the Android developers. Malware writers abused all possible vulnerabilities of Android to earn money and gain control over millions of devices without the user's knowledge. Some of the malware that made their way to the Google Play store and proved out to be very dangerous are listed below:

- **DroidKungFu :** In June 2011, the first version of *DroidKungFu* was discovered by researchers at NC State University followed by its second and third versions in July and August, respectively [51]. These researchers were also successful in identifying the fourth, fifth and sixth variants of *DroidKungFu* named DroidKungFu4, DroidKungFuSapp and DroidKungFuUpdate, respectively. The medium of infection of *DroidKungFu* was through repackaged applications and it was considered the most sophisticated family of Android malware. *DroidKungFu* comprised of root exploits, C&C servers, shadow payloads and obfuscation.

  *DroidKungFu* was the first malware family that had an encrypted root exploits to avoid easy detection by signature based analysis schemes and anti-virus tools. Also, the encryption keys of the encrypted root exploits varied with changing versions. However, only four out of six variants had encrypted root exploits. *DroidKungFu* also made a use of C&C servers to steal sensitive information from the Android device and perform particular tasks on the device according to the attacker's will. C&C servers were included in all versions of *DroidKungFu*. Moreover, attackers paid a lot of attention to keep the C&C servers secret from the malware researchers and security professionals. They started from storing the server address in plaintext to encrypted text and finally encrypted with home-made encryption scheme [51]. The malware writers also increased the number of C&C servers.

Shadow payload was one of the important constituents of *DroidKungFu*. It contained an embedded application with the same malicious payload within the *DroidKungFu* package. *DroidKungFu* installs this embedded application onto the Android device once the root exploit succeeds. The reason behind installing the embedded application was that even if the original repackaged application is removed by the user, this application will be functional. Initially, this embedded application, once installed, showed up as a fake Google search icon which was later fixed and displayed no icon on the screen. *DroidKungFu* was heavily obfuscated malware which had strings, server address, embedded application and root exploits encrypted with changing encryption key with varying versions.

- **AnserverBot :** *AnserverBot* was initially distributed in Chinese third party Android application stores and it piggybacked on popular or paid Android applications [50]. It was discovered in September 2011 and was considered as a sophisticated malware because it exploited several vulnerabilities to stay undetected. The three main techniques used by *AnserverBot* were anti-analysis, C&C servers and security software detection [51].

  One of the main aims of *AnserverBot* was to keep it secure from analysis done by security researchers and professionals. To avoid any analysis, *AnserverBot* detects any changes done to the package which might result into change of signature or the integrity of the application which contains it. If the application's integrity is not maintained, *AnserverBot* won't unfold its payload. Also, the developers of *AnserverBot* did a lot of obfuscation to make the code very hard to read. The payload of *AnserverBot* was divided into three different parts, the first one was the application installed and the other two were embedded in a package sharing the same name. *AnserverBot* loads one of them dynamically without actually installing it and the other one was installed using the update attack. This proved out to be the biggest challenge for malware analyzers.

*AnserverBot* also paid a special attention to avoid its detection by any security related software installed on the Android device like anti-virus. In particular, *AnserverBot* included encrypted string of package name of three of the most popular anti-virus software at that time and performed a search operation to match the application installed on the device. If the matching application is found, *AnserverBot* kills it by calling the restartPackage function. This technique helps to avoid any detection from well-known anti-virus software installed on the Android device.

*AnserverBot* uses two types of C&C servers. One of them is just a normal C&C server that receives commands from the attacker whereas the other one is used to upload the payload and server address for the first C&C server. The second type is maintained by service providers like Sina and Baidu. *AnserverBot* connects to the popular blog site to obtain the encrypted new payload and server address. This ensures that even if the traditional C&C server is offline, its payload and server address will be updated with the second type of C&C server.

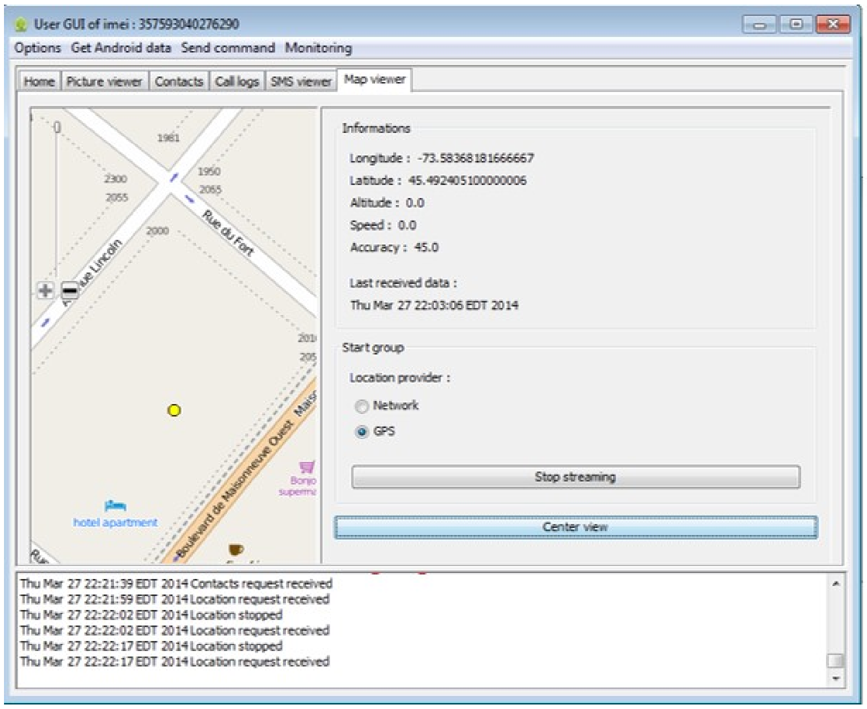
- **AndroRAT :** As the name suggests, *AndroRAT* is a remote access tool used to control Android devices remotely [44]. *AndroRAT* consists of a server and a client. The client application is the infected Android application that is installed onto the Android device whereas server application is a Jar file which can be executed on

Windows, MAC or Linux environment. Initially, *AndroRAT* was introduced as an open-source remote access tool in November 2012 which can gain almost complete control over an Android device. To remotely control the victim's Android device, the attacker needs to install the RAT application with modified server IP address and port onto the Android device. Once the application is installed and starts running on the device, it communicates with the server application and creates a connection with the server. After this, the server can perform the following actions over the controlled device:

- Monitor phone calls and SMSs
- Make phone calls and send SMSs
- Obtain all the contacts
- Download files stored on the device
- Obtain exact location of the device
- Display toast messages on device
- Take pictures
- Stream video or audio

Figure 6 shows an example illustrating the location of the Android device being controlled through *AndroRAT*. One of the problems with *AndroRAT* was the installation of the RAT application on the Android device. However, the recent introduction of *AndroRAT APK binder* has made this tool more dangerous and easier to spread. *AndroRAT APK binder* can be used to bind the RAT application with any popular or well-known application using the "bind+build" functionality of this tool. This makes the infection easier, faster and more widespread.

The biggest advantage of *AndroRAT* is that it does not need root privileges to perform the actions mentioned below. Moreover, variants to AndroRAT are introduced


User GUI of imei : 357593040276290
Options Get Android data Send command Monitoring
Home Picture viewer Contacts Call logs SMS viewer Map viewer
Rue du Fort
Rue Lincoln
Boulevard de Maisonneuve Ouest
hotel apartment
Stop streaming
Center view

Informations

Longitude : -73.58368181666667
Latitude : 45.492405100000006
Altitude : 0.0
Speed : 0.0
Accuracy : 45.0

Last received data :
Thu Mar 27 22:03:06 EDT 2014

Start group

Location provider :
Network
[✓ GPS]

```
Thu Mar 27 22:21:39 EDT 2014 Contacts request received
Thu Mar 27 22:21:59 EDT 2014 Location request received
Thu Mar 27 22:22:02 EDT 2014 Location stopped
Thu Mar 27 22:22:02 EDT 2014 Location request received
Thu Mar 27 22:22:17 EDT 2014 Location stopped
Thu Mar 27 22:22:17 EDT 2014 Location request received
```

Figure 6: Server application accessing exact location of the client RAT

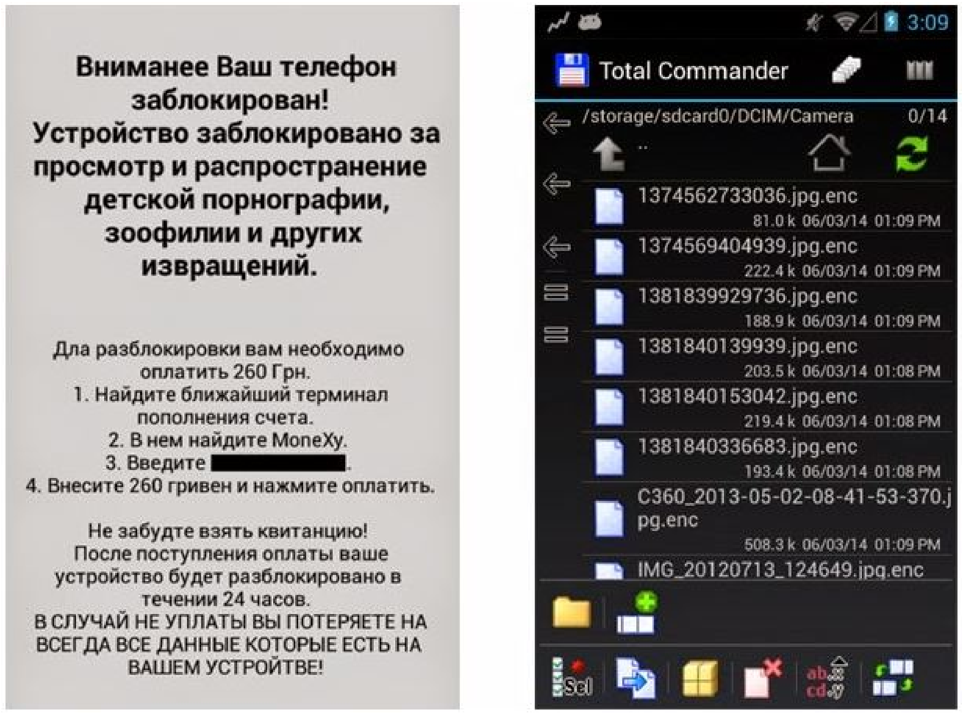
in underground markets every now and then with new features and functionalities. Dendroid is one of the variant of AndroRAT. Dendroid is a HTTP remote access tool with APK binder and a sophisticated PHP panel through which an attacker can control the infected Android devices [41].

### 2.4.2 Present

The current focus of Android malware is not to exploit any vulnerability in the operating system but perform the malicious activities by using the useful functionalities offered by Android platform. Malware writers have moved their focus on developing malware using the extra and powerful functionalities offered by Android platform. This technique helps to easily evade any signature based malware detection by well-known antivirus software. Some of the recent malware identified in the wild in the last few weeks are mentioned below:

- **Android/Simplocker.A :** Few weeks ago, a file-encrypting ransomware which is named as *Simplocker* was discovered by ESET's security engineers [20]. The sample was discovered in the form of an Android application "Sex xionix". According to their findings, this application acts as a Trojan, it scans the SD card installed on the Android device for certain type of files such as doc, docx, jpg, jpeg, txt, avi and png. It then encrypts them using the AES encryption algorithm. Meanwhile, it displays a ransom message to decrypt all the files. Figure 7 shows an example for the ransom message displayed by the application once it is installed and finished encrypting photos in camera directory. The message is written in Russian and the ransom is 260 Ukrainian hryvnias (21 US dollars) which makes it safe to assume that the source and target of this Trojan are Russians. The last few lines translates to *"After payment your device will be unlocked within 24 hours. In case of no PAYMENT YOU WILL LOSE ALL DATA ON your device!"*. To keep the identity secret, the application asks the user to pay the ransom using Monexy [36]. Simplocker also sends device specific information to its C&C server which is hosted on a TOR domain to maintain anonymity.

  Similarly, a fake anti-virus (Android/FakeAV) application named *Android Defender* was discovered and analyzed last year by Sophos [40]. The application poses itself to be a genuine anti-virus. According to Sophos engineers, this anti-virus, in the initial scan, prompts the user about two viruses, one Trojan and one malware which it offers to get rid of, only if the user clicks on *Buy and eliminate threats* button. Symantec discovered more aggressive version of *Android Defender* which almost locks down the device [42]. The User might not have an option of uninstalling the application or performing a factory restore as the malware attempts to prevent other applications from launching. Since the application is not stable on all Android applications, few lucky users might get a chance to uninstall the malware as it crashes.


**Вниманее Ваш телефон заблокирован!**
**Устройство заблокировано за просмотр и распространение детской порнографии, зоофилии и других извращений.**

Дла разблокировки вам необходимо оплатить 260 Грн.
1. Найдите ближайший терминал пополнения счета.
2. В нем найдите MoneXy.
3. Введите [redacted].
4. Внесите 260 гривен и нажмите оплатить.

