ADDIS ABABA UNIVERSITY SCHOOL OF GRADUATE STUDIES

INSTITUTE OF TECHNOLOGY

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Lightweight Security Auditing Tool for Android Smart Mobile Phone: Design and Implementation

By

Fetulhak Abdurahman

Thesis submitted in partial fulfillment of the requirements for the Degree of

Master of Science in Computer Engineering

School of electrical and computer engineering

June, 2014

Addis Ababa, Ethiopia

ADDIS ABABA UNIVERSITY SCHOOL OF GRADUATE STUDIES

INSTITUTE OF TECHNOLOGY

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Lightweight Security Auditing Tool for Android Smart Mobile Phone: Design and Implementation

By

Fetulhak Abdurahman

Advisor

Mr. Misikre Tadesse

ADDIS ABABA UNIVERSITY SCHOOL OF GRADUATE STUDIES

INSTITUTE OF TECHNOLOGY

Lightweight Security Auditing Tool for Android Smart Mobile Phone: Design and Implementation

By

Fetulhak Abdurahman

APPROVAL BY BOARD OF EXAMINER

_____	_____
Dean. School of Electrical and Computer Engineering	Signature
_____	_____
Advisor	Signature
_____	_____
Internal Examiner	Signature
_____	_____
External Examiner	Signature

Declaration

I, the undersigned, declare that this thesis work is my original work, has not been presented for a degree in this or any other university, and all sources of materials used for the thesis work have been fully acknowledged.

Name: Fetulhak Abdurahman Signature: _____

Place: Addis Ababa, Ethiopia

Date of submission: June 13, 2014

This thesis has been submitted for examination with my approval as a university advisor.

Adviosor: Mr. Misikre Tadesse Signature: _____

Acknowledgements

This research would not have been possible without the help and assistance of many persons. First I would like to express my gratitude to my supervisor Mr. Misikre Tadesse for his guidance and support throughout my thesis work. I am also thankful to my graduate student colleaguesTeklay Gebremichael**,** Salim Nigussie and Abdilbasit Hamidwho provided great company and support. I also want to thank for the financial and material support I received from Addis Ababa University.

Last but not least, I would like to thank my parents and my brother for their strong moral support. Thank you for being always there for me.

Table of Contents

List of Figures

List of Tables

Abbreviations

3G	Third Generation Cellular Network
aapt	Android Asset Packaging Tool
ADB	Android Debug Bridge
AM	Activity Manager
App	Application
API	Application Programming Interface
APK	Application Package
ARFF	Attribute-Relation File Format
ARM	Acorn RISC Machine
BSD	Berkeley Software Distribution
C&C	Command and Control Server
CPU	Central Processing Unit
DEX	Dalvik Executable code
DVM	Dalvik Virtual Machine
EDGE	Enhanced Data Rates for Global Seystem for Mobile Evolution
FN	False Negative
FNR	False Negative Rate
FP	False Positive
FPR	False Positive Rate
GB	Giga Byte
GHZ	Giga Hertz
GPRS	General Packet Radio Service
GPS	Global Position System
HIDS	Host-based Intrusion Detection Systems

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

ID Identifier

IDS Intrusion Detection System

IMEI International Mobile Equipment Identity

IMSI International Mobile Subscriber Identity

IP Internet Protocol

IPC Inter-Process Communication

MMS Multimedia Messaging Service

NIDS Network-based Intrusion Detection Systems

OS Operating Systems

PC Personal Computer

PDA Personal Digital Assistants

POSIX Portable Operating System Interface

RAM Random Access Memory

SDK Software Development Kit

SIM Subscriber Identity Module

SMS Short Message Service

SQL Structured Query Language

SVM Support Vector Machines

TN True Negative

TNR True Negative Rate

TP True Positive

TPR True Positive Rate

USB	Universal Serial Bus
UID	User Identifier
VM	Virtual Machine
Weka	Waikato Environment for Knowledge Analysis
WIDS	Wireless Intrusion Detection System
Wi-Fi	Wireless Fidelity
XML	Extensible Markup Language

Abstract

Due to the fast growing market in Android smartphone operating systems to date cyber criminals have naturally extended their target towards Google‘s Android mobile operating system. Threat researchers are reporting an alarming increase of detected malware for Android from 2012 to 2013. Static analysis techniques for malware detection are based on signatures of known malicious applications. It cannot detect new malware applications and the attacker will get window of opportunities until the threat databases are updated for the new malware. Malware detection techniques based on dynamic analysis are mostly designed as a cloud based services where the user must submit the application to know whether the application is malware or not.

As a solution to these problems, in this work we design and implement a host based lightweight security auditing tool that suits resource-constrained mobile devices in terms of low storage and computational requirements. Our proposed solution utilizes the open nature of the Android operating system and uses the public APIs provided by the Android SDK to collect features of known-benign and known-malicious applications. The collected features are then provided to machine learning algorithm to develop a baseline classification model. This classification model is then used to classify new or unknown applications either as malware or goodware and if it is malware it alerts the user about the infection.

Our proposed solution has been tested by analyzing both malicious and benign applications collected from different websites. The technique used is shown to be an effective means of detecting malware and alerting users about detection of malware, which suggests that it has the capability to stop the spread of the attack since once the user is aware of the malicious application he can take measures by uninstalling the application. Experimental results show that the proposed solution has detection rate of 96.73% in RandomForest machine learning model which is used during the final development of our proposed solution as an Android application and low rate of false positive rate(0.01). Performance impact on the Android system can also be ignored which is only 3.7-5.6% CPU overhead, 3-4% of RAM overhead and the battery exhaustion is only 2%.

Keywords: Smartphones, Android, Malware detection, Machine Learning, Classification

Chapter 1

1.1 Introduction

Personal Digital Assistants (PDAs), mobile phones and recently smartphones have evolved from simple mobile phones into sophisticated yet compact minicomputers which can connect to a wide spectrum of networks, including the Internet and corporate intranets. Designed as open, programmable, networked devices, smartphones are susceptible to various malware threats such as viruses, Trojan horses, and worms, all of which are well-known from desktop platforms. These devices enable users to access and browse the Internet, receive and send emails, SMSs, and MMSs, connect to other devices for exchanging information/synchronizing, and activate various applications, which make these devices attack targets [1].

A compromised smartphone can inflict severe damages to both users and the cellular service provider. Malware on a smartphone can make the phone partially or fully unusable; cause unwanted billing; steal private information (possibly by Phishing and Social Engineering); or infect the contacts in the phone-book. Possible attack vectors into smartphones include: Cellular networks, Internet connections (via Wi-Fi, GPRS/EDGE or 3G network access); USB/ActiveSync/Docking and other peripherals [1].

The challenges for smartphone security are becoming very similar to those that personal computers encounter and common desktop-security solutions are often being downsized to mobile devices. However, some of the desktop solutions (i.e., antivirus software) are inadequate for use on smartphones as they consume too much CPU and memory and might result in rapid draining of the power source. In addition, most antivirus detection capabilities depend on the existence of an updated malware signature repository, therefore the antivirus users are not protected whenever an attacker spreads previously unencountered malware. Since the response time of antivirus vendors may vary between several hours to several days to identify the new malware, generate a signature, and update their clients‘ signature database, hackers have a substantial window of opportunity. Some malware instances may target a specific and relatively small number of mobile devices (e.g., to extract confidential information or track owner‘s location) and will therefore take quite a time till they are discovered [1].

Mobile operating systems pre-installed on all currently sold smartphones need to meet different criteria than desktop and server operating systems, both in functionality and security. Mobile platforms often contain strongly interconnected, small and less-well controlled applications from various single developers, whereas desktop and server platforms obtain largely independent software from trusted sources. Also, users typically have full access to administrative functions on non-mobile platforms. Mobile platforms, however, restrict administrative control through users where the root user has full access to administrative functions. [8]

With an estimated market share of 70% to 80%, Android has become the most popular operating system for smartphones and tablets [2, 3]. Expecting a shipment of 1 billion Android devices in 2017 and with over 50 billion total app downloads since the first Android phone was released in 2008, cyber criminals naturally expanded their vicious activities towards Google‘s mobile platform. Mobile threat researchers indeed recognize an alarming increase of Android malware from 2012 to 2013 and estimate that the number of detected malicious apps is now in the range of 120,000 to 718,000 [4, 5, 6, 7]. In the summer of 2012, the sophisticated Eurograbber attack showed that mobile malware may be a very lucrative business by stealing an estimated €36,000,000 from bank customers in Italy, Germany, Spain and the Netherlands [9].

Android‘s open design allows users to install applications that do not necessarily originate from the Google Play Store. With over 1 million apps available for download via Google‘s official channel [10], and possibly another million spread among third-party app stores, we can estimate that there are over 20,000 new applications being released every month. This requires malware researchers and app store administrators to have access to a scalable solution for quickly analyzing new apps and identifying and isolating malicious applications.

Google reacted to the growing interest of miscreants in Android by revealing Bouncer in February 2012, a service that checks apps submitted to the Google Play Store for malware [11]. However, research has shown that Bouncer‘s detection rate is still fairly low and that it can easily be bypassed [12, 13]. A large body of similar research on Android malware has been proposed, but none of them provide a complete solution to obtain a thorough understanding of unknown applications: work done by [14, 15] limited by system call analysis only, [16] focuses on taint tracking i.e tracking the flow of sensitive data which leaves the smartphone, [17, 18]

track only specific API invocations, and work done by [19] is bound to use an emulator as sandboxing during analysis of malicious applications.

1.2 Statement of the problem

Even though a lot of work is done on the security mechanism of Android platform there are still vulnerabilities which allow malicious attacks to control access to sensitive information using different techniques. The Google play which is an application market is the best place to infect Android applications with malicious code. Once the malicious application is on the Google play market users download and install the application then the malicious application sends users sensitive and personal data to the attacker without the notice of the owner.

Android has a complex security architecture based on capabilities that can be obtained through a range of permissions that apps may request and users can grant those apps when being installed. Unfortunately, the current documentation of this security architecture is not very well described, and the existing documentation is sometimes incomplete and contradictory, leading to possible vulnerabilities that malicious apps could exploit.

Android is vulnerable to malicious attacks due to the ability of users to load different software off market which opens the door for malicious applications. The other main factor which increases the risk for Android users is problem of software updates and lack of patches.

Most antimalware applications in the market today use static analysis for detection of malicious applications because it is fast and simple. However, static analysis is based on signatures of known malicious applications it cannot detect new malware applications and the attacker will get window of opportunities until the threat databases updated for the new malware.

As described above Android users are still attacked by malicious software therefore it needs an efficient and enhanced technique to detect those malicious attacks and alert the user to stop them. In this thesis a lightweight auditing tool is implemented to detect those malicious attacks and it will be tested for different real world malicious attacks. Auditing tool keeps an eye on the installed applications for performing suspected activity and alerts Android users for such threats.

1.3 Objectives

The objective of this Master's thesis is to design and implement a lightweight security auditing tool for the Android platform. The specific objectives will be divided in to:

- Monitoring the behavior of the system and applications installed on the Android emulator/device and analyze the collected data to detect malicious activities.
- Evaluating our security auditing tool using training and testing set applications, which consists of malware and normal applications, using machine learning classifiers.

1.4 Methodology

In order to achieve the objectives of this thesis we have followed the following distinct phases during the thesis stay:

i. Study of the security requirements and threats in smartphones and the existing malware detection techniques for such systems. This is achieved by reading literature review and contacting professionals online.

ii. The collection of datasets to support our experimental analysis. That is, the collection of both malicious and benign applications from different websites.

iii. Extraction of the features used to learn the machine learning algorithms. These features are extracted by running the applications collected in phase two on Android emulator, a virtual Android mobile device able to run on the computer. The extracted features are used to evaluate the effectiveness and accuracy of our proposed framework.

iv. The design and implementation of the proposed framework, a lightweight behavior-based malware detection technique for Android platform. This is mainly concerned with coding the new, lightweight, which has low computational overhead and low memory consumption, and advanced Android malware detection methods used in this thesis work. The framework is implemented as an Android application, by using Eclipse integrated with Android SDK tool.

v. Evaluating and analyzing the framework designed and implemented in phase four using the dataset(s) collected in phase three.

1.5 Thesis outline

The remainder of this Thesis is organized into the following five chapters: In Chapter 2, we provide background information of the Android architecture. It presents general information on malicious applications, threats and the security model used on the Android platform. Chapter 3 of the report presents a brief review of literature. This chapter reviews different works which are related with our thesis. Chapter 4 discusses the design and implementation of the proposed solution. It points out the different techniques used in our thesis to monitor the behavior of the system and the implementation of the framework on the Android Emulator. Chapter 5 analyzes the experimental results performed on the proposed framework using the whole feature sets monitored from the system and performance evaluation is done using different machine learning algorithms. Finally, conclusions and recommendations for future work are presented in chapter 6.

Chapter 2

Background Information

Before we discuss the details of our analysis framework, it is important to understand how Android and Android applications work. In this chapter, we provide a short introduction into the Android architecture.

2.1 Android System Architecture

The Android software stack is shown in Figure 2.1. In this figure, green items are components written in native code (C/C++), while blue items are Java components interpreted and executed by the Dalvik Virtual Machine. The bottom red layer represents the Linux kernel components and runs in kernel space. In the following subsections, we briefly compiled the various layers from the existing studies [20, 21, and 22].

2.1.1 Linux kernel

Android uses a specialized version of the Linux Kernel with a few special additions. These include wakelocks (mechanisms to indicate that apps need to have the device stay on), a memory management system that is more aggressive in preserving memory, the Binder IPC driver, logger and other features that are important for a mobile embedded platform like Android. The users of the device do not have the privilege to acess the Linux subsystem and some sensitive information which is found under the /system partition which is a read-only partition unless the device is rooted. However, root access can be obtained by exploiting security flaws in Android, which is used frequently by the open-source community to enhance the capabilities of their devices but also by malicious parties to install viruses and malware. Like the rest of the Android, the Linux kernel is also open-source software so that it is freely available for development.

Figure 2-1 Android low level system architecture

2.1.2 Libraries

A set of native C/C++ libraries is exposed to the Application Framework and Android Runtime via the Libraries component. These are mostly external libraries with only very minor modifications such as OpenSSL, WebKit and bzip2. The essential C libraries, codename Bionic, were ported from BSD‘s libc and were rewritten to support ARM hardware and Android‘s own implementation of threads based on Linux futexes.

2.1.3 Android runtime

The middleware component called Android Runtime consists of the Dalvik Virtual Machine (Dalvik VM or DVM) and a set of Core Libraries. The Dalvik VM is responsible for the execution of applications that are written in the Java programming language and is discussed in more detail in Section 2.2. The core libraries are an implementation of general purpose APIs and

can be used by the applications executed by the Dalvik VM. Android distinguishes two categories of core libraries.

- Dalvik VM-specific libraries.
- Java programming language interoperability libraries.

The first set allows in processing or modifying VM-specific information and is mainly used when bytecode needs to be loaded into memory. The second category provides the familiar environment for Java programmers and comes from Apache‘s Harmony. It implements most of the popular Java packages such as java.lang and java.util.

2.1.4 Application framework

The Application Framework provides high level building blocks to applications in the form of various android packages. Most components in this layer are implemented as applications and run as background processes on the device. Some components are responsible for managing basic phone functions like receiving phone calls or text messages or monitoring power usage. A couple of components deserve a bit more attention:

Activity Manager: The Activity Manager (AM) is a process-like manager that keeps track of active applications. It is responsible for killing background processes if the device is running out of memory. It also has the capability to detect unresponsive applications when an app does not respond to an input event within 5 seconds (such as a key press or screen touch). It then prompts an Application Not Responding (ANR) dialog shown in Figure 2.2.

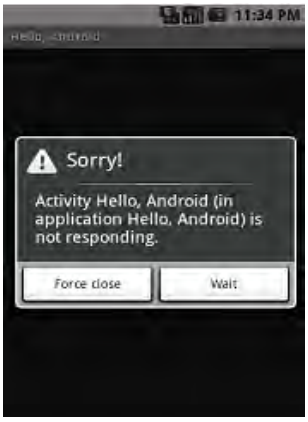


Figure 2-2 ANR dialog [22]

Content Providers: Content Providers are one of the primary building blocks for Android applications. They are used to share data between multiple applications. Contact list data, for example, can be accessed by multiple applications and must thus be stored in a content provider.

Telephony Manager: The Telephony Manager provides access to information about the telephony services on the device such as the phone‘s unique device identifier (IMEI) or the current cell location. It is also responsible for managing phone calls.

Location Manager: The Location Manager provides access to the system location services which allow applications to obtain periodic updates of the device‘s geographical location by using the device‘s GPS sensor.

2.1.5 Applications

Applications or apps are built on top of the Application Framework and are responsible for the interaction between end-users and the device. It is unlikely that an average user ever has to deal with components not in this layer. Pre-installed applications offer a number of basic tasks a user would like to perform (making phone calls, browsing the web, reading e-mail, etc.), but users are free to install third-party applications to use other features (e.g., play games, watch videos, read news, use GPS navigation, etc.).

2.2 Dalvik Virtual Machine

Android‘s design encourages software developers to write applications that offer users extra functionality. Google decided to use Java as the platform‘s main programming language as it is one of the most popular languages: Java has been the number one programming language almost continuously over the last decade, and a large number of development tools are available for it (e.g., Eclipse and NetBeans). Java source code is normally compiled to and distributed as Java bytecode which, at runtime, is interpreted and executed by a Virtual Machine (VM). For Android, however, Google decided to use a different bytecode and VM format named Dalvik. During the compilation process of Android applications, Java bytecode is converted to Dalvik bytecode which can later be executed by the specially designed Dalvik VM.

2.2.1 Hardware constraints

The Android platform was specifically designed to run on mobile devices and thus comes has to overcome some challenging hardware restrictions when com-pared to regular desktop operating systems: mobile phones are limited in size and are powered by only a battery. Due to this mobile character, initial devices contained a relatively slow CPU and had only little amount of RAM left once the system was booted. Despite these ancient specifications, the Android platform does rely on modern OS principles: each application is supposed to run in its own process and has its own memory space which means that each application should run in its own VM.

It was argued that the hardware constraints, made it hard to fulfill the security requirements using existing Java virtual machines [23]. To overcome these issues, Android uses the Dalvik VM. A special instance of the DVM is started at boot time which will become the parent of all future VMs. This VM is called the Zygote process and preloads and preinitializes all system classes (the core libraries discussed in Section 2.1.3). Once started, it listens on a socket and fork()s on command whenever a new application start is requested. Using fork() instead of starting a new VM from scratch increases the speedup time and by sharing the memory pages that contain the preloaded system classes, Android also reduces the memory footprint for running applications.

Furthermore, as opposed to regular stack-based virtual machines — a mechanism that can be ported to any platform — the DVM is register-based and is designed to specifically run on ARM processors. This has allowed the VM developers to add more speed optimizations.

2.2.2 Bytecode

The bytecode interpreted by the DVM is so-called DEX bytecode (Dalvik EXecutable code). DEX code is obtained by converting Java bytecode using the dx tool. The main difference between the DEX file format and Java bytecode is that all code is repacked into one output file (classes.dex), while removing duplicate function signatures, string values and code blocks. Naturally, this results in the use of more pointers within DEX bytecode than in Java .class files. In general, however, .dex files are about 5% smaller than their counter-part, compressed .jar files. It is worth mentioning that during the installation of an Android application, the included classes.dex file is verified and optimized by the OS. Verification is done to reduce runtime bugs

and to make sure that the program cannot misbehave. Optimization involves static linking, inlining of special (native) methods (e.g. calls to equals()), and pruning empty methods.

2.3 Android applications

Android applications are distributed as Android Package (APK) files. APK files are signed ZIP files that contain the app‘s bytecode along with all its data, resources, third-party libraries and a manifest file that describes the app‘s capabilities. Figure 2.3 shows the simplified process of how Java source code projects are translated to APK files.