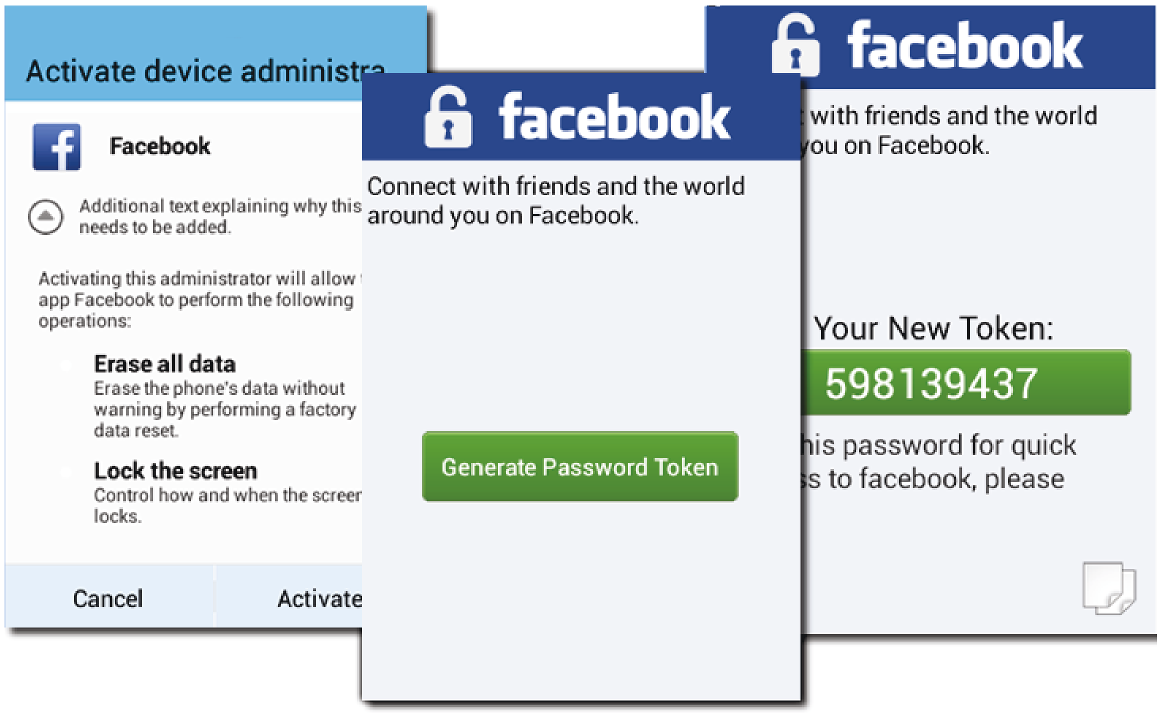
Не забудте взять квитанцию!
После поступления оплаты ваше устройство будет разблокировано в течении 24 часов.
В СЛУЧАЙ НЕ УПЛАТЫ ВЫ ПОТЕРЯЕТЕ НА ВСЕГДА ВСЕ ДАННЫЕ КОТОРЫЕ ЕСТЬ НА ВАШЕМ УСТРОЙТВЕ!

```
3:09
Total Commander
/storage/sdcard0/DCIM/Camera                0/14
..
1374562733036.jpg.enc
                        81.0 k 06/03/14 01:09 PM
1374569404939.jpg.enc
                       222.4 k 06/03/14 01:09 PM
1381839929736.jpg.enc
                       188.9 k 06/03/14 01:09 PM
1381840139939.jpg.enc
                       203.5 k 06/03/14 01:08 PM
1381840153042.jpg.enc
                       219.4 k 06/03/14 01:08 PM
1381840336683.jpg.enc
                       193.4 k 06/03/14 01:08 PM
C360_2013-05-02-08-41-53-370.jpg.enc
                       508.3 k 06/03/14 01:09 PM
IMG_20120713_124649.jpg.enc
```

Figure 7: Ransom message and encrypted files by *Android/Simplocker.A* [20]

- **iBanking :** It was initially introduced as an SMS stealer which is now turned into a sophisticated and powerful Android Trojan. According to Symantec [43], the source of *iBanking* is Russian cybercrime gangs and it is proved to be the most expensive malware sold in the underground markets/forums, priced at 5000 US dollars with updates and technical support. The strongest aspect of *iBanking* is that its application code is obfuscated. Social engineering techniques are used by the attackers to lure the user to install *iBanking* application.

  Consider a user infected with a Trojan on her PC. When the user visits a banking, social networking, e-mail account website, she is shown a message instructing her install a two-step verification token generator to keep the account secure from the ever increasing attacks. The user is prompted to enter her phone number and select the type of operating system she is using. If the operating system is Android, the user will be given a link to download the token generator client. She can also use a QR


Activate device administra

Facebook

Additional text explaining why this needs to be added.

Activating this administrator will allow app Facebook to perform the following operations:

**Erase all data**
Erase the phone's data without warning by performing a factory data reset.

**Lock the screen**
Control how and when the screer locks.

Connect with friends and the world around you on Facebook.

with friends and the world ou on Facebook.

Your New Token:
598139437

his password for quick s to facebook, please

Figure 8: An Example for Fake Facebook token generator used by *iBanking* [21]

code[2] to download the token generator client. The victim also receives a text message containing a link to *iBanking* application. The installed application is a complete imitation of a genuine banking, social networking or email account application which generates token for two-step verification. Figure 8 shows a fake Facebook token generator used by *iBanking*.

Once installed on a victim's device, *iBanking* can be controlled by the attacker either through HTTP or SMS control. The attacker can perform the following tasks using *iBanking* C&C server:

- Steal sensitive information
- Intercept, forward SMS and phone calls to the server
- Upload contacts, GPS location and recorded audio to the server

[2]QR code is also known as Quick Response code which is a trademark for matrix barcode. Smartphones can be used to scan QR codes which are mapped to a website URL

  - Redirect a call to the attacker's number
  - Prevent its removal or uninstallation
  - Restore device to factory defaults

- **Android/Samsapo.A :** *Android/Samsapo.A* was first discovered by ESET malware researcher and security professional [19]. This malware uses a technique similar to computer worms and tries to spread itself. It relies totally on social engineering where a victim installs the application thinking of it as a useful link. Hence, the installation process is totally dependent on the user's ignorance. If an application gets installed on an Android device, it sends an SMS message to all users in the contact list with a message that says *"Is this your photo?"* in Russian and a link which contains the malware package. The capabilities of *Android/Samsapo.A* can be summarized as follows:

  - It looks like a system utility with package name *com.android.tools.system v1.0*
  - No icon is generated after installation and the application has no GUI
  - Download malicious files
  - Upload sensitive information to server
  - Register for premium message service
  - Block phone calls and modify alarm settings

### 2.4.3 Future

The future of Android malware is vast. Unlike past and similar to present, future malware may focus on the use of advanced functionalities offered by the Android operating system to obtain sensitive information and perform malicious activities. Hasan *et al.* [26] explore a *new generation of Android malware* that exploits the various services available to Android applications such as the wide variety of sensors. The idea of exploiting the wide variety of sensors is to generate a malware that can target a very large amount of victims.

Malware in the past used to depend upon the traditional command and control servers which can be operated through a centralized server or by sending a text message. However, these new generation of malware would not need any centralized server to take control of the infected Android devices, it may make use of out-of-band channels. These communication channels allow the attacker to reach out to a very large number of users at once without raising an alarm in the user's mind.

This technique offers a very high level of anonymity and keeps it almost impossible for any malware analyzer to detect any malicious intent. The idea is to make use of sensor enabled channels instead of network-based communications which makes it very hard to identify the malicious activity by monitoring cellular or wireless communication data. In [26], Hasan *et al.* presented C&C servers based on visual, magnetic, audio and vibrational signals. For example, a malware can monitor the audio sensor and trigger the malicious payload when a specific song, announcement or message is played on radio or television. This technique can trigger billions of devices to perform a specific action at the same time. Examples of such malicious actions include performing factory restore, DDoS attack on a website, playing same song on billions of devices, send premium messages or make phone calls. Think of a situation where all the Android devices in the airport after listening to a particular announcement to perform a DDoS attack on a carefully chosen airport server causing all flight-ground communication to fail.

A malware with such capabilities can easily cause a worldwide disaster. Using this technique, a botnet can be controlled without using traditional networks and will be extremely hard for any malware analyzer or antivirus to detect it. This kind of malware can be used as a weapon against many facilities around the world. An intrusion detection system can be installed to monitor sensors and identify any malicious activity. However, a sophisticated malware can use a virtualization layer between the sensor and the application to avoid detection. There is a serious need to think about eradicating these kind of attacks before they even come into existence.

## 2.5 Efforts Made to Improve Android Security

Due to sudden increase in the number of Android malware, researchers too have moved their focus and resources towards securing the Android platform from the rising threats. Most of the security researchers and professionals have realized that Android is a very powerful platform and all its power is being controlled by its user. If the user is smart enough to properly use these powers, then there will be no need to take any actions against the rising threats. However, it is almost impossible for every user to totally understand the Android platform. Most of the users are even unaware of the possible actions they may allow an application to perform by allowing a specific permission.

Due to the gaps between Android security and user's understandings, malware professionals and researchers have developed ways to save users from the possible malicious actions performed by any application with malicious intents. We have divided the work related to our thesis into three different parts as mentioned below:

- Assessing current Android security
- Extensions proposed to improve Android security
- Tools developed to analyze Android applications

### 2.5.1 Assessing current Android security

In [51], Yajin and Xuxian collected over 1200 malware samples from 49 different families to systematically characterize the existing Android malicious applications. According to

Yajin and Xuxian, more than 14 different families of Android malware were discovered on Google Play store. Furthermore, according to their characterization results, around 86% of Android malware are repackaged version of most popular applications, around 37% of Android malware focus on exploiting platform level vulnerabilities, about 45% of the malware tries to subscribe to premium services by sending out SMSs without the user's knowledge and 93% of them showed bot like capabilities. Then, Yajin and Xuxian chose four leading antivirus software for Android devices and installed them on different smartphones. After this, they executed all the samples, one by one, on those smartphones and monitored the response from the anti-virus software. They received very astonishing results where only one of the anti-virus was able to reach an accuracy of about 79% whereas the worst detection rate was as low as 20%. According to them, current anti-malware solutions are not good enough to diligently identify the presence of even known malicious applications and there is a need to develop better anti-malware solutions to fight the outgrowing number of malicious applications.

### 2.5.2 Extensions proposed to improve Android security

Enck *et al.* [18] proposed an extension to current Android platform named *TaintDroid*. *TaintDroid* is designed to provide the user with adequate amount of visibility and control over third party applications. *TaintDroid* performs a real-time dynamic taint tracking to detect if any third party application tries to access sensitive information. To perform dynamic taint tracking, *TaintDroid* defines labels or taint for data that is privacy-sensitive which includes user data, device specific information or data stored on the device. Enck *et al.* claimed that *TaintDroid* can track multiple labels at the same time and record the information to generate logs. The key goal of *TaintDroid* is to achieve very high efficiency due to limited amount of resources in smartphones. *TaintDroid* achieved high efficiency by combining four different levels of granularity together, namely, file-level, variable-level, message-level and method-level. *TaintDroid* was tested with 30 samples collected randomly from distinct repositories and the results came out promising. It detected that 20 out of 30 applications misused the user's private information in 68 instances. Moreover, 15 out of 30 applications sent the user's current location to the remote servers.

Russello *et al.* [39] presented an Android extension named *YAASE* (yet another Android security extension). *YAASE* provides a policy enforcing mechanism and incorporates tainting mechanism from *TaintDroid* [18] for labelling the sensitive data and monitoring how the data is spread within the operating system and leaves the operating system through the Internet connection. Due to the policy enforcing mechanism introduced in *YAASE*, applications are limited to access specific labels only which provides an opportunity to the user to keep the sensitive data far away from the user. It also allows the users to assign network capabilities to an application and limit them to certain links only resulting into controlled Internet access; hence, keeping the device safe from any data being uploaded to the remote servers. *YAASE* tags the data using user-defined labels such as public, private and confidential; hence, providing access to the public tags to all the applications while keeping private and confidential data untouched. Russello *et al.* claimed that the results show possible overheads. However, it can be further optimized to reduce the overheads by limiting the focus on monitoring data leaving the operating system through Internet connections.