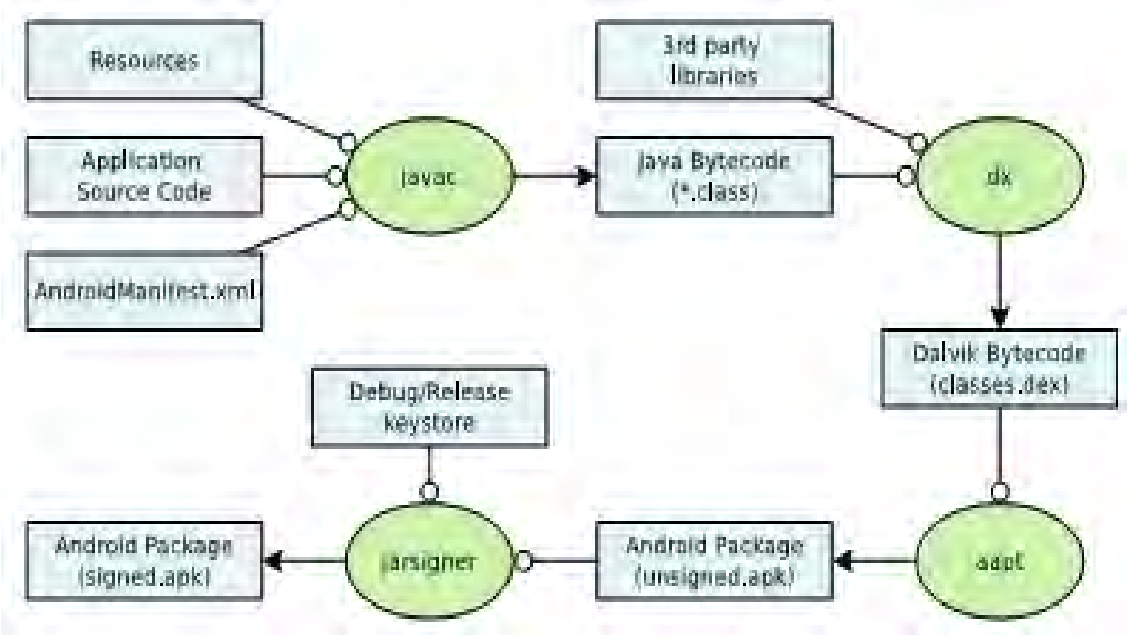


Figure 2-3 Android application build process [22]

To improve security, apps run in a sandboxed environment. During installation, applications receive a unique Linux user ID from the Android OS. Permissions for files in an application are then set so that only the application itself has access to them. Additionally, when started, each application is granted its own VM which means that code is isolated from other applications. It is stated by the Android documentation that this way, Android implements the principle of least privilege as each application has access to only the components it requires to do its work.

2.3.1 Application components

We now outline a number of core application components such as activities, services, content providers, broadcast receivers and intents that are used to build Android apps. For more information on Android application fundamentals, we refer to the official documentation [24].

Activities

An activity represents a single screen with a particular user interface. Apps are likely to have a number of activities, each with a different purpose. A music player, for instance, might have one activity that shows a list of available albums and another activity to show the song that is currently being played with buttons to pause, enable shuffle, or fast forward. Each activity is independent of the others and, if allowed by the app, can be started by other applications. An e-mail client, for example, might have the possibility to start the music app‘s play activity to start playback of a received audio file.

Services

Services are components that run in the background to perform long-running operations and do not provide a user interface. The music application, for example, will have a music service that is responsible for playing music in the background while the user is in a different application. Services can be started by other components of the app such as an activity or a broadcast receiver.

Content providers

Content providers are used to share data between multiple applications. They manage a shared set of application data. Contact information, for example, is stored in a content provider so that other applications can query it when necessary. A music player may use a content provider to store information about the current song being played, which could then be used by a social media app to update a user‘s _current listening‘ status.

Broadcast receivers

A broadcast receiver listens for specific system-wide broadcast announcements and has the possibility to react upon these. Most broadcasts are initiated from the system and announce that, for example, the system completed the boot procedure, the battery is low, or an incoming SMS text message was received. Broadcast receivers do not have a user interface and are generally used to act as a gateway to other components. They might, for example, initiate a background service to perform some work based on a specific event.

Two types of broadcasts are distinguished: non-ordered and ordered. Non-ordered broadcast are sent to all interested receivers at the same time. This means that a receiver cannot interfere with other receivers. An example of such broadcast is the battery low announcement. Ordered broadcasts, on the other hand, are first passed to the receiver with the highest priority, before being forwarded to the receiver with the second highest priority, etc. An example for this is the incoming SMS text message announcement.

Broadcast receivers that receive ordered broadcasts can, when done processing the announcement, decide to abort the broadcast so that it is not forwarded to other receivers. In the example of incoming text messages, this allows vendors to develop an alternative text message manager that can disable the existing messaging application by simply using a higher priority receiver and aborting the broadcast once it finished handling the incoming message.

Intents

Activities, services and broadcast receivers are activated by an asynchronous message called an intent. For activities and services, intents define an action that should be performed (e.g., view or send). They may include additional data that specifies what to act on. A music player application, for example, may send view intent to a browser component to open a webpage with information on the currently selected artist.

For broadcast receivers, the intent simply defines the current announcement that is being broadcast. For an incoming SMS text message, the additional data field will contain the content of the message and the sender‘s phone number.

2.3.2 Manifest

Each Android application comes with an AndroidManifest.xml file that informs the system about the app‘s components. Activities and services that are not declared in the manifest can never run. Broadcast receivers, however, can be either declared in the manifest or may be registered dynamically via the registerReceiver() method. The manifest also specifies application requirements such as special hardware requirements (e.g., having a camera or GPS sensor), or the minimal API version necessary to run this app.

In order to access protected components (e.g., camera access, or access to the user‘s contact list), an application needs to be granted permission. All necessary permissions must be defined in the app‘s AndroidManifest.xml. This way, during installation, the Android OS can prompt the user with an overview of used permissions after which a user explicitly has to grant the app access to use these components.

Within the OS, protected components are element of a unique Linux group ID. By granting an app permissions, it‘s VM becomes a member of the accompanying groups and can thus access the restricted components.

2.3.3 Native code

It may be helpful for certain types of applications to use native code languages like C and C++ so that they can reuse existing code libraries written in these languages. Typical good candidates for native code usage are self-contained, CPU intensive operations such as signal processing, game engines, and so on. Unlike Java bytecode, native code runs directly on the processor and is thus not interpreted by the Dalvik VM. The increased complexity in Android native code should be compromised with suitable performance increase. Native code is not faster so the developer should optimize the native code and try to understand which piece of the application code should be native. If the developer does not have knowledge about native code or has no reason to use the native code it is better to stay away from it.

2.3.4 Distribution

Android users are free to install any (third-party) application via the Google Play Store (previously known as the Android Market). Google Play is an online application distribution

platform where users can download and install free or paid applications from various developers including Google. To protect the Play Store from malicious applications, Google uses an in-house developed automated anti-virus system named Google Bouncer.

Users have the possibility to install applications from other sources than Google Play. For this, a user must enable the unknown sources option in the device‘s settings overview and explicitly accepts the risks of doing so. By using external installation sources, users can install APK files downloaded from the web directly, or choose to use third-party markets. These third-party markets sometimes offer a specialized type of applications, such as MiKandi‘s Adult app store, or target users from specific countries, like Chinese app stores Anzhi and Xiaomi (a popular Chinese phone manufacturer).

2.4 Android Threat

There are several threats targeting the android smart mobile phone, below is some of the most common threats and their explanations. A single malicious application can show more than one of these threats.

2.4.1 Spyware

There are types of threats that spy on the users or steal the data on the user‘s smartphone. There are a number of apps that are the equivalent to commercial keyloggers found on PCs. These apps offer their services to ‗track‘ your kids, spouse or employees. These behaviors are easy to incorporate into an app and this begins with the easy task of requesting the necessary permissions. For example, requesting ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION, and READ_SMS will grant you access to SMS messages and GPS location. Of course you will have to add the appropriate code, and if it is not a rooted device, permissions must be approved. Threats which have used these spying techniques are NickySpy, Spitmo, GGTracker and GoldenEagle. NickySpy is interesting in that it utilizes the MediaRecorder() class to turn on the microphone and discretely record and save conversations to the SDCard. It is also able to send captured data to a remote server, although this functionality is not hard wired in [25].

2.4.2 Root exploit

There are numerous examples of Trojans with root capabilities. These Trojans often have command & control functionality similar to what has been seen with PC botnets. Because these apps root, they gain escalated privileges and are able to bypass Android‘s permission model; thereby granting access to all functionality on the device without user notification. Taking advantage of known exploits in the Android OS, malware authors bundle these exploits in their APK‘s. The rooting exploits are the same ones made available by hackers for those willing to intentionally root their device. The two most prevalent ones target versions 2.1 and 2.2 of the Android OS (rage-againstthecage and exploid). Two notable examples of rooting malware are DroidDream and DroidKungFU. At the time these Trojans were discovered, version 2.1 and 2.2 were the most distributed versions of the Android platform [25].

Trojans misusing these root exploits are among the most dangerous malicious applications and can cause all kinds of havoc, completely out of sight from the user. Like most Trojans, the malicious application pretends to be normal until it is installed on the user's device. When installed, it attempts to use one or more root exploits to gain root access to the device. An application with root access can replace, modify and install applications as it wishes, and as an example, the DroidKungFu Trojan installs a backdoor on the phone once it has gained root access. It then disguises this backdoor from the user both by using an innocent-looking name and hiding the application icon from the user. This backdoor can then be used to install other malicious applications on the device or simply stealing private information [25].

2.4.3 Botnet

A botnet is a network of compromised devices, usually computers, which an attacker can use for his own purposes often to steal sensitive data or as part of a denial of service attack. The owners of the compromised devices might not even be aware of the infection beyond noticing that the device is operating slower than usual. The recent version of the DroidKungFu Trojan was used to create a botnet consisting of compromised Android devices [25].

2.4.4 SMS Trojans

Android malware has evolved its tactics and distribution over the last two years. Two big news makers for Android malware were TrojanSMS, a premium-SMS Trojan, and DroidKungFu, a

bot with rooting capabilities. The premium-SMS Trojan is a lucrative form of malware that is simple to develop and has the added benefit that it uses alluring tactics which users tend to fall for. Recently, two groups that were caught distributing SMS Trojan‘s received some justice. The hackers responsible for the Foncy campaign were arrested in France with damages estimated around $150,000 and the other was a company, A1 Agregator Limited, who was responsible for the payment system RuFraud. They were fined $78,300. The SMS Trojan „FakePlayer‘ was the first fake app to charge for its use. It poses as a legit media player but would send out premium-SMS messages without the user‘s knowledge. There was little sophistication to the malware other than tricking users in to running the program. These APK‘s are very small in size, around 15KB, and when launched display a message in Russian, „Click OK to access the video library” and a second message of „Wait, requested access to the video library.” While you are waiting, premium-SMS messages are sent, costing the unsuspecting user money [25].