Nauman *et al.* [37] presented a framework named Apex that allows policy enforcement mechanism for Android platform. Their biggest concern is why users need to allow all the permissions to an application to use it and not grant only specific permissions. They also constraint the use of resources by any specific application even after it is installed on the device. Nauman *et al.* aimed at allowing users to control the resources accessible by any third party application. *Apex* is an extension for current Android operating systems which can be practiced by modifying very small part of the code. Their idea was to add this functionality while installing the application. Users should be able to select which permissions to allow and which permissions to deny before an application is installed onto the device. Also, if an application asks for *SEND_SMS* permissions, it should specify how many text messages an application is planning to send out every day, weekly or monthly so that users can use *Apex* GUI to constraint that application to that many number of short messages. Similarly, other resources can be constrained to keep the device secure from any unknown data leakage by applications with malicious intentions.

Tang *et al.* [45] proposed an extension to the current Android operating system using a security distance model named as *ASESD* (extension of Android Security Enforcement using Security Distance model). According to Tang *et al.*, a permission alone cannot do much to compromise sensitive data on an Android device but certain combinations of permissions can be a huge threat to the users. Also, some combinations are extremely malicious whereas others are used to provide normal functionality to the users. For example, the combination of *READ_PHONE_STATE* and *RECEIVE_SMS* or *INTERNET* and *RECEIVE_SMS* permissions poses no threat to the user. However, a combination of *READ_PHONE_STATE* and *INTERNET* or *RECEIVE_SMS* and *SEND_SMS* can result into a huge amount of data leakage. *ASESD* calculates the security distance based upon the severity of the permission combinations used by an application. Security distance is defined as safe, normal, dangerous and severe dangerous, based upon the security distance 0, 1, 5 and 25, respectively. Security distance helps in determining the threat level of an application. Tang *et al.* tested their proposed system on 100 Android applications where about 80 applications lied between security distance or threat level of 1-20, less than 20 applications lied between 21-50 and a few between 51-100.

### 2.5.3 Tools developed to analyze Android applications

Burguera *et al.* [16] proposed a solution to analyze Android applications and differentiate between malicious and benign application using Crowdsourcing. Keeping in mind the limitations of resources and the limited battery life of a smartphone, they came up with an idea of completely analyzing the applications on a dedicated remote server instead of the Android device. The authors also developed a client named *Crowdroid* which can be downloaded and installed from the Google Play store. The client, once installed on an Android device, monitors and records all the system calls (Linux Kernel) issued by any application installed on that device. *Crowdroid* uses a well-known Linux system call tracing tool named *Strace*. Later, the monitored data is sent to the centralized remote server which parses the received data from different users. The server generates a set of system call feature vector that defines the application behavior. There are more than 250 Linux system calls and this server counts the number of times a particular system call is issued by an application in the data received from *Crowdroid*. The feature vector is nothing but the count of every system call in an application. According to the authors, the accuracy is directly preoperational to the number of users practicing *Crowdroid*. Finally, Burguera *et al.* performed partitioning clustering techniques to cluster the data into benign and malicious applications. The clustering algorithm they used was *k*-means because of its efficiency, speed and simplicity. Burguera *et al.* tested their system for both self-written and real malware specimens. They claimed that they obtained an accuracy of 100% for all the self-written malware. Burguera *et al.* also tested it with two real world malware specimens which resulted in 100% and 85% accuracy.

Grace *et al.* [24] built an Android zero-day malware detection tool named *RiskRanker*. They wanted to overcome the limitation of detecting only the known families of malware most frequently used by anti-virus companies through signature based detection. Grace *et al.* claimed that *RiskRanker* aims towards analyzing the application's behavior to assess security risks posed by zero-day vulnerabilities as well as known vulnerabilities. *RiskRanker* analyzes applications collected from any source through two different analysis modules, namely, *first-order* and *second-order* module. *First-order* analysis module focuses on applications which do not use any obfuscation technique and then distinguishes that application into high or medium-risk application where application using platform level exploits (for e.g., RATC, Asroot and Exploid) are categorized as high-risk, and applications that might result into spending money without user's direct involvement are categorized as medium-risk. Whereas, *second-order* focuses more on applications that use obfuscation techniques and dynamic code loading. However, the applications cannot be assumed to be malicious only because of obfuscation and dynamic code loading. The authors claimed that, if an application use these techniques with any malicious intentions that might make an application really dangerous. Grace *et al.* collected more than 118K applications and performed analysis on all these applications using *RiskRanker* in less than four days; hence, achieved very high analysis speed. The authors were successful in identifying a large number of applications with zero-day vulnerabilities and known vulnerabilities (from known malware families). Grace *et al.* also claimed that *RiskRanker* is the only tool that identifies applications with zero-day vulnerabilities with minimal analysis time.

Blasing *et al.* [14] proposed a sandbox named *AASandbox* which can perform both static and dynamic analysis on Android applications and then detect suspicious applications. They suggested that it can be deployed onto a cloud server to perform analysis of Android applications in a much faster way. In static analysis, an application under analysis in disassembled and then searched for known malicious signatures or patterns. To perform dynamic analysis on Android applications, they used a LKM (Loadable Kernel Module) to load a custom kernel image onto the Android emulator which allows logging of low level interactions of Android applications with the device. Similar to *Crowdroid* [16], *AASandbox* also counts the number of times a particular system call is issued by an application. The information obtained during the static and dynamic analysis of Android applications is then logged and can be used for further investigations.

In [27], Isohara *et al.* proposed a kernel based behavior analysis system for Android applications. Similar to *Crowdroid* [16], they have also used *Strace*. *Strace* records all the system calls issued by a process and any signal received by a process. Kernel based behavior analysis for Android malware detection consists of two main modules, namely, log collector and log analyzer. Log collector logs all the kernel level logs using *Strace* whereas log analyzer parses the collected logs. Log collector collects the logged system calls issued be all the applications on a device and sends it to a remote server for further analysis by the log analyzer. Log analyzer removes the noise from the logs received by filtering it to show the logs regarding the application in question and then further filters it for the selected number of system calls. Then, to identify any information leakage, it searches for signatures such as IMEI, IMSI and Android ID. For identifying any platform level exploitations, it searches for signatures like asroot, exploid and gingerbreak. It also looks for any application that uses "su" command in its activities. The authors performed their analysis over 230 samples and claimed that their solution detected 37 leakage applications, 14 platform exploits and 13 applications using "su".

Zhou *et al.* [52] presented a system named *DroidRanger* which detected the malicious applications amongst the samples collected from five app stores including Google Play store. The collected applications were stored in a repository which feeds the applications to *DroidRanger* one by one for analysis. They claimed that their tool can detect new malware applications which belong to known malware families and malicious applications containing zero-day vulnerabilities. To detect malware from known malware families, it performs a *permission based behavioral footprinting*, looks for API call sequence, searches for possible SMS interception or monitoring (abortBroadcast method) and matches the structural, hierarchical layout of an application. To detect unknown malware, *DroidRanger* performs *heuristics-based filtering* and *dynamic execution monitoring*. The authors analyzed more than 200K Android applications from well-known market places including the Google Play store. *DroidRanger* detected 211 malicious applications which include 32 from Google Play store and also uncovered two zero-day vulnerabilities in 40 applications which include one application from Google Play store.

Wu et al. [48] proposed an application analysis technique purely based upon static features named as *DroidMat*. This techniques use the permissions, API calls, intent message passing and deployment of components to identify the behavior of the application. The first step in the analysis of an Android application is to extract the useful information from Android manifest file such as permissions used and identify different components used by the application. Based on the information collected from the manifest file, it drills down the components used in an application such as activity, service and receiver to identify API related to the permissions used. *DroidMat* applies $k$-means clustering with the number of clusters being decided by the *Singular Value Decomposition* method. Finally, to classify the application into benign and malicious categories, it applies $k$-nearest neighbor algorithm. The testing results of *DroidMat* were promising. Wu *et al.* claimed that even without using any dynamic analysis, *DroidMat* offers a high accuracy of about 97%. Moreover, the absence of the dynamic analysis makes the analysis process extremely fast.

Alazab *et al.* [4] investigates the behavior of Android applications by using a well-known dynamic analysis tool named *DroidBox*. In this thesis, we have also utilized *DroidBox* to perform dynamic analysis of Android applications. Alazab *et al.* used *DroidBox* to obtain behavioral and treemap graphs for each application under analysis and generate patterns to identify and classify the applications into different families. They collected malware samples from different families and then compared behavioral and treemap graphs of the applications to classify similar graphs into same families. Alazab *et al.* claimed that the results achieved by their analysis have proved that behavioral and treemap graphs generated by *DroidBox* can be used to accurately classify malware samples into different families. The authors also claim that the comparison between their system and three well-known antivirus vendors proved the antivirus vendors wrong. Moreover, they also discovered that some of the applications which were tagged as benign by these vendors turned out to be leaking data.

# Chapter 3

# AndroSAT

## 3.1 Introduction

In this chapter, 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. This can be achieved 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 and logs API level interactions with the operating system.

*AndroSAT* comprises a local web-server where we can upload the Android applications to our Droid-Repository (MySQL database of Android applications to be analyzed) through a PHP webpage (DroidShelf). As depicted in Figure 9, *AndroSAT* includes three main modules, namely:

- Droid-Shelf
- Droid-Repository

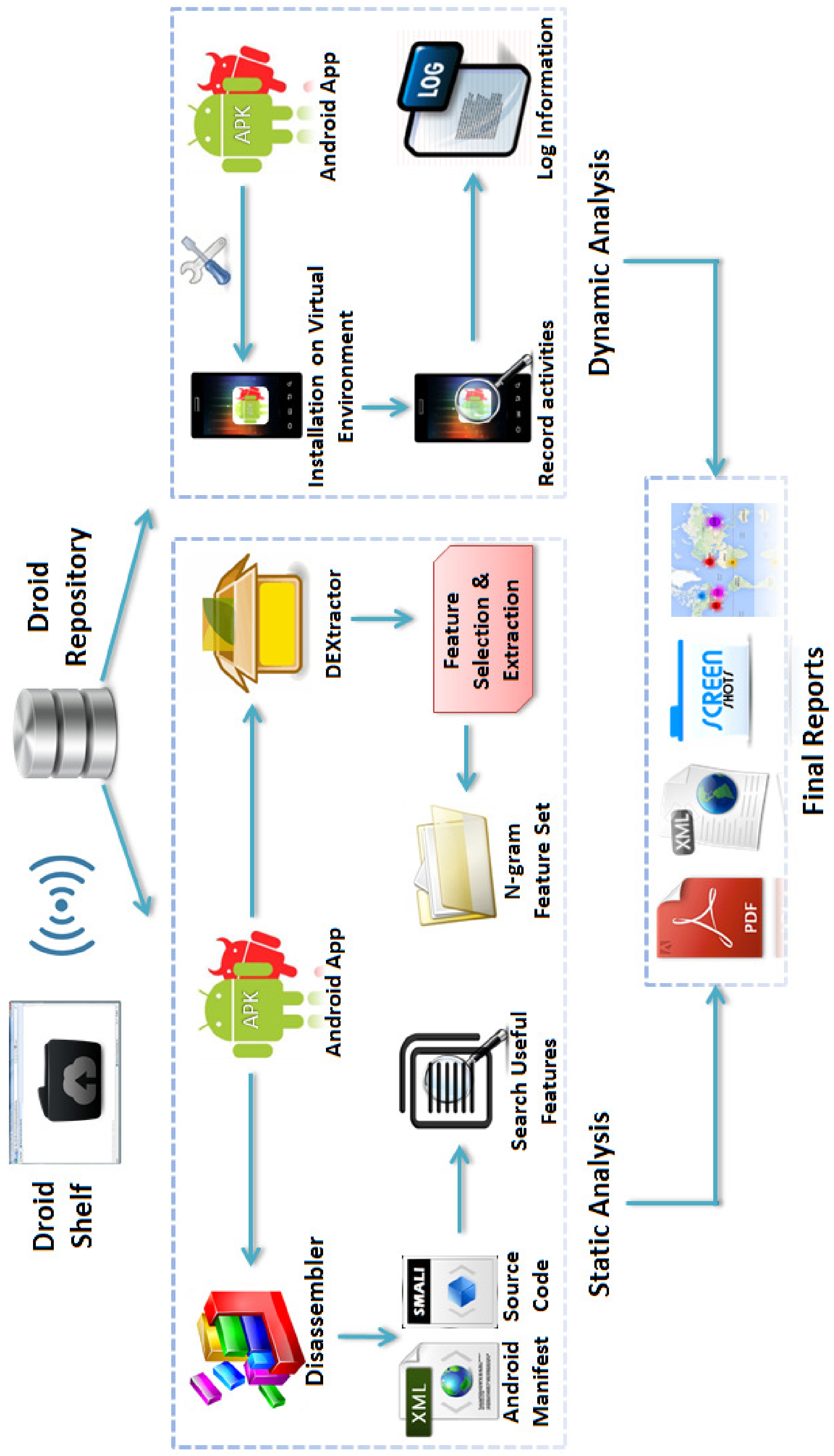


Figure 9: Overview of *AndroSAT*

- Analyzer

*Analyzer* further includes two modules, 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 be processed using several add-ons and analysis tools.

Moreover, *AndroSAT* uses the Android emulator to perform the dynamic analysis. As a matter of fact, all security professionals and researchers prefer to use a virtual machine as their testing and analyzing environment. They do that for plenty of reasons including:

- A malware with malicious intents might corrupt the operating system and make the device inoperative forever. On the other hand, using a virtual machine as the analysis environment helps to keep the physical device intact.
- It is easier to automatically install new applications in the virtual machine.
- Generating a fresh copy of the operating system for analysis of each application is only feasible in a virtual machine. Doing this ensures that no residue is left behind by the previous analysis process.
- Using a virtual machine allows random or specific clicks over the user interface of the application. This helps to unveil all the facets of the application under analysis.
- In a virtual machine, it is easier to record interactions between the operating system and an installed application.
- Virtual environments make it easier to switch between different versions of the operating system.
- It is easier to feed a virtual machine with mock user data every time a fresh copy is generated.

Because of the reasons mentioned above, most of the security professionals and researchers remain inclined towards using a virtual environment for the dynamic analysis of malicious applications. However, using a virtual environment also has its own drawbacks. All the malware writers are aware that dynamic analysis tools use virtual machines. Hence, these malware writers spend a lot of time in making their malicious applications vaccinated against virtual environments. Their aim is to write a malicious application which first verifies on which environment it is being run. Only after confirming that the application is not running in a virtual environment, it reveals and executes its malicious payload. Doing so allows malware writers to keep the malicious code from being revealed to automated security analysis tools. These analysis tools might make the malware useless if the information extracted from the analysis is used to generate signatures for ant-virus and IDS tools.

Malware writers these days tend to use anti-sandbox code in their applications which makes it very hard for a virtual environment to reveal the malicious payload. Anti-sandbox codes are made to diligently identify the presence of a virtual environment. Many malware writers sell these codes for thousands of dollars in underground forums and software markets.

Throughout our work, we paid a special attention on how to keep our emulator undetected and vaccinated towards anti-sandbox codes. There are many well-known techniques used by Android malware writers to identify if the application is running on an emulator or not. These techniques are based upon hardware and software specifications of an Android device, such as the IMEI number. For example, IMEI number of an emulator is 000000000000000 by default. If an application issues an API call to obtain the IMEI number of an Android device and the default string is returned, the application will not execute the malicious code on that device. We developed several techniques to modify many attributes of the sandbox. These techniques make it difficult for a malicious application to detect the presence of a virtual machine. Some of the fields we modified to make the virtual machine difficult to detect include:

- IMEI number
- IMSI number
- Android ID
- Fingerprint
- BootLoader
- Model and Manufacturer
- Service Provider
- CPUInfo and MEMInfo

We developed an automated tool which can modify these fields of an emulator to avoid detection. To modify these fields we applied three different techniques:

- **Replacing Build.prop file :** Build.prop is a part of system.img image file which is used to boot the emulator. In this technique, Build.prop file is extracted from system.img and then modified to reflect the properties of a genuine device such as model, brand, manufacturer, brand and board. After this, Build.prop file is repacked into system.img file.
- **Updating contents of Emulator's Executable :** This technique involves updating the contents of the emulator's executable. Some of the useful fields that can be updated through this technique are IMEI, IMSI, SIM serial number and service provider.

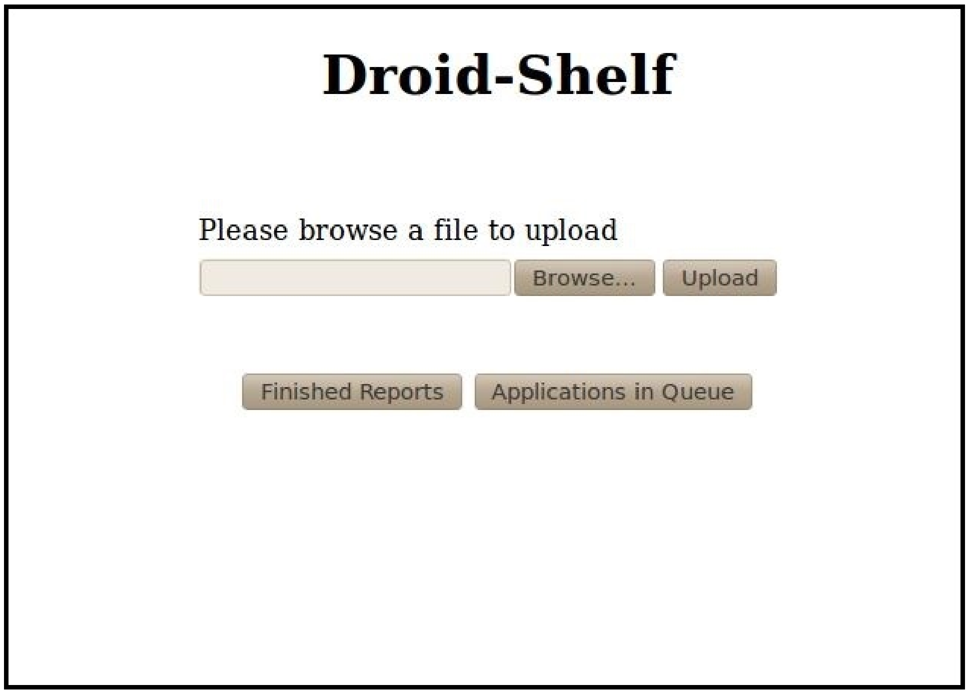


Figure 10: The main menu of *Droid-Shelf*

- **Loadable Kernel Module (LKM) :** This technique is used to modify the information about the installed CPU and memory in the emulator which is stored in the CPUInfo and the MEMInfo process, respectively. Loadable kernel module can be used to replace CPUInfo and MEMInfo processes with dummy processes containing modified information.

### 3.1.1 Droid-Shelf

*Droid-Shelf* is a web-based user interface where a user can easily upload the Android applications that need to be analyzed by *AndroSAT*. The interface is easy to use and understand as shown in Figure 10. The web interface gives an option to browse the Android application and then upload it to a database (*Droid-Repository*). It also allows the user to view the reports of analyzed applications. Moreover, the user also have an option to view the applications which are in queue and waiting for the analysis.

### 3.1.2 Droid-Repository

*Droid-Repository* is a MySQL database which stores all the Android applications that are uploaded to the database through Droid-Shelf. Only the unique Android applications are stored onto the *Droid-Repository*. To make sure that a duplicate copy of the same application is not uploaded, *Droid-Shelf* calculates the hash of the application to be uploaded and compares it with applications stored in the database. Only if no match is found, the application is considered unique and is uploaded to the *Droid-Repository* for further analysis.

Also, *Droid-Repository* keeps track of the applications which were analyzed in the past so as to avoid analyzing the same application again. A specific field in the database, related to the application is updated with the analysis status. It can be either “Waiting”, “Ongoing” or “Finished”. When an application is uploaded to the database, the value is set to “Waiting” by default. When the application is being analyzed, the value is updated to “Ongoing” and then to “Finished” when the analysis of that application ends. All the applications stored on *Droid-Repository* are downloaded by *AndroSAT* to a local directory to perform static and dynamic analysis on them. Once the analysis is finished, the application is moved to its corresponding “Finished Reports” directory.

### 3.1.3 Analyzer

The third, and the most important building block of *AndroSAT* is the *Analyzer*. *Analyzer* goes through the *Droid-Repository* and searches for the applications that are waiting to be analyzed by *AndroSAT*. Then, the application waiting for analysis is downloaded to the local directory. It then passes the application location to the static and dynamic analysis modules of *AndroSAT*. Once the analysis is finished, the application is moved by the analyzer to its corresponding “Finished Reports” directory along with the newly generated XML, PDF and screenshots. *Analyzer* is divided into two parts based upon the different analysis techniques:

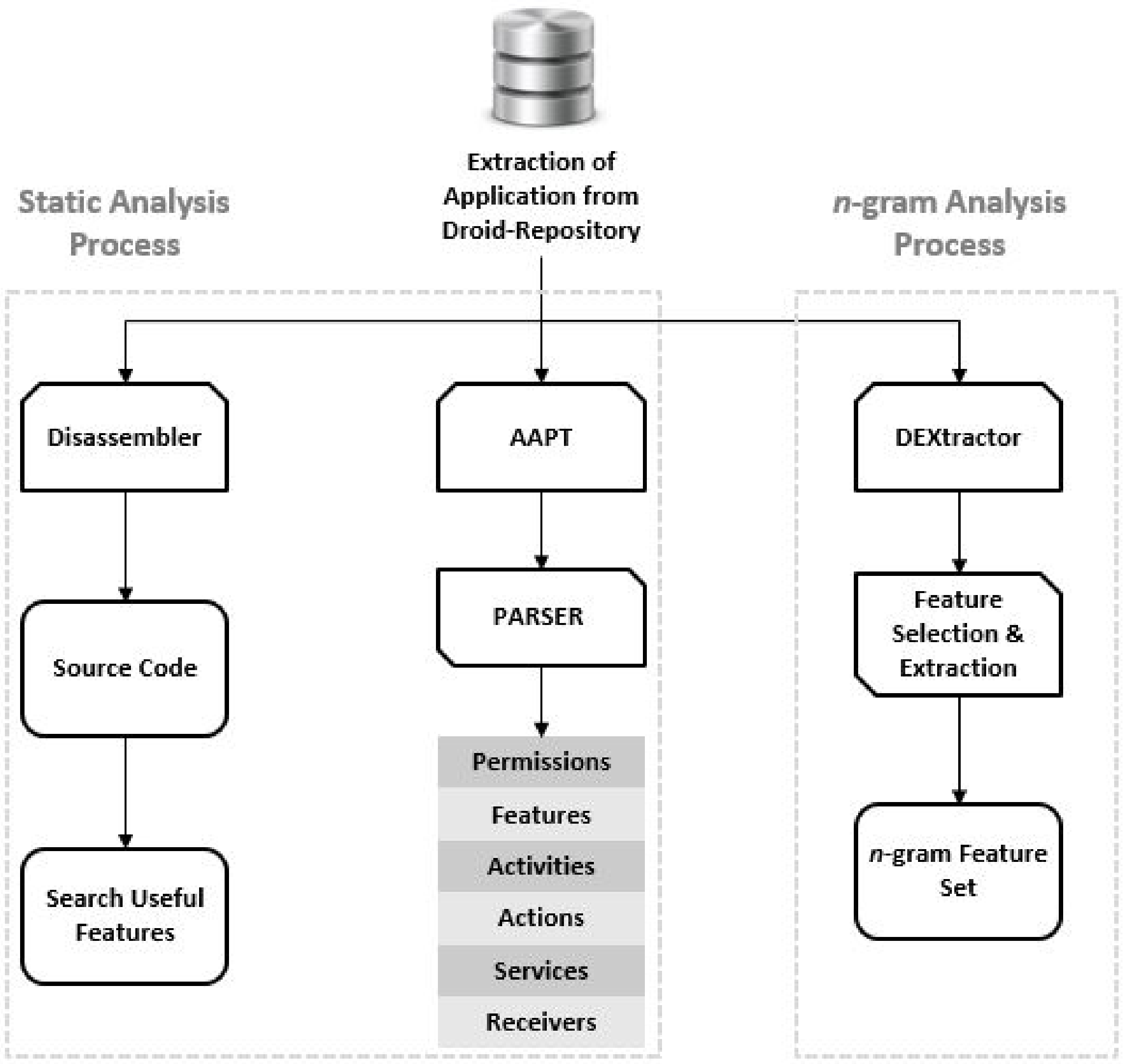


Figure 11: Overview of the static analysis module

- Static Analysis Module
- Dynamic Analysis Module

## 3.2 Static Analysis Module

Static analysis techniques aim to analyze Android applications without executing them. The objective of these techniques is to understand the intention of an application and predict what kind of operations and functionalities it might perform. Independent of the operating system, static analysis techniques have proved to be very fruitful.

Static analysis techniques are used by most well-known analyzers to obtain useful data present inside the APK package. This includes useful information such as permissions, services, activities and receivers which are stored inside the AndroidManifest file and API calls that can be obtained by disassembling the APK package. Static analysis proved to be quite effective and efficient as the analysis requires very little time as compared to dynamic analysis techniques.

As shown in Figures 9 and 11, 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 permissions, activities, intents and actions, services, and receivers) from the AndroidManifest file.