2.4.5 Drive-by-download

In May of 2012 the first reports of drive-by downloads targeting Android browsers were seen. Drive-by downloads have been the bane of Internet browsing in the desktop environment for many years now and this infection vector has evolved to target Android devices. This is just another example of how fast the Android malware landscape is evolving. This particular threat typically utilizes a hidden iframe tag located at the bottom of a hacked website. These websites specifically look for Android user-agent strings before serving the malicious iframe, therefore the payload would only be delivered when visiting the site with an Android browser. When the site is visited with an Android browser, the iframe would trigger the browser to download and execute the payload. Another common vector for infection which primarily targets PC‘s is the web exploit kit. BlackHole is a very common example which cycles through various PC based exploits in its attempt to execute a malicious binary. It is not long before web exploit kits will have a module looking to infect mobile devices [25].

2.5 Android Security Overview

Android seeks to be the most secure and usable operating system for mobile platforms by repurposing traditional operating system security controls to:

- Protect user data

- Protect system resources (including the network)
- Provide application isolation

To achieve these objectives, Android provides these key security features:

- Robust security at the OS level through the Linux kernel
- Mandatory application sandbox for all applications
- Secure interprocess communication
- Application signing
- Application-defined and user-granted permissions

At the operating system level, the Android platform provides the security of the Linux kernel, as well as a secure inter-process communication (IPC) facility to enable secure communication between applications running in different processes. These security features at the OS level ensure that even native code is constrained by the Application Sandbox. Whether that code is the result of included application behavior or exploitation of application vulnerability, the system would prevent the rogue application from harming other applications, the Android system, or the device itself.

The foundation of the Android platform is the Linux kernel. The Linux kernel itself has been in widespread use for years, and is used in millions of security-sensitive environments. Through its history of constantly being researched, attacked, and fixed by thousands of developers, Linux has become a stable and secure kernel trusted by many corporations and security professionals [26].

As the base for a mobile computing environment, the Linux kernel provides Android with several key security features, including:

- A user-based permissions model
- Process isolation
- Extensible mechanism for secure IPC
- The ability to remove unnecessary and potentially insecure parts of the kernel

2.5.1 Permissions

Every application that a user installs comes with a request for a set of application-specific permissions that is set by the applications developer. These permissions allow the application to access system (or other application‘s) data or services. These include things like READ_CONTACTS, which grants permission to read the user‘s contact book and SEND_SMS, which allows the application to send SMS messages. Additionally, custom permissions may be created and used by the application. For instance, an application could use a custom permission to restrict other applications from using its service. Permissions for each application are found in their individual AndroidManifest.xml files created by the developer and placed on the mobile device as part of the installation process. These permissions are displayed to the user who must agree to them in order to install the application. An example of this process is shown in Figure 3.1. The permission system [27] provides a good framework for determining what resources an application will have access to once it has been installed on a device. Developers can also create their own permissions [28], which can be used to give other applications access to features in the application. These permissions are not explicitly stated to the users during installation, but can be determined by examining the Android manifest file.

There are four protection levels for permissions:

Normal – Permissions for minor features like VIBRATE. Android package installer will not ask the user for approval for these permissions [29].

Dangerous – Permissions for features that can reconfigure the device or incur fees. Users will be explicitly warned about dangerous permissions on install.

Signature – The permissions are only granted to other applications signed with the same key as this program. Signature permissions are only available to an application that is signed with the same certificate as the certificate that was used to sign the application declaring the permission [29].

SignatureOrSystem – These permissions are for programs installed as part of the system image and typically aren‘t be used by developers.

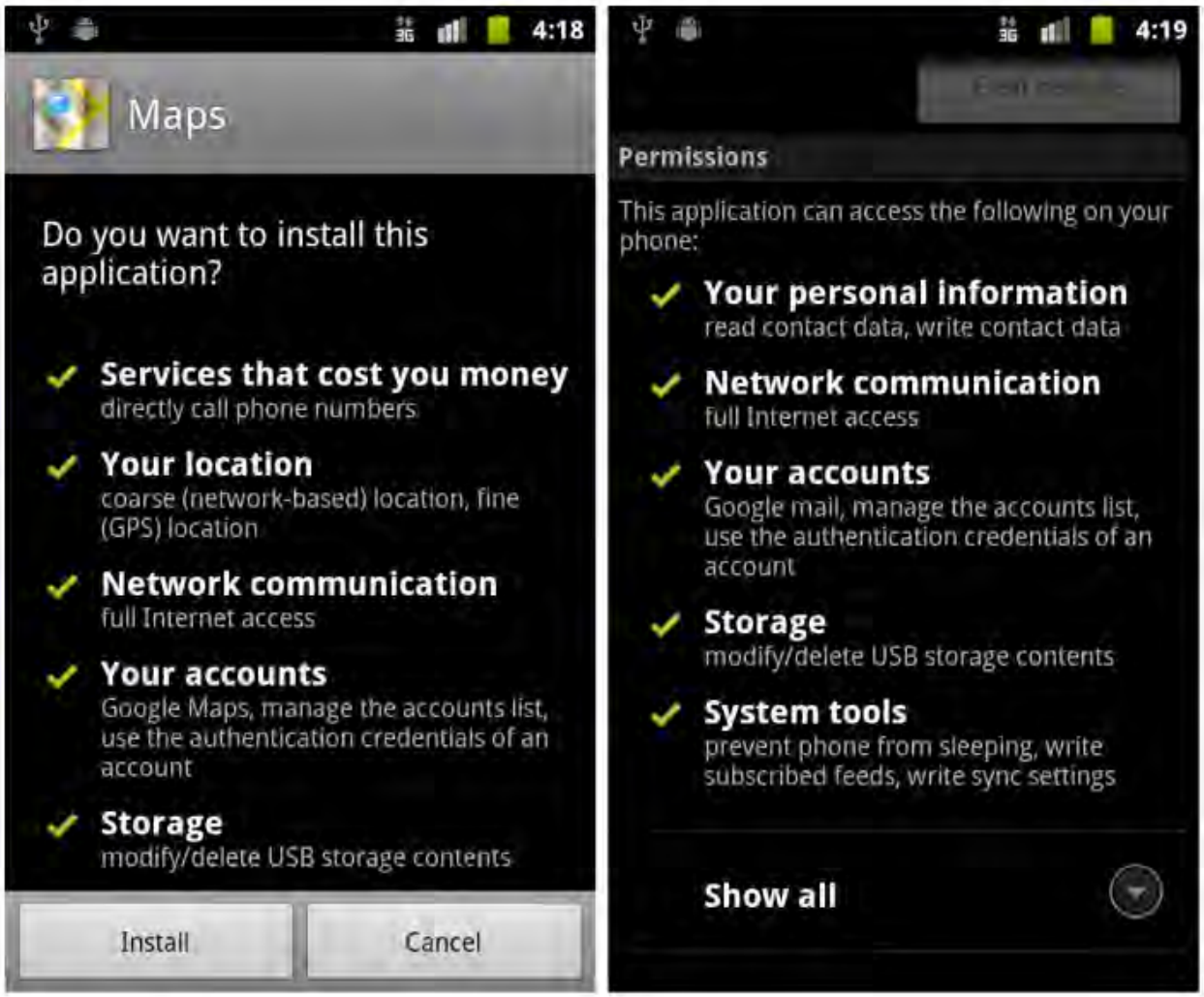


Figure 2-4 Request of permission during installation (left) and the permissions of an installed application (right) [27]

This system of requesting and granting permission puts a great deal of the responsibility for security in the hands of both the developer and user. The user must be aware of both what the application is advertising it does and the permissions it requests. Fortunately, the application reviews on the Android Market and developer‘s reputation may help alert naïve users who attempt to install malicious or insecure applications on their devices.

There are however some issues with the permission system, but these are problems with implementation of the permissions rather than the permissions them-selves. Effectively, the permission system can be circumvented as demonstrated by [30] in their Blackhat talk, where

they revealed that the RECEIVE BOOT COMPLETED permission is not actually checked. This means that any application could register to start when the phone is turned on, and the system would not actually verify whether or not the application had requested this permission. It is currently unknown if this affects any other permissions.

2.5.2 Sandbox

Each Android package (.apk) file installed on an Android-powered device is given its own unique Linux (POSIX) user ID. This user ID is assigned when the application is installed on the device. Consequently, the code of two different packages cannot run in the same process. In a way, this creates a sandbox that prevents one application from touching other applications (or, vice versa, other applications from touching it). In order for two applications to share the same permissions set and possibly run in the same process, they must share the same user ID, which is only possible through the use of the sharedUserID feature. To share the same user ID, two applications must explicitly declare the usage of the same sharedUserID and both must bear the same digital signature .As mentioned in Section 2.5.1, the developer can effectively open the gate to the sandbox by allowing other applications to access features of the application by declaring their own permissions. This makes it possible for other applications to interact with the application despite the sandbox [8, 27].

2.5.3 Application signing

Each application in Android is packaged in an .apk archive for installation. The .apk archive is similar to a Java standard jar file in that it holds all the code (.dex files) for the application. In addition it also contains all the application‘s non-code resources such as images. The Android system requires that all installed applications must be digitally signed (code and non-code resources).The signed apk is valid as long as its certificate is valid and the enclosed public key successfully verifies the signature. Signing applications in Android is used to verify that two or more applications are from the same owner (―same -origin‖ verification). This feature is used by the sharedUserId mechanism and by the permission mechanism to verify Signature and SignatureOrSystem protection level permissions [8].

2.5.4 Remote kill switch

The Google Play application has the ability to remotely remove applications from users' handsets when the application is violating the Developer Distribution Agreement [12] or the Developer Program Policies. In most cases, applications that violate these agreements are malicious in one way or another, and this capability has been utilized to remove malicious applications on more than one occasion after the applications have been removed from the market itself. The remote kill switch is however only useful against applications installed through the Google Play market. Applications installed through unofficial channels are not affected by this feature [31].

2.5.5 File System and User/Group Permissions

As in any Unix/Linux-like operating system, basic access control is implemented through a three-class permission model. It distinguishes between the owner of a file system resource, the owner‘s group and others. For each of these three entities, distinct permissions can be set to read, write or execute. This system provides a means of controlling access to files and resources. For example, only a file‘s owner may write (alter) a document, while members of the owner‘s group may read it and others may not even view it at all. In traditional desktop and server environments, many processes often share the same group or even user ID (namely the user ID of the user who started a program). As a result, they are granted access to all files belonging to the other programs started by that same ID [8].