In order to perform the static analysis, the analyzed application is first fetched from the Droid-Repository. Then, useful information from the *AndroidManifest* file is extracted. To do so, the contents of the *AndroidManifest* file are obtained 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 development and is included in the Android SDK package [7]. After this, the application is fed to the disassembler which disassembles the APK package to obtain the SMALI and Java code of the application. The disassembly process is performed using APKTool [1] and Apk2java [2], which are open source reverse engineering tools, to obtain SMALI and Java code, respectively. Once the source code is obtained from the Android application undergoing analysis, we search for malicious functions/API calls and URLs hardcoded inside the application. SMALI is used to search URLs embedded in the application whereas Apk2java is used to search dangerous API or function calls. After that, the application is fed to the *DEXtractor* that extracts the *classes.dex* file from the package. *Classes.dex* file is then fed to the feature selection and extraction module which generates an *n-gram* feature set. In what follows we provide some further details on the n-gram analysis process, static analysis process and the different features extracted from both the AndroidManifest and source code during the static analysis of an application.

### 3.2.1 Static Analysis Process

The static analysis process goes down three ways: the first one provides the information extracted from the AndroidManifest file, the second searches for useful features in the source code of the Android application. The third part performs the *n*-gram analysis. The steps of the static analysis process can be summarized as follows:

1. Extract application from *Droid-Repository*
2. Feed the extracted application to the Android Asset Packaging Tool (AAPT) to obtain the information stored in AndroidManifest file
3. The information extracted in previous step is fed to the parser, to obtain useful features such as permissions, services, activities and receivers
4. The application is fed to the open-source disassemblers APKTool and Apk2java to obtain SMALI and Java source code
5. The SMALI and Java source code are then searched to obtain hard-coded URLs and dangerous API/function calls, respectively
6. The application is fed to the DEXtractor which extracts the *classes.dex* file from the APK package
7. Useful *n*-gram features are selected and extracted from the *classes.dex* file to generate *n*-gram feature set

### 3.2.2 *N*-gram Aanlysis Process

Different forms of n-gram analysis have been previously used for malware detection in the Windows and Linux/Unix environments. Different from Portable Executable (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 the 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 (CDF) [28] 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 applications. Surprisingly, as will be shown in the next chapter, applying this simple analysis method to .dex files even without any preprocessing 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.

### 3.2.3 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 which 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 [9]. Permissions requested by an application implicitly inform users 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 features as preferences, i.e., they can still function even if the feature is not granted. Features come with a required attribute that helps developers to specify whether the application can work normally with or without using that 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 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 occurs. The main purpose of using a broadcast receiver is to receive a notification once an event occurs. For example, if there is a new incoming message, a broadcast about the new incoming message is sent out and applications which 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 receiving the incoming messages that are not intended to them and monitoring 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 which 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.

### 3.2.4 Feature Extraction from Source Code

In this section, we list the features extracted from the decompiled Java code.

- getLastKnownLocation: This function is used to get the last known location from a particular location provider. Getting this information does not require starting the location provider which makes it more dangerous. Even if this information is stale, it is always useful in some context 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.

- 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 which 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.

## 3.3 Dynamic Analysis Module

The main advantage of the static analysis described above is that it can be performed relatively very efficiently without the need to execute the applications and hence avoids any risk associated with executing malicious applications. On the other hand, some malware writers use different obfuscation and cryptographic techniques that make it very hard 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 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

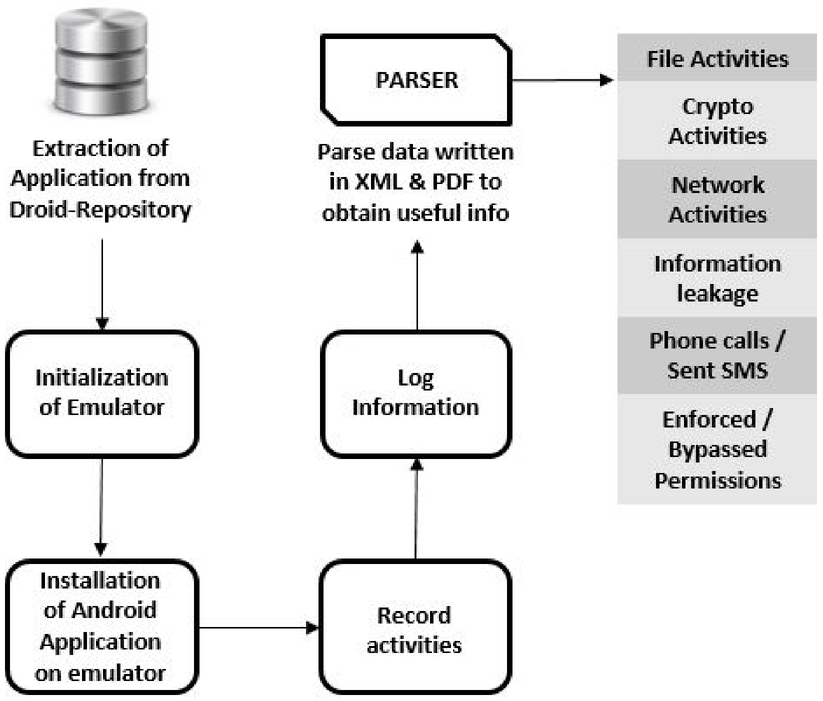

Extraction of Application from Droid-Repository
Initialization of Emulator
Installation of Android Application on emulator
Record activities
Log Information
PARSER
Parse data written in XML & PDF to obtain useful info
File Activities
Crypto Activities
Network Activities
Information leakage
Phone calls / Sent SMS
Enforced / Bypassed Permissions

Figure 12: Overview of the dynamic analysis module

analysis.

The main disadvantage of dynamic analysis, however, is that the usefulness of the analysis is somewhat correlated to the duration of the analysis interval. Some malicious activities may not be invoked by the app during the, usually short, analysis interval. This happens either because the conditions to trigger these events do not occur during the dynamic analysis process or because the malicious app can detect the presence of an analyzer. Anti-debugging and virtual machine detection techniques have long been used by Windows malware writers. To make the virtual environment looks like a genuine smartphone, we made some changes to the Android emulator. For example, we modified IMEI, IMSI, SIM serial number, product, brand and other information related to the phone hardware and software as explained in section 3.1.

During the dynamic analysis process, the application is installed onto the system and its activities and interactions are logged to analyze its actions. We use a sandbox to execute the Android application in question in a controlled virtual environment.

Figure 12 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 [31]. 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 to log the system and API level interactions of an Android application. Once the emulator is up, the application is installed onto the emulator for further analysis. Immediately after successful installation of the application, *AndroSAT* starts DroidBox to monitor the application at runtime for a configurable interval (the default is two minutes). Meanwhile, the MonkeyRunner tool [10] launches the main activity and performs random gestures on it, while DroidBox consistently logs any API level interactions of the application with the operating system. In what follows, we provide some further details on the dynamic analysis process and the features collected during this process.

### 3.3.1 Dynamic Analysis Process

Unlike the static analysis process, most parts of the dynamic analysis, as explained above, have to be performed in a sequential manner. However, whenever possible, we used multithreading to make the process more efficient. The steps of the dynamic analysis process can be summarized as follows:

1. Extract the target application from *Droid-Repository*
2. Initialize the emulator with *DroidBox's* custom image which makes monitoring API level calls feasible
3. Install the extracted application onto the emulator using *ADB* (Android Debug Bridge) [8]

4. Start *DroidBox* through command line to start monitoring and logging the interactions between the application and the operating system for the duration of the analysis interval

5. Launch main activity of the application installed in step 3 using MonkeyRunner [10]

6. Perform random gestures on the main activity launched in the previous step and take screenshots using MonkeyRunner

7. Logged interactions are fed to the parser to obtain useful information such as file activities, network activities, phone calls and sent SMSs

8. Extracted information is written to XML and PDF files for further analysis

### 3.3.2 Features collected during the 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 (e.g., key generation, encryption, and decryption). It also includes information regarding the type of operation performed by the application, algorithms used (e.g., AES, DES), key used and the data involved.

- Network Activities: This 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 very hard for any malware analyzer or sandbox to detect the malicious activities 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 any 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 [49].

- Information Leakage: Information leakage can occur through files, SMSs, and networks. 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, the timestamp, phone number and the contents of the text message are logged.
- Phone call: If an Android application tries to make a phone call, the timestamp and the dialed 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.

## 3.4 Using AndroSAT

Figure 13 shows the Graphical User Interface of *AndroSAT* running on Ubuntu. This tool provides security professionals with many handy options. Before starting the analysis of an application, it is necessary for the user to select one of the following options as the source for the Android applications that needs to be analyzed:

- **Analyze Applications from a Specific Directory :** If this radio button is selected while hitting the *Start Analysis* button, a browse window pops-up. Then, the user is asked to browse and open the first application from the directory containing the applications that needs to be analyzed.
- **Analyze Applications from Droid-Repository :** Unlike previous button, if this button is selected while hitting the *Start Analysis* button, the analysis will automatically fetch the applications from the pre-configured database *Droid-Repository*. *AndroSAT*

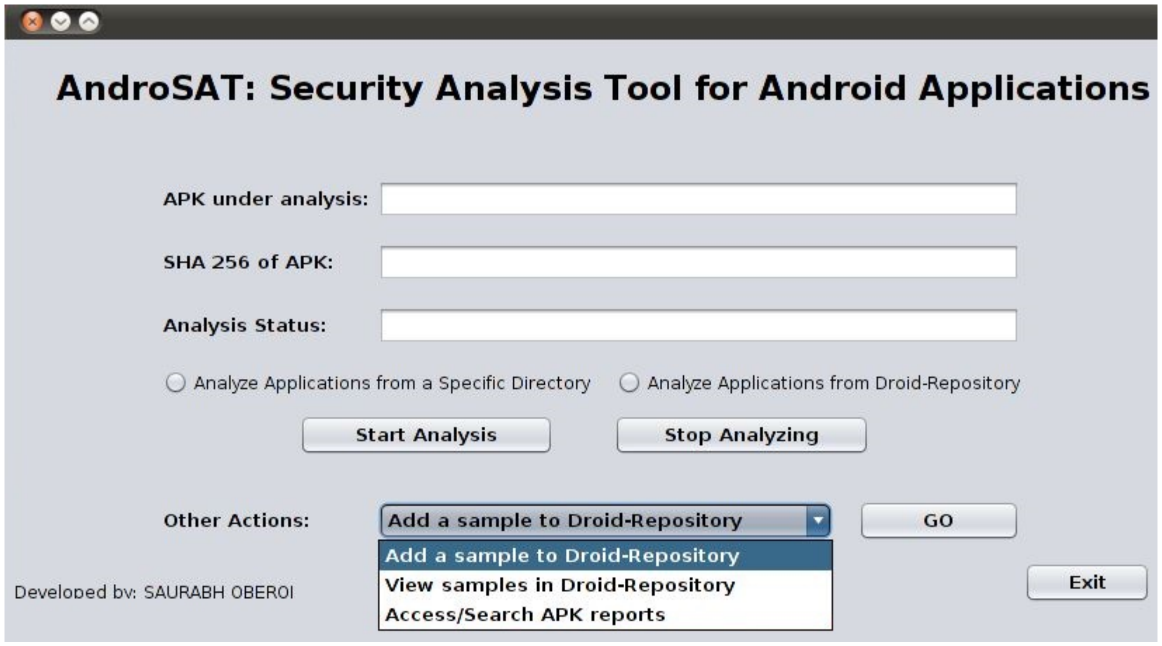

AndroSAT: Security Analysis Tool for Android Applications
APK under analysis:
SHA 256 of APK:
Analysis Status:
Analyze Applications from a Specific Directory
Analyze Applications from Droid-Repository
Start Analysis
Stop Analyzing
Other Actions:
Add a sample to Droid-Repository
GO
Add a sample to Droid-Repository
View samples in Droid-Repository
Access/Search APK reports
Developed by: SAURABH OBEROI
Exit

Figure 13: Graphical User Interface of *AndroSAT*

goes through the *Droid-Repository* and searches for the applications that are "Waiting" to be analyzed.

After the *Start Analysis* button is pressed, *AndroSAT* analyzes all the applications present in that specific directory or *Droid-Repository* unless the user interrupts the analysis by pressing the *Stop Analysis* button. In case of *Analyze Applications from a Specific Directory*, the applications analyzed up to that point will be moved to the directory containing all the reports of analyzed apps. Later on, if the user starts the analysis from the same directory, she does not need to analyze the same applications again. Similarly, in the case of *Analyze Applications from Droid-Repository*, the analysis status field is updated to "Finished" whenever an application is completely analyzed. If the user decides to stop the analysis process and start it later, *AndroSAT* starts the analysis of applications with "Waiting" analysis status. Hence, avoiding analysis of same applications again. However, if the user does not select any of the radio buttons, the analysis does not start and user gets an alert to select one of the radio buttons.