However, for mobile operating systems this is not sufficient. Finer permission distinction is needed, as an open app market is not a strongly monitored and trustworthy software source. With the traditional approach, any app executed under the device owner‘s user ID would be able to access any other app‘s data. Hence, the Android kernel assigns each app its own user ID on installation. This ensures that an app can only access its own files, the temporary directory and permission protected system resources are available through API calls. This establishes a permission-based file system sandbox [8].

2.5.6 Google Bouncer

Google have responded to criticism about Google Play with introducing a new layer of security, named Bouncer. Bouncer checks new applications when they are uploaded to the market to

identify potentially malicious applications, even going as far as to simulate the application running on an Android device to catch any hidden behavior. This is however an automated process that uses the characteristics of known malware to analyze the applications, which means that novel malware, will not be detected by the bouncer [32].

2.5.7 Anti-malware applications

To identify and remove malware, anti-malware software for mobile devices examines all files in specified locations, email attachments, the memory, system configuration, MMS, Bluetooth objects and other relevant areas. It usually identifies and exterminates known malware based on a signature repository. Anti-malware is a well-known solution and is extensively used in other platforms. Signature-based solutions provide low false-positives, but will only detect known malware and require continuous updating of the signature repository. At this time, the anti-malware solution does not seem to be effective for mobile devices [8].

2.6 Intrusion Detection System

2.6.1 Definition

An Intrusion Detection System, also known as an IDS [35], is a device or software application which monitors a network or system for malicious activities [34].There are many different types of IDS. The aim of an IDS is to identify and detect anomalies in the system or device that is being monitored. Some classes of IDS will be described below.

Network-Based

The Network-Based Intrusion Detection System (NIDS) is an intrusion detection system that analyzes network traffic, makes decisions about the purpose of the traffic and scans the network for suspicious activity [33].

Wireless

The Wireless Intrusion Detection System (WIDS) is similar to the NIDS. Instead of analyzing wired network traffic it can analyze wireless traffic to detect suspicious activity [33].

Host-Based

Host-Based Intrusion Detection Systems (HIDS) monitor all activity that occurs on the host being monitored. This system is capable of monitoring features of the system such as power consumption, opened files, system call logs, etc [33].In this thesis work we use a Host-Based Intrusion Detection System.

2.6.2 Detection types

As regards types of IDS detection, we can divide these into two: Signature-Based or Misuse detection and Anomaly-Based detection.

Misuse detection

The technique of Misuse detection searches for specific indications or patterns of attacks, identifying raw byte sequences, protocol type, port numbers, etc. The aim of this type of detection is to find patterns in raw data. Signatures are then created by a group of experts who analyze the code, behavior and manifestation of the malware. Most antivirus companies still use this technique to create malware signatures and patterns. One of the disadvantages of this detection type is that the system must be familiar with all malware patterns and signatures in advance. This type of detection limits the ability to detect new malware [33].

The process of finding and identifying new types of attacks and malware manually takes experts a great deal of time. Antivirus companies are trying to come up with different alternatives in order to avoid this problem through use of automated processes.

Anomaly-Based detection

Anomaly-Based Intrusion Detection Systems use a prior training phase to establish a model for normal system activity. This mode of detection is first trained on the normal behavior of the system or application to be monitored. Using this model of normal behavior, it is possible to detect anomalous activities that are occurring in the system by searching the system for strange behavior. This technique is more complex and requires more resources than Misuse detection. Despite this, it has the advantage of being able to detect new attacks [33].

Typically, Misuse detection tries to identify/classify the new object by consulting known malware or malicious behavior patterns stored in a signature database. Unknown objects are

compared with database objects, and if a match is found between the unknown object being analyzed and the database object, the unknown object will be considered suspicious or malware. If there is no match, it will be classified as unknown [33].Anomaly-Based detection, on the other hand, creates a pattern of normal behavior based on the system's model of normality. New objects will be compared with the normal behavior pattern, and if any of the objects show any abnormal activity compared to that pattern of normal behavior, they will be considered malicious applications.

Chapter 3

Related Work

Research focusing on Android security and Mobile malware in general has been an increasingly popular topic over the last decade. This chapter provides an extensive overview of related work in the field of Android security.

3.1 Background and Surveys

Malware has been a threat for computers for many years and continues to cause irreparable damage to infected systems. The first attempts to identify and analyze malware on smartphones started by adapting existing PC security solutions and applying them to mobile phones. This was not a feasible solution in light of the high demand placed on resources by antivirus techniques and the power and memory constraints of mobile devices. Since malware and intrusion detection systems have already been the subject of massive research, we will give just a brief review of the evolution of malware and malware detection techniques [36, 37].

The summary of most common malware detection techniques are examined in [38]. Their report examined 45 different malware detection techniques in the fields of anomaly-based detection, specification-based detection and signature-based detection. All techniques explained in this report are very useful background information in order to understand the first approaches to malware detection that can also be used in smartphones.

The paper [39, 40] explored the detection of malicious applications and used different approaches to detection based on dynamic analysis of malicious or infected applications. They used different approaches and detection techniques based on dynamic analysis that are used to detect malicious or infected applications. The paper provides useful information about malware detection techniques and tools used in dynamic analysis of malware. [41] Introduced battery-based intrusion detection, a host-based intrusion detection system. This technique monitors anomalous behavior of smartphone batteries based on power consumption. [42] Evaluated the power consumption of devices with a client application installed on a smartphone using the Symbian OS.The application monitored power consumption data and sent a report to a remote server to analyze and detect anomalies in the system.

SmartSiren, a collaborative virus detection application for Windows Mobile 5 [43]. It collects the communication activity from smartphones and performs system log file analysis to detect anomalous behavior in the system. The system uses a proxy-based architecture that interacts with a client installed on devices in order to avoid a heavy processing load. [44] Presented a novel approach to static malware detection in resource-limited mobile environments. Their approach detects malware by extracting function calls from binaries in order to apply a clustering algorithm to the data. This technique was used for detecting Symbian OS malware depending on a mobile phone's features, such as device efficiency, speed and limited resource usage. In 2006 Symbian was the most widely used smartphone OS and many malware detection techniques were developed for this platform. Due to the imminent growth of smartphones with the Android OS, malware researchers decided to switch their malware detection techniques and security mechanisms to Android platform [45].

A number of studies focus on analyzing Android‘s security mechanisms. Before the first Android phones were even released in 2008, [41] was the first to discuss the security of Android smartphones with a focus on its Linux side. They state that Android‘s open character represents a great opportunity for researching security aspects on mobile devices. One of the first studies done on Android‘s permission model was done by [39]. This work details Android‘s internal components and their interaction.

A research on malicious applications for Android [46] proposed a solution based on monitoring events occurring at the Linux kernel level. They used a monitoring application to extract features such as executed system calls, modified files, etc. from the Linux kernel. These features were used to create the smartphone normality pattern. The same group proposed static analysis in 2009[47] and an Android application sandbox system in 2010[64]. The first report presented a collaborative scenario in which different devices could perform static analysis of malware directly on the phone. The second method used an Android application sandbox, a totally secure environment, to perform static and dynamic analysis. Static analysis disassembled Android APK files to detect malware patterns. During dynamic analysis, all of the events occurring on the device (opened files, accessed files, battery consumption, etc.) were monitored. This sandbox provided a secure environment where malware applications could be executed without any risk of infection.

TaintDroid [16] proposed real-time monitoring and analysis of sensitive data with dynamic taint tracking. This technique taints data from privacy-sensitive sources and applies labels as sensitive data propagates through program variables, files, and inter-process messages. When tainted data leaves the system, the application scans for suspicious outgoing data. [48, 49, 50] have proposed another solution for malware detection on smartphones based on Support Vector Machines (SVM) and learning machines, an extension to the Android mobile phone platform that tracks the flow of privacy-sensitive data through third-party applications. Their research work consists of monitoring smartphone devices to determine their normal behavior and using collected data to train a learning machine. This learning machine will learn the normality model of the smart-phone and applications and alert the user every time it detects a suspicious action. The system proposed by [51, 52] in which they perform a complete malware analysis of the phone in a virtual environment on a remote server. In both reports, they explain how to create replicas from Android devices and apply malware detection techniques to these Android mobile phones. The replicas are an equivalent version of the real mobile devices, and will be sent to the remote server for malware analysis. Mobile phone replicas will run in a secure virtual environment where different malware detection techniques are applied.

Androguard [69] is cloud based Android application analysis framework which decompile and analyze a given application, with the goal to detect malicious applications via signature matching.

An interesting system that uses dynamic analysis for android malware detection is Crowdroid [68]. It uses system call traces of running apps on different Android devices and applies clustering algorithms to detect malwares. The tool they mainly used was strace, which is also available in Linux. Their system hijacked system calls to collect information of events generated by Android applications and created an output file. The output file could be uploaded to remote servers for further analysis to detect malware. In [67] they describe that the method of monitoring and intercepting system calls on kernel level lacks practicability. The most important reasons are listed on their paper are as follows:

1. It is not efficient. System calls are basic interfaces provided by an operating system, and they are the only entrance for processes to enter kernel mode from user mode.

2. It is not applicable for real Android devices. The kernel of a real Android device cannot use loadable kernel modules. In other words, the code written by developers will never have the chance to run on kernel level without recompiling the kernel. Therefore, it is impossible to monitor and intercept system calls on kernel level for real Android devices.

In 2010, [53] performed a comprehensive security assessment on the Android framework. They list a number of possible countermeasures for tackling indicated high-risk threats to Android. Most of these threats are still applicable:

1. Maliciously using the permissions granted to an installed application.

2. Exploiting vulnerability in the Linux kernel or system libraries such as root exploitation.

3. Exposing private content.

4. Draining resources.

5. Compromising the internal/protected network.

The paper proposes several recommendations to improve Android‘s security mechanisms in respect to these threats. Most research started from that moment on focused on threat groups 1) and 3) as it turned out that a large part of Android malware originates from these threats. This is confirmed by a Symantec Research white paper from 2011 that addresses the motivations of recent Android Malware [54]. This paper concludes that the current mobile monetization schemes have a low revenue-per-infection ratio. It was expected that this ratio would increase when more devices store credentials backed by monetary funds. Something realized less than a year in later already in 2012 with the Eurograbber attack campaign, responsible for stealing over €36,000,000 [9]. In [55], they study 1100 popular free Android applications using the ded decompiler. A survey paper from 2011 provides an overview of mobile network security and used attack vectors and make statements on future research [56]. Similar work was done by [57].