When the analysis process starts, “APK under analysis” in the GUI is updated with the name of the application under analysis. “SHA 256 of APK” is updated with the SHA256 hash of the application and “Analysis status” keeps the user informed about the current analysis phase. Moreover, while the applications are being analyzed, the user can perform the following “Other Actions”:

- **Add a sample to *Droid-Repository* :** If the user selects this option from the drop down list and presses the button “GO”, the user will be redirected to Droid-Shelf where she can upload new applications that needs to be analyzed.
- **View samples in *Droid-Repository* :** If the user selects this option from the drop down list and presses the button “GO”, the user will be redirected to a PHP webpage showing all the applications stored in the *Droid-Repository*.
- **Access/Search APK reports :** If the user selects this option from the drop down list and presses the button “GO”, the user will be redirected to the directory containing all the “Finished Reports”.

At any time in the analysis process, the user can press the “Exit” button to quit *AndroSAT*.

# Chapter 4

# Experimental Results

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 [51] and from another third party. The benign samples were obtained from F-Droid [22] 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 [3]. 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.

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 applications, 41 out of 962 malicious applications and 2 out of 970 benign applications used cryptography at runtime.

Generally, an Android application with different versions have different package contents and hence the checksum of the packages differ. However, according to our experimental results, the checksum of *classes.dex* file from different versions of an application came out to be the same. This tells us that some malware writers might add junk data in the APK package to make the different versions of an application look different. Meanwhile, the content of *classes.dex* file remains the same.

*AndroSAT* is not limited to case studies mentioned in this thesis. The information collected in the form of XML and PDFs can be used to perform further analysis. These case studies prove the wide usage possibilities of reports generated by *AndroSAT*. We incorporated the reports generated during the analysis process of applications and performed three case studies, namely, analyzing the frequency of use of different Android permissions and dynamic operations for both malicious and benign apps, producing cyber-intelligence information, and malware detection. These three case studies are explained in detail below.

## 4.1 Frequent Permissions and Dynamic Operations

Our first case study focuses on identifying the frequent permissions and dynamic operations to understand the distinguishing factors amongst malicious and benign Android applications.

Figure 14 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 14 READ_-PHONE_STATE permission is used by $\approx$86% of the malicious applications as compared to $\approx$12% of the benign applications. This permission is most widely used to obtain system sensitive data such as IMEI, IMSI, phone number and SIM serial number. Similarly, the frequency of use of INTERNET, ACCESS_WIFI_STATE, ACCESS_NETWORK_-STATE, READ_SMS and WRITE_SMS permissions shows noticeable differences. Figure 15 shows the top 15 permissions used by the analyzed benign applications and their frequency as compared to the analyzed malicious applications.

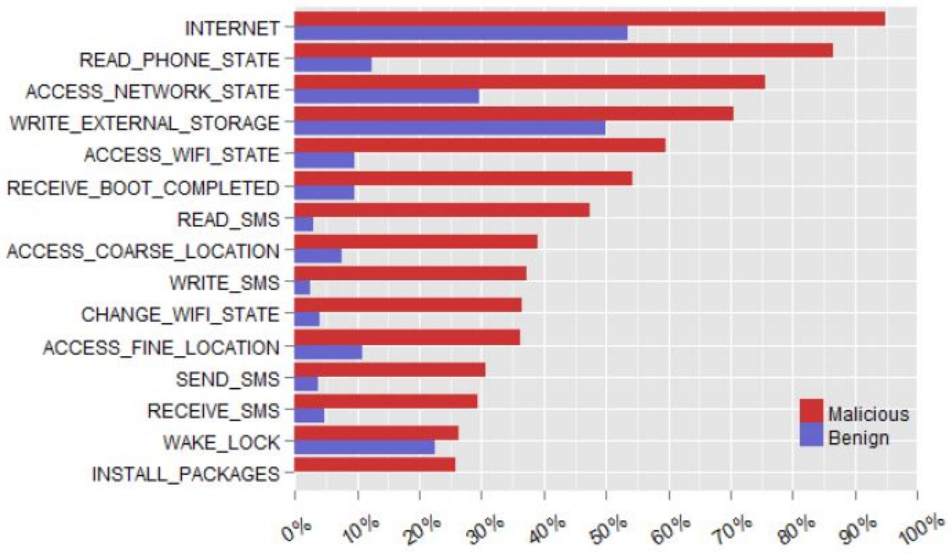


Figure 14: Top 15 permissions used by malicious applications

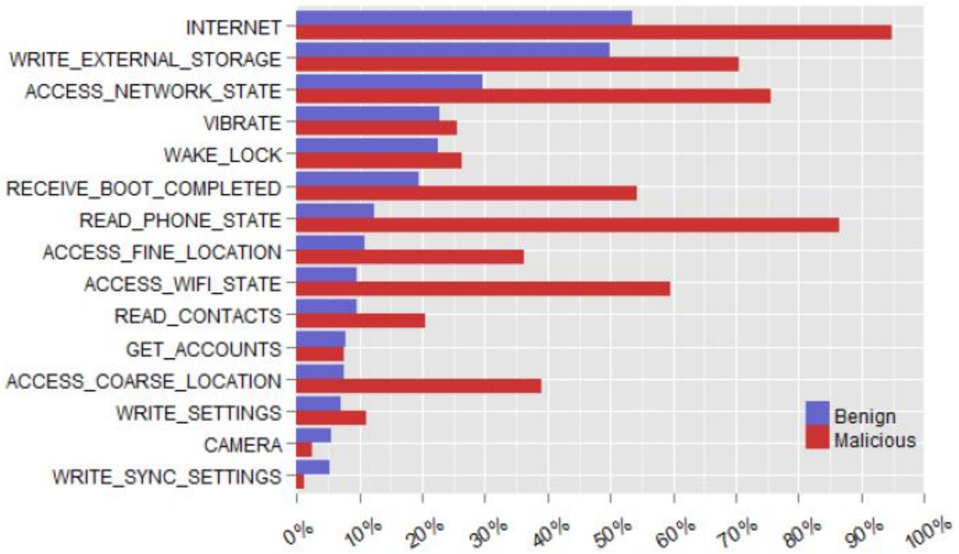


Figure 15: Top 15 permissions used by benign applications

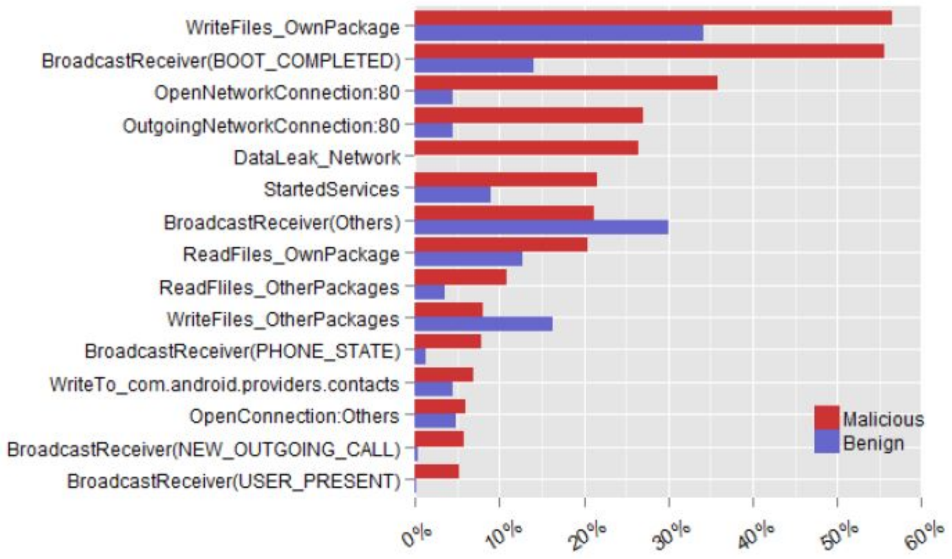


Figure 16: Top 15 dynamic operations performed by malicious applications

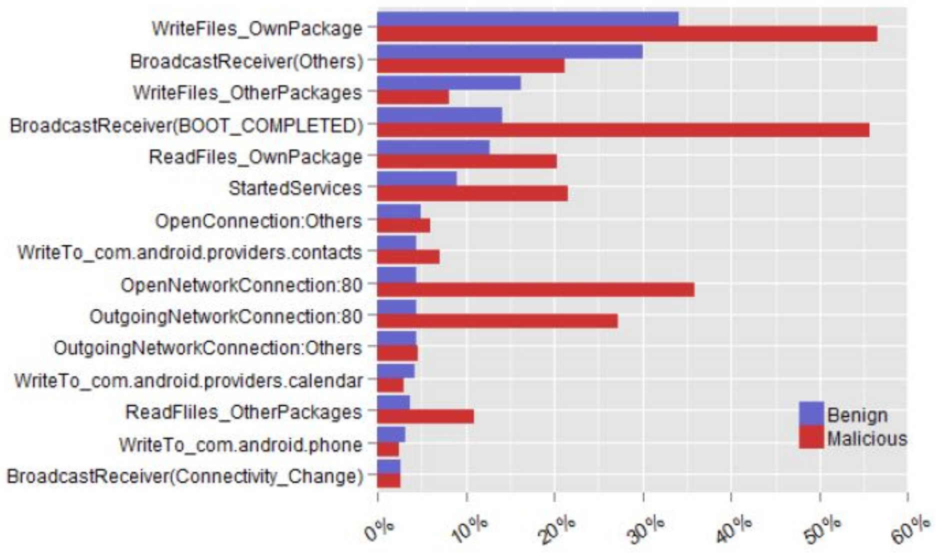


Figure 17: Top 15 dynamic operations performed by benign applications

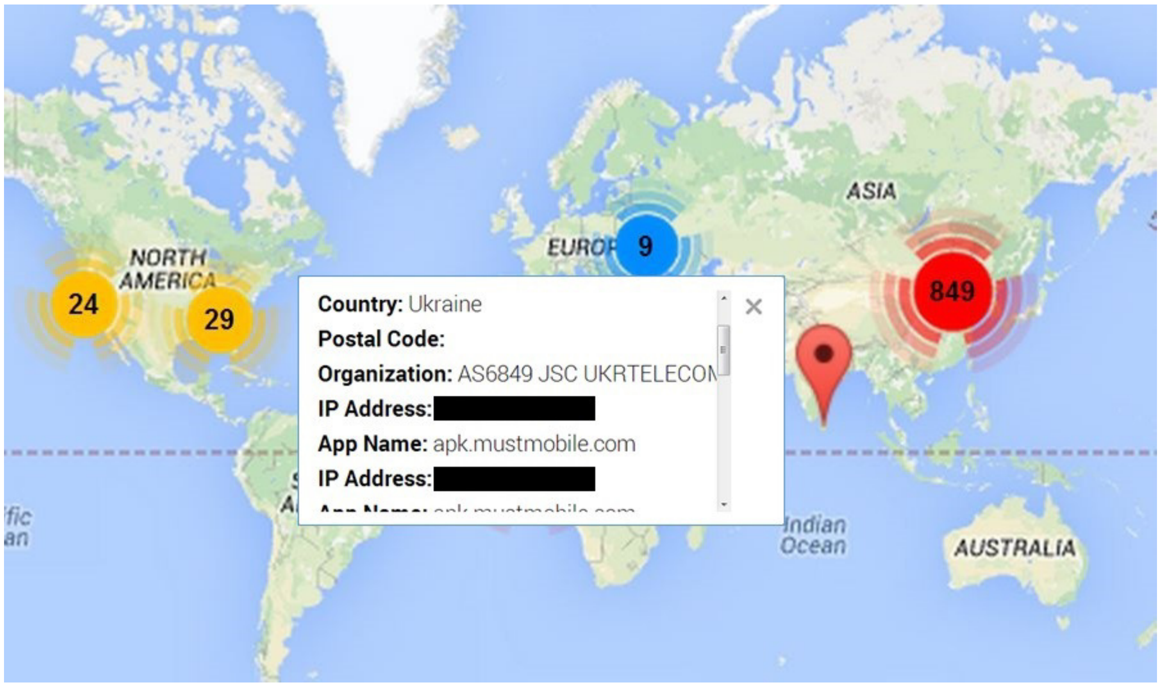
In total, 962 malicious applications used 10,203 permissions which comes to an average of 10.6 permissions per application. On the other hand, 970 benign applications used 3,838 permissions which comes to an average of around 3.95 permissions per application. These results confirm that, on average, the number of permissions requested by a benign application is less than the number of permissions requested by a malicious application.

Figure 16 shows the top 15 dynamic operations performed by the analyzed malicious applications. As shown in the figure, there are many operations that are frequently performed by the malicious applications. These include BroadcastReceiver(BOOT_COMPLETED), OpenNetworkConnection:80[1], and DataLeak_Network[2]. Applications with malicious intents use BOOT_COMPLETED broadcast receiver to receive a notification whenever an Android device boots up. This notification helps the malware to launch the intended malicious activity or service as soon as the system restarts. Data leakage through network is another frequently used dynamic operations amongst the malicious apps. It has a high occurrence in the malicious dataset, 254 as compared to 0 in the benign dataset. Figure 17 shows the top 15 dynamic operations performed by the benign applications.