On [58] they performed analysis of 46 pieces of iOS, Android, and Symbian malware that spread in the wild from 2009 to 2011. [59] Provides a more extensive research which covers 1260 Android malware samples distributed among 49 different malware families. The huge data set

was released to the research community as the Malgenome Project and has been used by many subsequent research papers. They categorize existing ways Android malware is distributed into three social engineering-based techniques: 1) repackaging; 2) update attacks; and 3) drive-by downloads. For all techniques described in the paper, however, users are always tricked in installing the often over-privileged malicious application.

An existing work which is partially similar to our approach in monitoring the system behavior and relies on machine learning techniques is Andromaly [1], which monitors both the smartphone and user's behaviors by observing several parameters, spanning from sensors activities to CPU usage. 88 features are used to describe these behaviors; the features are then pre-processed by feature selection algorithms. The authors developed four malicious applications manually to evaluate the ability to detect anomalies.

The other research work similar in permission combination analysis to ours is [66]. It defines security rules manually by studying the behavior of different Android applications. The permission rules used in this work are based on the potential for misuse that means they assume if an applications request for sensitive system resources then their rule will alert the user about its maliciousness.The rules are not generated by performing experments on malicious or benign applications. Thus their work is as good as their manually generated rules or patterns.

Most of the other approaches only monitor misbehaviors on a limited number of functionalities such as outgoing/incoming traffic, SMS, Bluetooth and IM, or power consumption and, therefore, their detection accuracy may be higher but their technique of monitoring behavior of system is less general.

Angry Birds Bonus Level, Tip Calculator, Tap Snake, Monkey Jump and Steamy Window are the most famous malicious applications to date on the Android platform. Furthermore, more than 50 infected applications were found on Google's Android market in March 2011, all of them infected with the DroidDream Trojan application [60]. Another attack targeting the Android platform was carried out by J. Oberheide. He developed the Angry Birds Bonus Level for the Android OS. This application was a proof-of-concept malware application to showcase the weak security of the Android marketplace. The Angry Birds Bonus Level malware purports to be an additional bonus level for the famous game Angry Birds. The malicious application downloads

and installs three additional applications on the user's device in order to steal sensitive information. These applications were available in Android's official marketplace for over five months, but were removed after they were discovered to be stealing sensitive information from mobile phone devices [61].

A mobile security service provider [62] discovered a spyware application called Tip Calculator in the Android market. The spyware sent all incoming and outgoing SMS messages in the system to a designated email address. Another piece of spyware with similar characteristics discovered in non-official Android repositories was Steamy Window [63]. A Trojan Horse called Android Pjapps modifies the original version of this application and wages an attack by subscribing to a SMS premium service.

The purpose of this thesis is to improve on and contribute to malware detection strategies for the Android OS by offering up new ideas and techniques. The approach used in the thesis is based on detecting Android malware by monitoring different android device and application behaviours.The aim of monitoring the system behavior is to obtain data enabling us to differentiate between normal and malicious use of a device. A lightweight application is installed on the device which collects different features of the android system and passes the collected features to a machine learning algorithm within a specified period of time for detection analysis.

Chapter 4

Design and Implementation

In this chapter, the scope of the lightweight android security analysis framework will be defined. The first part discuss about the framework‘s design in Section 4.1, followed by implementation of the framework in Section 4.2.

4.1 Design

In order to implement a lightweight host-based framework for behavioral analysis of Android smartphone, we have to come up with a solution for main problems such as:

- What kind of information would we like to collect from the Android platform and the application?
- How to design such framework in resource constraint smart mobile phones?
- Which features best represent the behavior of Android smartphones?

4.1.1 What to collect

In general, in order to apply any machine learning classifier it is important to first be able to collect relevant features from the targeted system as such overview provides good insight about the system. We selected the features/group of features based on their availability (i.e., the ability and ease of their extraction) and based on our experimental analysis of the features that will be most helpful in detecting a malware. Since Android is an open source and extensible platform it allows to extract as many features as we would like [1]. However, extraction of a large number of features on a mobile device is a very inefficient and resource wasting process. Additionally, machine learning classification models and detection with a large number of features is much more computationally expensive. Furthermore, the presence of redundant or irrelevant features may decrease the accuracy of the machine learning algorithm. Thus, in this thesis work only system information that is accessible from Android application-level framework using the APIs ,provided by the Android Software Development Kit (SDK), which represent the system and application behavior are used to avoid computationally extensive analysis. In order to compare

the feature sets used in our thesis work, we have analyzed and evaluated all feature sets used in this thesis separately and, then combined them for final evaluation of the whole framework.

4.1.1.1 Requested Permissions of Apps

In order to obtain these features, we first extracted the permissions used by each of the installed applications. To this extent, we employed the aapt tool (Android Asset Packaging Tool), available within the set of tools provided by the Android SDK. The requested permission of applications were selected for two main reasons: first gathering these features using android API calls has low computing overhead and second they can manifest the behavior of the applications.

Several existing studies have examined Android applications' use of permissions. But most of them use the number of permissions required by the application and they also take into consideration the existence of certain dangerous permissions to predict whether a certain application is goodware or malware. This approach lacks practicability since there are different research works that show the number of permissions required by each of the categories (malware and benign) is similar. Therefore, malware authors can easily defeat the number of permission-based classifiers through merely declaring permissions which are found in benign applications. The number of permission-based classifiers will not also be able to correctly classify repackaged android malware applications which are based on legitimate applications but embeds extra payload to achieve a malicious goal.

In this thesis work a new approach which cannot be tricked by the above techniques is provided. The proposed method in this thesis for the permission feature is that, instead of analyzing individual permissions requested by the applications, we try to find combination of dangerous permissions which are found in malware applications only as well as those found in both. We also analyzed the deviation of these permission combinations in terms of usage by the two application categories (malware and benign).

4.1.1.2 Intent action of Apps

Intents are used to communicate between components of an application or components of different applications. The intent messages are used to activate three of the application components activities, services, and broadcast receivers. In this thesis work we have analyzed the intent action since it can tell us what actions can be taken. Most well-known android malwares

like to listen for special kind of Android system intent information and try to activate their activity to run at the background and send the users sensitive information to remote servers owned by attackers. Thus the intent information is what we want to monitor to find out its relation with android malwares. We have analyzed the intent information of the collected benign and malware applications using the aapt tool in android SDK.

4.1.1.3 Network behavior of Apps

Each mobile application has its own network behavior and monitoring the network behavior of a certain application can be used to identify whether the application is malware or not. Researches usually focus on monitoring network behavior at the level of network layers, such as switches, routers, and gateways. But nowadays researchers show that monitoring applications network behaviors can be used for behavioral malware detection in smartphones. This approach is effective since many malware applications use network communication for their needs, such as sending malicious data or getting user‘s sensitive information/data from the device remotely. Such types of behavior affect the network behavior of the application and an abnormal network behavior of an application running on smart mobile phones, like Android; can be identified by using machine learning algorithms.

In this thesis we have collected network features that can best represent the behavior of android applications by using APIs provided by Android SDK. Below is a list of the collected features:

- Average number of sent\received bytes and average increments in their values: Since malicious applications have a background service that sends and receives data without the user intention they have significant difference with their benign counterparts. Thus these features are chosen for this thesis work.
- Application state: to check whether the application is running in foreground or background. Most of the time malware applications behave to run in background and send sensitive user information to remote server. Therefore it is a reasonable feature for malware detection.
- Average number of sent\received packets and average increments in their values: one of the most characteristics of malwares of android is that they steal user information and send it to a backend server. Therefore the malicious transfer of data with user information

included will have distinguishable size from the other packet sizes originating from benign applications.

4.1.2 Framework design

We decided to build a Java based framework that consists of real time monitoring, collection and analysis of various features of the android platform. The theoretical design for this framework is show in Figure 4.1. There are three major components included in the framework: the feature extractor unit, the collector unit and the classifier.

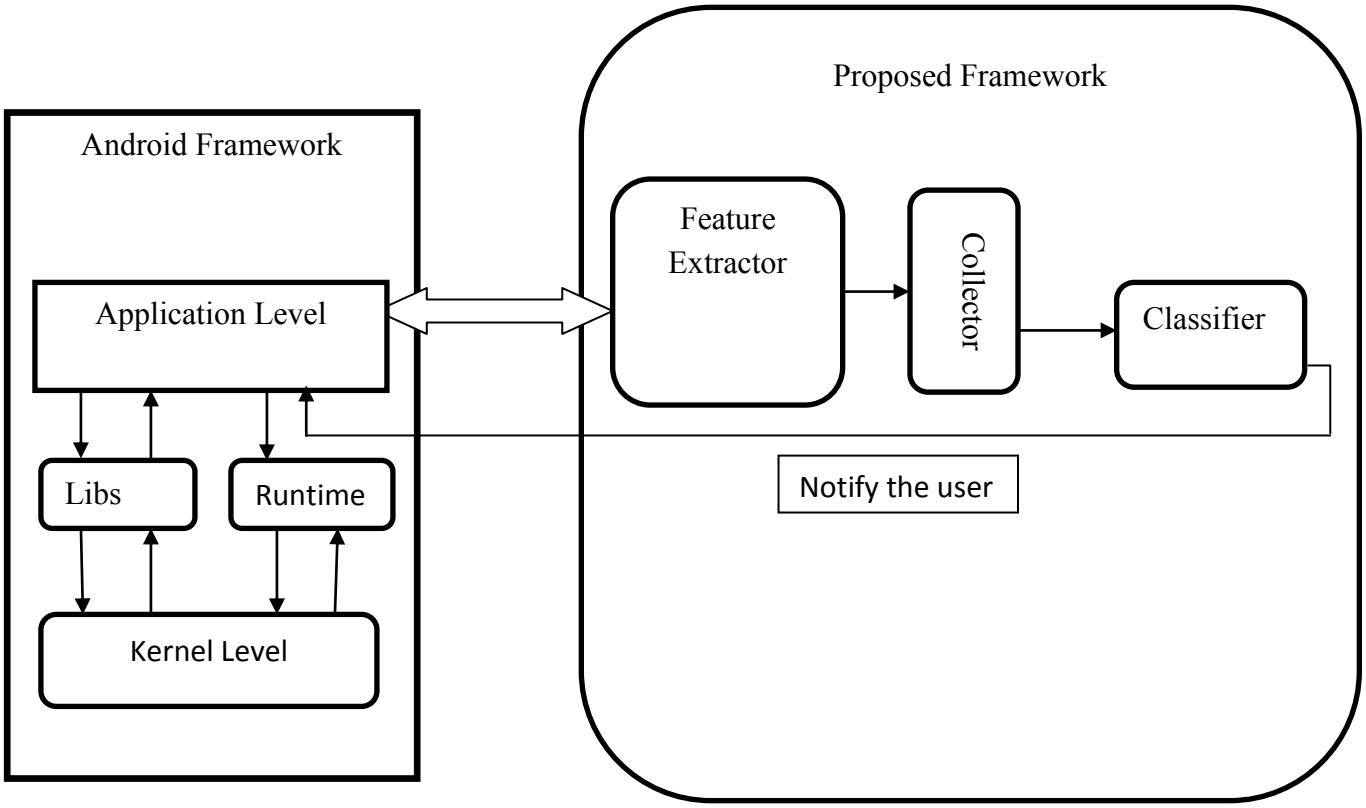


Figure 4-1 Design for Lightweight Android Security Auditing Tool

The feature extractor unit is responsible for extracting features described in section 4.1.1 above from Android application level framework using the APIs provided by the Android SDK. The collector unit is responsible for collecting the features extracted, within the specified period of time (which is 5 seconds in our thesis work), and prepare them in ARFF file format. The

classifier unit then states the vectors given as input from the collector unit as malicious or not. If malicious vectors detected the classifier unit sends a notification to the user.

4.1.3 Dataset description

Benign App Dataset: we have collected the benign apps from Google Play which is one of the largest and most reliable Android markets. We have collected 2,226 applications from all the categories found on the market. The number of benign applications and their category is shown in Table 4.1.

Category	Count	Category	Count
Arcade and Action	120	Music & Audio	108
Books and references	105	News & magazines	105
Business	54	Personalization	109
Card Games	20	Photography	45
Casuals	33	Productivity	48
Comics	53	Puzzles and brain	32
Communication	99	Races	23
Education	83	Shopping	42
Entertainment	105	Social	110
Finance	50	Sports	68
Health	46	Tools	111
Libraries & Demos	97	Transportation	45
Lifestyle	87	Travels	53
Medicine	103	Weather	48
Multimedia & Video	103	Widget	121

Table 4-1The number of applications in each category

Malware Dataset: The Contagio Mobile Dump [70] was used as the source of our malicious data set. Contagio mobile dump site have malicious applications which are uploaded by the public and any one can download it from their database for research propose only. We manually

collected 219 malware applications from this site. The total number of applications is shown in Table 4.2.

Applications	Count
Benign	2226
Malware	219

Table 4-2The total number of applications in our data set

4.2 Implementation

We have implemented the framework on Android emulator, with OS gingearbread version 2.3.6and Linux kernel version 2.6.35.7. The emulator provides a controlled environment for managing different functionalities such as phone calls, SMS messages, network traffic etc. In order to extract and collect the features we have developed an Android application, using Java, running on the Android Emulator or the device. The application collects the most essential features used in this thesis work such as dangerous permission combination used by installed applications, network behavior of running apps, and intent information used for inter process communication. The collector unit records the features collected in .arff file format and the file is sent to classifier which has been previously trained to determine if the vectors collected are similar to those obtained from previously seen malwares. For classification we used Weka version 3.6.6, an open source library in Java that includes several classification tools.

During the training and testing of the classifiers we used a self-written shell script running on a desktop computer with Intel core i3 2350M CPU at 2.30 GHZ and 4GB of RAM. The operating system of this machine is Microsoft Windows 7. The aim of the shell script is to collect as much feature as possible from the Android emulator and applications running on it. The script was used to:

1. Take as input the training APK files in the Training folder and testing APK files in the Testing folder

2. Install/uninstall applications on the emulator
3. Start/stop the feature extractor application
4. Activate the ADB Monkey tool to interact with the applications
5. Output the collected features in ARFF file format
6. Train and test classifiers using the collected feature vectors in step-5

The Training and Testing data folders contain both malicious and benign applications. To train the model classifier we used the APK files found in the Training folder and APK files in the Testing folder are used to test the model. For training phase we use 1557 benign applications and 154 malware applications and for testing phase we use 669 benign and 65 malware applications.

The shell script will install applications from the Training folder and activate the Monkey tool to interact with the installed application. The script then starts monitoring and collecting the features used in this thesis work during the running of the applications on the emulator. Afterwards, the script will uninstall the application from the emulator and create a clean instance of the system or emulator. This ensures that every application has the same initial emulator condition during the feature collection. Applications in the Testing data folder will undergo the same steps as the Training data folder applications. Figure 4.2 describes the procedure of the shell script in detail.

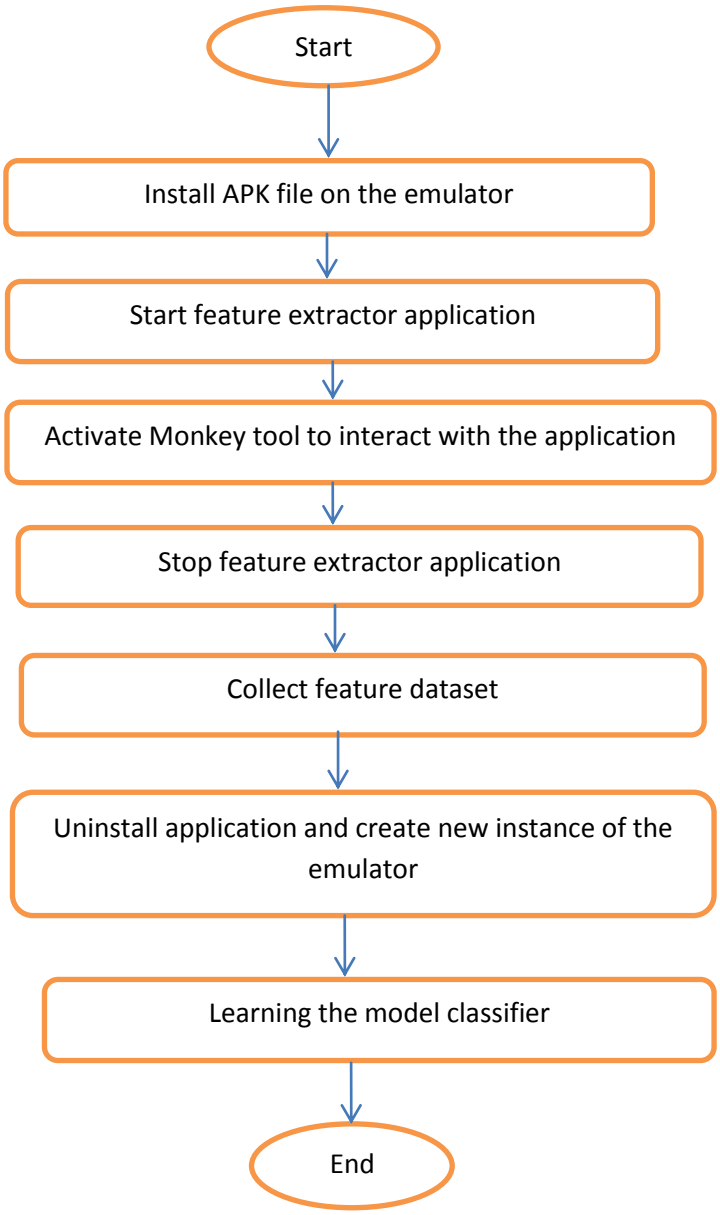


Figure 4-2 Script process during feature extraction and model learning

4.2.1 Tools used during implementation

Table 4.3 describes the tools used during the implementation of the proposed framework.

Tools	Description
Microsoft Windows 7	It was used as the main OS during the framework design.
Eclipse	Eclipse is a platform for programming, development, and Compilation of Java, C++ and many other programming languages. We use Eclipse which is bundled with Android SDK tool to develop the proposed framework.
Android emulator	It is a virtual mobile device included with the Android SDK. We use the emulator to run the applications and extract the features without the need for real devices.
Shell scripts	It was used to automate the feature extraction process using the Android emulator running in Microsoft Windows 7.
Samsung GT-S5300	It is Android OS based mobile device. We use it to measure the performance overhead of the framework.

Table 4-3 Tools used during implementation

Chapter 5

Experimental Result and Evaluation

In this section we will describe in detail the experimental results which are used to show whether or not the features used in this thesis work are effective in detecting malware application behaviors. First we analyze the features individually and evaluate their performance in terms of their detection accuracy using different machine learning algorithms. We use J.48, BaysNet, Naïve Bayes and Random forest machine learning algorithms during evaluation of our proposed solution. During individual feature analysis we do not use the entire applications we have collected since monitoring the behavior of the application using the emulator was taking long time. Then we take the combined features to evaluate our proposed malware detection framework using different machine learning algorithms. This time we monitor the behavior of the entire application. The machine learning algorithm with higher detection accuracy was used as a classifier model during the framework development as Android application.

5.1 Analyzing the requested permission feature

Even though there are existing research works[65,66,71] which focus on analyzing the android application permission request and their combinations none of them try to compare the permission usage in malware and benign applications to find out permission combinations that can be used to detect malicious android application.

In this theses work we have developed a new approach to analyze the android application permission system. We have used an association rule mining algorithm to find out the most widely used dangerous combination of permissions in both malicious and benign applications and compare and contrast their usage in terms of the support value they provide to each category of application (malware and benign). This approach is effective since it cannot be tricked by malware developers unlike those approaches which used the number of requested permission and individual permission based analysis to detect malware. The flow chart given in figure 5.1 shows the procedure followed during analysis of the permission feature in our thesis work.

Our approach for analysis of the permission-based feature set is divided in to two sections:

1. Finding out the most repeatedly used permission combinations

To find out the most repeatedly used permission combinations we have used an association rule mining algorithm called Apriori algorithm. The algorithm works as follows:

I. First generate level-K permission candidates that have high support value than the minimum support given as input for the algorithm. For the minimum support value we have selected the best value by performing experiments with different support value given as input for the algorithm.
The support value is how frequent a certain item set (permission or combination of permission) occurs in the given dataset and it is calculated by:

$$\text{Supportvalue}(X) = \frac{\text{number of transactions that contain X}}{\text{total number of transactions in the given dataset}}$$

In our thesis the transaction is the individual permissions requested by the given application.