## 4.2 Cyber-intelligence

Cyber-intelligence is performed at a large scale to identify cyber criminals and users with malicious intentions. In short, cyber-intelligence is tracking, analyzing and countering any threat related to digital media. One of the main objectives of cyber-intelligence is tracking sources of online threats. In our work, we utilized the URLs and IP addresses obtained during the static and dynamic analysis process of malicious applications to track possible malicious servers around the world. The URLs and IP addresses obtained during the static and dynamic analysis were either hard coded into the package or visited by the malicious

[1]This operation represents opened port 80 for network connections
[2]This operation represents sensitive data leakage through network



Country: Ukraine
Postal Code:
Organization: AS6849 JSC UKRTELECOM
IP Address: [redacted]
App Name: apk.mustmobile.com
IP Address: [redacted]

Figure 18: Geographical Presentation of the locations of suspected IPS

applications during the dynamic analysis process. We also converted the extracted URLs to their corresponding IP addresses. Then we mapped the collected IP addresses to their source location and obtained their corresponding latitude and longitude using *MaxMind's GeoIP2* database [34]. *GeoIP2* allows to identify the country, city, location and organization of the IP address. To map these latitude and longitude obtained from the database onto the map, we used Google Maps API which is free to use and can be easily adjusted to fit our requirements [23]. We mapped the IP addresses on to the Google Maps to generate a graphical representation of the geographical locations of possible malicious servers (and their ISPs). Figure 18 shows a sample output of this analysis (IP addresses are not shown for privacy and liability concerns).

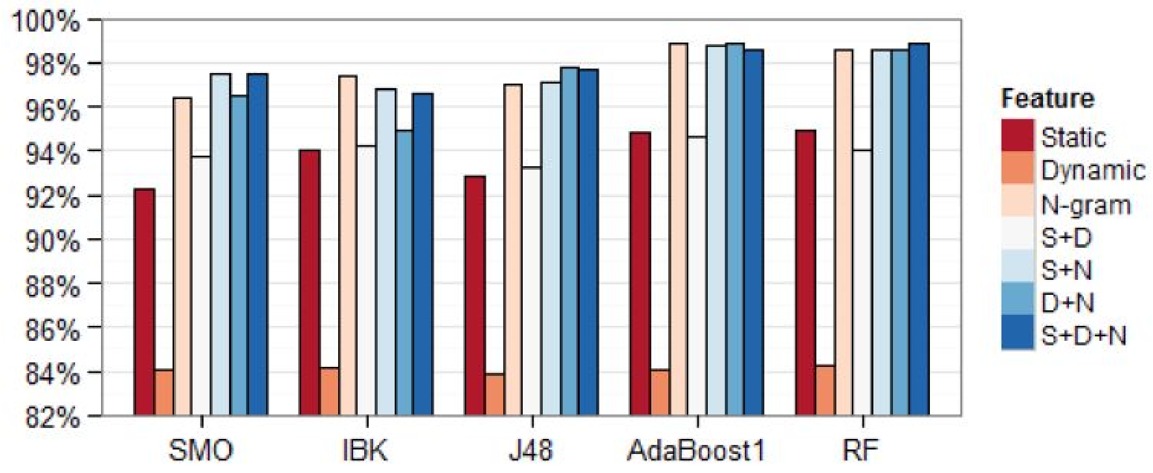


Figure 19: Accuracy obtained by different classification techniques

## 4.3 Malware Detection

The most widely used technique to separate benign and malicious applications from the mixed dataset is classification. We used classification as one of our case studies to distinguish malicious applications from benign and to confirm the accuracy of the *AndroSAT* framework. Throughout this experiment, we incorporated 134 static, 285 dynamic and 400 n-grams based features. We performed the classification over 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*. *CDF* is used to reduce the features and to obtain a feature-set containing the top features for classification.

We employed five different algorithms supported by WEKA [46] for classification with 10-fold cross-validation: SMO [25], IBK [25], J48 [25], AdaBoost1(J48 as base classifier) [25], and RandomForest [25]. In what follows we provide some brief overview of these

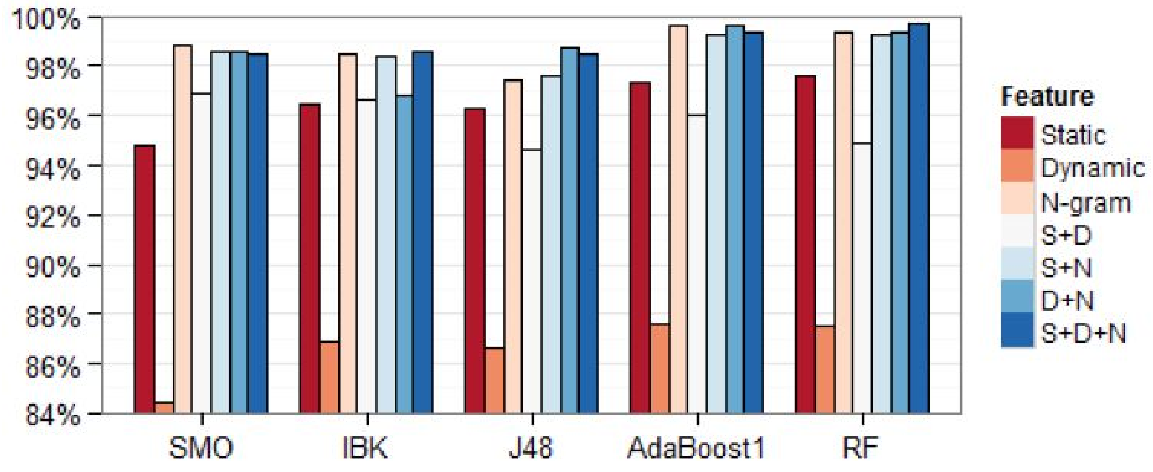


Figure 20: Precision obtained by different classification techniques

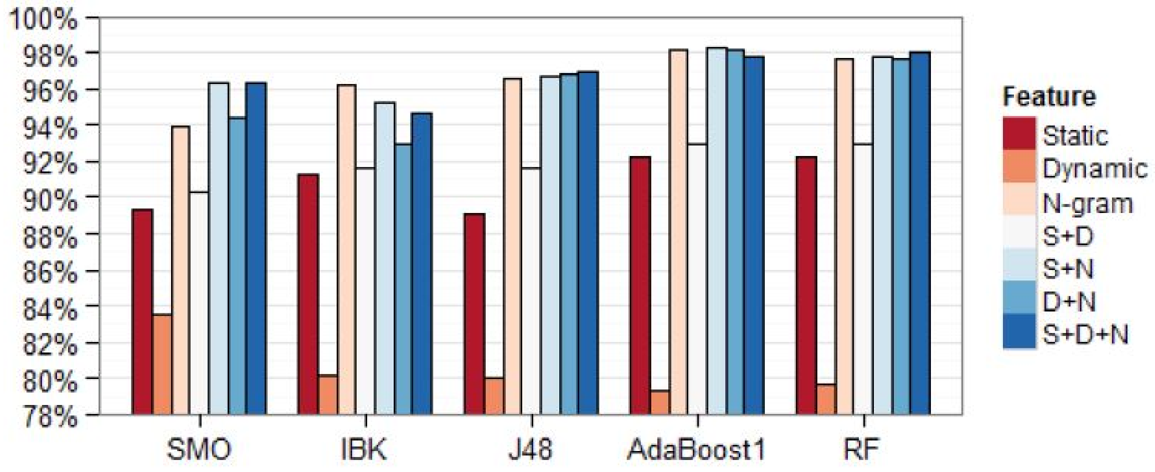


Figure 21: Recall obtained by different classification techniques

classifiers.

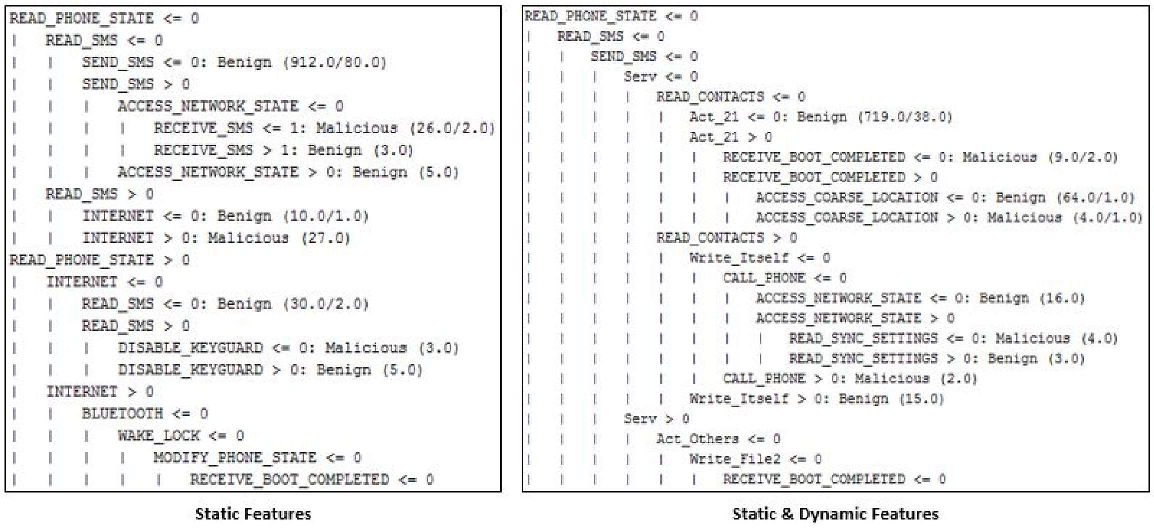
- **AdaBoost1 :** AdaBoost1 is known as a boosting algorithm which is widely used to boost the performance of any learning algorithm [25]. It is used in conjunction with the weak learning algorithm to significantly reduce the errors and noise. We have used this technique in this thesis with J48 as the base classifier and obtained promising results.

- **SMO :** SMO stands for sequential minimal optimization and is used for training SVMs (support vector machines) [25]. Sequential minimal optimization solves the quadratic problems for training SVM. The problem is divided into the smallest possible sub-problems and then solved analytically.

- **Decision Tree :** Decision tree is a tree or flowchart like structure which is most widely used in decision analysis to help identify the path to reach the goal [25]. A decision tree consists of root, leaves, internal nodes and branches. The trails from the root node to the leaf node are explained as rules that are used to perform the classification. The internal nodes represent attributes with some test, branches represent the result to the test and leaves represent the class labels.

- **IBK :** IBK is based on the *k*-nearest neighbors algorithm [25]. As the word neighbor suggests, the objects are classified into different classes on the bases of the majority of their neighbors. If an object is nearest to the neighboring objects with malicious as the class label, then it is also classified to the same class.

Our experimental results show that the AdaBoost1 and RandomForest models achieve better accuracy compared to the other models. Figures 19, 20 and 21 show the results obtained for the five different feature sets in terms of accuracy, precision, and recall. From Figure 19, it is clear that *n*-gram features using AdaBoost1, D+N features using AdaBoost1



```
READ_PHONE_STATE <= 0
|   READ_SMS <= 0
|   |   SEND_SMS <= 0: Benign (912.0/80.0)
|   |   SEND_SMS > 0
|   |   |   ACCESS_NETWORK_STATE <= 0
|   |   |   |   RECEIVE_SMS <= 1: Malicious (26.0/2.0)
|   |   |   |   RECEIVE_SMS > 1: Benign (3.0)
|   |   |   ACCESS_NETWORK_STATE > 0: Benign (5.0)
|   READ_SMS > 0
|   |   INTERNET <= 0: Benign (10.0/1.0)
|   |   INTERNET > 0: Malicious (27.0)
READ_PHONE_STATE > 0
|   INTERNET <= 0
|   |   READ_SMS <= 0: Benign (30.0/2.0)
|   |   READ_SMS > 0
|   |   |   DISABLE_KEYGUARD <= 0: Malicious (3.0)
|   |   |   DISABLE_KEYGUARD > 0: Benign (5.0)
|   INTERNET > 0
|   |   BLUETOOTH <= 0
|   |   |   WAKE_LOCK <= 0
|   |   |   |   MODIFY_PHONE_STATE <= 0
|   |   |   |   |   RECEIVE_BOOT_COMPLETED <= 0
```

```
READ_PHONE_STATE <= 0
|   READ_SMS <= 0
|   |   SEND_SMS <= 0
|   |   |   Serv <= 0
|   |   |   |   READ_CONTACTS <= 0
|   |   |   |   |   Act_21 <= 0: Benign (719.0/38.0)
|   |   |   |   |   Act_21 > 0
|   |   |   |   |   |   RECEIVE_BOOT_COMPLETED <= 0: Malicious (9.0/2.0)
|   |   |   |   |   |   RECEIVE_BOOT_COMPLETED > 0
|   |   |   |   |   |   |   ACCESS_COARSE_LOCATION <= 0: Benign (64.0/1.0)
|   |   |   |   |   |   |   ACCESS_COARSE_LOCATION > 0: Malicious (4.0/1.0)
|   |   |   |   READ_CONTACTS > 0
|   |   |   |   |   Write_Itself <= 0
|   |   |   |   |   |   CALL_PHONE <= 0
|   |   |   |   |   |   |   ACCESS_NETWORK_STATE <= 0: Benign (16.0)
|   |   |   |   |   |   |   ACCESS_NETWORK_STATE > 0
|   |   |   |   |   |   |   |   READ_SYNC_SETTINGS <= 0: Malicious (4.0)
|   |   |   |   |   |   |   |   READ_SYNC_SETTINGS > 0: Benign (3.0)
|   |   |   |   |   |   CALL_PHONE > 0: Malicious (2.0)
|   |   |   |   |   Write_Itself > 0: Benign (15.0)
|   |   |   Serv > 0
|   |   |   |   Act_Others <= 0
|   |   |   |   |   Write_File2 <= 0
|   |   |   |   |   |   RECEIVE_BOOT_COMPLETED <= 0
```

Figure 22: Partial Decision Trees obtained using static, and static & dynamic features

and S+D+N features using RandomForest provide the highest accuracy (≈98%). Figure 20 and 21 show the corresponding precision and recall, respectively. It should be noted that this relatively high accuracy should be interpreted with care since it might have resulted because of the limited variance in the characteristics of the analyzed samples.

Moreover, we used decision trees obtained while performing classification for only static features and combination of static and dynamic features. Decision trees are most widely used because of their transparent mechanism and it is easy to follow the structure of the tree to understand the classification rules and criteria. We used this technique to better understand the factors that play an important role in deciding the class label for a sample. Figure 22 shows the partial decision trees obtained while using only static and then combination of static and dynamic features.

From both decision trees, it is clear that READ_PHONE_STATE, READ_SMS and SEND_SMS permissions play a very important role in deciding whether the application under analysis is malicious or benign. The decision tree obtained from only static analysis features makes a lot of sense; for example, an application that reads SMS and uses Internet might leak privacy sensitive data to some attacker or malware writer. Similarly, an application that sends an SMS and has a capability to receive SMS might subscribe user to premium messaging services or extract useful information from received SMS like one time passwords. An application using RECEIVE_SMS permission has the capability to receive the incoming text messages and also block those messages from reaching the inbox; hence, resulting into data stealing.

# Chapter 5

# Conclusions and Future Work

## 5.1 Conclusion

The Android operating system powers more than a billion smart devices. Everyday, more than one million Android devices are activated with better hardware configuration as compared to the laptop and desktop computers used in the 90's. The sudden increase in the use of Android powered devices led to a huge amount of Android malware creation and dissemination.

In this thesis, we developed a prototype which can analyze Android applications and lay down its possible actions, functionalities and interactions with the Android emulator. It generates XML and PDF files containing useful information about an application's intentions. This information can be used to perform numerous security related case studies.

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 applications, producing cyber-intelligence information, and malware detection.

The implemented prototype is not limited to the analysis techniques presented in this thesis. It can be further extended to allow for more useful add-ons that may provide detailed investigation of malicious Android applications. Moreover, many different case studies can be performed with the data collected in the form of XML and PDF files to better understand intentions of the Android applications.

## 5.2 Future Work

There are few limitations with *AndroSAT* that needs to be addressed in our future research and development. In what follows, we summarize some of these weaknesses and limitations:

- The analysis of each application takes a relatively long time (about 5-6 minutes), which is not practical when analyzing thousands of applications. The default time to finish the dynamic analysis process is 2 minutes. Initializing a fresh copy of the emulator everytime an application is analyzed, takes about 2 minutes. In our future work, we would like to bring down the analysis time to 3-4 minutes. This will allow faster analysis of applications. To achieve this, we plan to discover techniques which can generate a fresh copy of the emulator without rebooting it everytime. Furthermore, we aim to bring down the time required to perform the static analysis of an application.

- Whenever an application is analyzed, the dynamic analysis process uses a fresh copy of the emulator, vaccinated against anti-sandbox code. However, there is no user data stored inside the emulator which makes it easier for the attackers to recognize the presence of a sandbox. Soon malware writers will introduce malware that does not execute their payload in the absence of user data. Our aim in our forthcoming work is to feed the emulator with some carefully crafted user data such as e-mails, contacts, call logs, text messages, third-party applications, pictures, videos and songs.

- *DroidBox* is a very effective and efficient open-source dynamic analysis tool available online. However, it does not track native API calls. Many malware writers include native code in their applications which helps to keep the malicious applications safe from commonly used security radars. According to our knowledge, native API calls are not enough to perform malicious activities. Nevertheless, the use of native API calls provides a backdoor to the malicious applications to hide some of its malicious intents from the outside world. In our future work, we plan to discover techniques which can track native API calls and detect applications which are trying to hide their malicious activities from security analysis tools.

# Bibliography

[1] Android-Apktool: A tool for reverse engineering Android APK files. [Online]. Available: https://code.google.com/p/android-apktool/.

[2] Apk2java: Batch file to automate APK decompilation process. [Online]. Available: https://code.google.com/p/apk2java/.

[3] VirusTotal. [Online]. Available: https://www.virustotal.com/en/about/.

[4] M. Alazab, V. Monsamy, L. Batten, P. Lantz, and R. Tian. Analysis of Malicious and Benign Android Applications. In *32nd International Conference on Distributed Computing Systems Workshops (ICDCSW), 2012*, pages 608–616, June 2012.

[5] Android. Android - Meet Android. [Online]. Available: http://www.android.com/meet-android/.

[6] Android. Android: the world's most popular mobile platform. [Online]. Available: http://developer.android.com/about/index.html.

[7] Android Developers. Android Asset Packaging Tool. [Online]. Available: http://developer.android.com/tools/building/index.html.

[8] Android Developers. Android Debug Bridge. [Online]. Available: http://developer.android.com/tools/help/adb.html.

[9] Android Developers. Android Permissions. [Online]. Available: http://developer.android.com/reference/android/Manifest.permission.html.

[10] Android Developers. Monkey Runner. [Online]. Available: http://developer.android.com/tools/help/ monkeyrunner_concepts.html.

[11] Android Devices. Android Security Overview. [Online]. Available: https://source.android.com/devices/tech/security/.

[12] Android Devices. Notes on the implementation of encryption in Android 3.0. [Online]. Available: https://source.android.com/devices/tech/encryption/android_-crypto_implementation.html.

[13] Android Devices. Validating Security-Enhanced Linux in Android. [Online]. Available: https://source.android.com/devices/tech/security/se-linux.html.

[14] T. Blasing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, and S. Albayrak. An Android Application Sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software (MALWARE), 2010*, pages 55–62, Oct 2010.

[15] S. Brahler. Analysis of the Android Architecture. *Karlsruhe institute for technology*, 2010.

[16] 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*, pages 15–26. ACM, 2011.

[17] S. Dredge. Kaspersky: forget lone hackers, mobile malware is serious business. [Online]. Available: http://www.theguardian.com/technology/2014/feb/26/kaspersky-android-malware-banking-trojans.

[18] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, volume 10, pages 1–6, 2010.

[19] ESET. Android malware worm catches unwary users. [Online]. Available: http://www.welivesecurity.com/2014/04/30/android-sms-malware-catches-unwary-users/.

[20] ESET. ESET Analyzes First Android File-Encrypting, TOR-enabled Ransomware. [Online]. Available: http://www.welivesecurity.com/2014/06/04/simplocker/.

[21] ESET. Facebook Webinject Leads to iBanking Mobile Bot. [Online]. Available: http://www.welivesecurity.com/2014/04/16/facebook-webinject-leads-to-ibanking-mobile-bot/.

[22] F-Droid. F-Droid: Free and Open Source Android App Repository. [Online]. Available: https://f-droid.org/.

[23] Google. Google Maps API. [Online]. Available: https://developers.google.com/maps/.

[24] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 281–294. ACM, 2012.

[25] M. A. Hall, E. Frank, and I. H. Witten. Data Mining: Practical Machine Learning Tools and Techniques. Burlington, MA: Morgan Kaufmann, 2013.

[26] R. Hasan, N. Saxena, T. Haleviz, S. Zawoad, and D. Rinehart. Sensing-enabled Channels for Hard-to-detect Command and Control of Mobile Devices. In *Proceedings of*

*the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 469–480. ACM, 2013.

[27] T. Isohara, K. Takemori, and A. Kubota. Kernel-based Behavior Analysis for Android Malware Detection. In *Seventh International Conference on Computational Intelligence and Security (CIS), 2011*, pages 1011–1015, Dec 2011.

[28] S. Jain and Y. Meena. Byte Level *n*-gram Analysis for Malware Detection. In *Computer Networks and Intelligent Computing*, volume 157 of *Communications in Computer and Information Science*, pages 51–59. Springer Berlin Heidelberg, 2011.

[29] Juniper Networks. Third Annual Mobile Threats Report. [Online]. Available: http://www.juniper.net/us/en/local/pdf/additional-resources/3rd-jnpr-mobile-threats-report-exec-summary.pdf.

[30] Juniper Research. Smartphone Shipments Reach a Record 250m in Q3 2013, with Nokia Back in the Top 3. [Online]. Available: http://www.juniperresearch.com/viewpressrelease.php?pr=408.

[31] P. Lantz. Droidbox: Android Application Sandbox. [Online]. Available: https://code.google.com/p/droidbox/.

[32] Q. Li and G. Clark. Mobile Security: A Look Ahead. *Security Privacy, IEEE*, 11(1):78–81, Jan 2013.

[33] A. Ludwig. Android: Practical Security from the Ground Up. [Online]. Available: http://goo.gl/7xZ4cd.

[34] MaxMind. GeoIP Web Services and Databases. [Online]. Available: http://dev.maxmind.com/geoip/geoip2/whats-new-in-geoip2/.

[35] C. Miller. Mobile Attacks and Defense. *Security Privacy, IEEE*, 9(4):68–70, July 2011.

[36] Monexy. Monexy: Defining the future of payments. [Online]. Available: http://monexy.biz/.

[37] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332. ACM, 2010.

[38] A. Orozco. Is Google Acknowledging Android Is Not Secure? hmm... [Online]. Available: http://blog.malwarebytes.org/mobile-2/2013/06/is-google-acknowledging-android-is-not-secure-hmm/.

[39] G. Russello, B. Crispo, E. Fernandes, and Y. Zhauniarovich. YAASE: Yet Another Android Security Extension. In *IEEE third international conference on Privacy, security, risk and trust (passat), 2011 and IEEE third international conference on social computing (socialcom), 2011*, pages 1033–1040, Oct 2011.

[40] SOPHOS. Android malware in pictures - a blow-by-blow account of mobile scareware. [Online]. Available: http://nakedsecurity.sophos.com/2013/05/31/android-malware-in-pictures-a-blow-by-blow-account-of-mobile-scareware/.

[41] Symantec. Android RATs Branch out with Dendroid. [Online]. Available: http://www.symantec.com/connect/blogs/android-rats-branch-out-dendroid.

[42] Symantec. FakeAV holds Android Phones for Ransom. [Online]. Available: http://www.symantec.com/connect/blogs/fakeav-holds-android-phones-ransom.

[43] Symantec. iBanking: Exploiting the Full Potential of Android Malware. [Online]. Available: http://www.symantec.com/connect/blogs/ibanking-exploiting-full-potential-android-malware.

[44] Symantec. Remote Access Tool Takes Aim with Android APK Binder. [Online]. Available: http://www.symantec.com/connect/blogs/remote-access-tool-takes-aim-android-apk-binder.

[45] W. Tang, G. Jin, J. He, and X. Jiang. Extending Android Security Enforcement with a Security Distance Model. In *International Conference on Internet Technology and Applications (iTAP), 2011*, pages 1–4, Aug 2011.

[46] The University of Waikato. WEKA. [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/.

[47] Wikipedia. Android Version History. [Online]. Available: http://en.wikipedia.org/wiki/Android_version_history.

[48] 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 *Seventh Asia Joint Conference on Information Security (Asia JCIS), 2012*, pages 62–69, Aug 2012.

[49] T. Wyatt, D. L. Richardson, and A. Lineberry. These Aren't The Permissions You're Looking For. [Online]. Available: https://www.defcon.org/images/defcon-18/dc-18-presentations/Lineberry/DEFCON-18-Lineberry-Not-The-Permissions-You-Are-Looking-For.pdf.

[50] Y. Zhou and X. Jiang. An Analysis of the Anserverbot Trojan. Technical report, Tech. Rep., 9 2011. [Online]. Available: http://www. csc. ncsu. edu/faculty/jiang/pubs/AnserverBot Analysis. pdf, 2011.

[51] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy (SP), 2012*, pages 95–109, May 2012.

[52] 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 *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, pages 5–8, 2012.