II. Then generate the next level-(k+1) permission candidate from the previous level-(k) permission candidate and the algorithm continues until no more candidates to be generated.

2. Calculating the deviation of the permission combinations in terms of their support value. This is step is explained in detail in the following paragraph.

Once we generate the most repeatedly used permission combinations by using step 1 above then we try to find out an interesting permission combination that is unique to malware class or common for both malware and benign class applications. This step is done by comparing the difference between the support values of the permission combinations of malware and benign datasets generated in step 1. We have taken an experimentally selected value of minimum threshold to compare the difference in support value for a given permission combination. For example if Msupp(x) be support value for malicious dataset for permission combination x and Bsupp(x) be the support value for the benign dataset then this permission combination will be interesting if and only if Msupp(x) – Bsupp(x) > minsuppdiff where minsuppdiff is the minimum threshold value given as input for calculating the deviation.

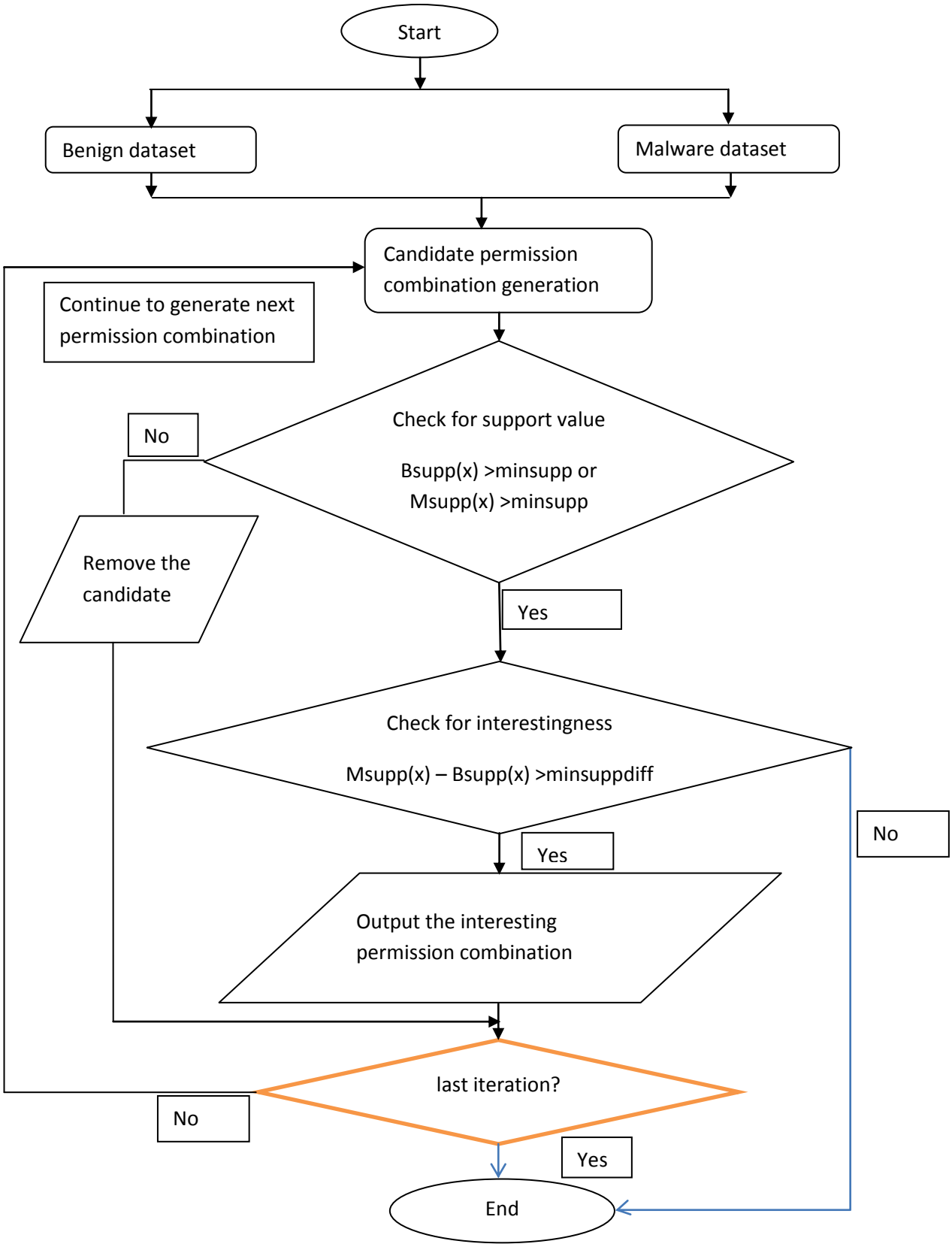


Figure 5-1 Flow chart diagram of the permission feature analysis

Experiment 1: Minimum support value of 0.05 and minimum threshold for difference (minsuppdiff) of 0.20

No	Generated permission combination
1	[android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.RECEIVE_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
2	[android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.RECEIVE_SMS, android.permission.SEND_SMS]
3	[android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.RECEIVE_SMS]
4	[android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
5	[android.permission.INTERNET, android.permission.READ_SMS, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
6	[android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.SEND_SMS]
7	[android.permission.INTERNET, android.permission.RECEIVE_SMS, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
8	[android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.RECEIVE_SMS, android.permission.SEND_SMS]
9	[android.permission.ACCESS_NETWORK_STATE, android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.SEND_SMS]
10	[android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
11	[android.permission.INTERNET, android.permission.READ_SMS, android.permission.RECEIVE_SMS, android.permission.SEND_SMS]
12	[android.permission.INTERNET,android.permission.READ_PHONE_STATE, android.permission.READ_SMS,android.permission.RECEIVE_SMS,android.permission.SEND_SMS]

Table 5-1 Permission combinations generated using experiment 1

We have analyzed the collected benign and malware datasets experimentally by collecting the most essential permission combinations that are unique in malware dataset and common in both datasets. During the experimental analysis of the permission features we have selected different values for the minimum support value and minimum threshold for the difference (minsuppdiff) to get the best value. By comparing the detection accuracy, from the machine learning algorithm, of these experimental results we finally chose to use 0.03 for the minimum support value and 0.1 for the minimum threshold for difference. Table 5.1 shows the permission combinations generated by using the experiment 1.

Next, we trained our classifier by using permission combinations statistics collected by running benign and malware applications on android emulator and we have collected the permission combination features for 589 benign and 180 malware applications. As described at the beginning of the chapter we have used partial data for individual feature experimental analysis. We fed the collected permission combination statistics to the J.48, BaysNet, Naïve Bayes and Random forest classifiers in the WEKA tool and we use testing set data, consisting of 220 benign instances and 37 malware instances, to evaluate the classifiers. The classification results are shown in Table 5.2. It shows that J48 classifier achieves 89.10% accuracy rate, the Random forest achieves 85.60%accuracy rate, the Naïve Bayes achieves 89.50% accuracy rate and the BayesNet classifier achieves 89.88% accuracy rate. But as we can see from the confusion matrix all the classifiers classify large number of malware instances as normal instances (high false positive rate). This is due to lack of permission combinations which can identify malware behavior properly.

To evaluate the performance of the machine learning classification models we also calculate the true positive ratio and true negative ratio as shown in Table 5.2. The true positive ratio is proportion of malware instances classified correctly which is given by the formula:

$$TPR = TP/ (TP+FN)$$

Where TP: number malware instances classified correctly

FN: number of malware instances classified as benign/normal instances

Algorithm	Correctly classified	Incorrectly classified	TPR	TNR	Confusion matrix
J48	229 / 89.10%	28 / 10.9%	0.35	0.98	m n <- - - classified as 13 24 m=malware 4 216 n=normal
Randomforest	220 /85.60%	37/14.40%	0.32	0.94	m n <- - - classified as 12 25 m=malware 12 208 n=normal
Naïve Bayes	230/89.50%	27/10.50%	0.54	0.95	m n <- - - classified as 20 17 m=malware 10 210 n=normal
BayesNet	231/89.88%	26/10.12%	0.54	0.96	m n <- - - classified as 20 17 m=malware 9 211 n=normal

Table 5-2 Permission combination based classifier results for experiment 1

The true negative ratio is proportion of benign or normal instances classified correctly which is given by the formula:

$$TNR = TN/ (TN+FP)$$

Where TN: number of benign or normal instances classified correctly

FP: number of benign or normal instances classified as malware instances

Experiment 2: Minimum support value of 0.04 and minimum threshold for difference (minsuppdiff) of 0.12

No	Generated permission combinations
1	[android.permission.ACCESS_NETWORK_STATE, android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.RECEIVE_SMS, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
2	[android.permission.CALL_PHONE, android.permission.INTERNET, android.permission.READ_CONTACTS, android.permission.READ_PHONE_STATE, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
3	[android.permission.CALL_PHONE, android.permission.INTERNET, android.permission.READ_CONTACTS, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.SEND_SMS]
4	[android.permission.INTERNET, android.permission.READ_CONTACTS, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
5	[android.permission.CALL_PHONE, android.permission.INTERNET, android.permission.READ_CONTACTS, android.permission.READ_SMS, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
6	[android.permission.ACCESS_NETWORK_STATE, android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.RECEIVE_SMS, android.permission.SEND_SMS]
7	[android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.RECEIVE_SMS, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
8	[android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.RECEIVE_SMS, android.permission.SEND_SMS, android.permission.WRITE_SMS]
9	[android.permission.CALL_PHONE, android.permission.INTERNET, android.permission.READ_CONTACTS, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
10	[android.permission.CALL_PHONE, android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
11	[android.permission.ACCESS_NETWORK_STATE, android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.RECEIVE_BOOT_COMPLETED, android.permission.RECEIVE_SMS, android.permission.SEND_SMS]
12	[android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.RECEIVE_BOOT_COMPLETED, android.permission.RECEIVE_SMS, android.permission.SEND_SMS]

Table 5-3 Permission combinations generated using experiment 2

We have used the same training and testing set applications for all the experiments. The results obtained for experiment 2 are shown in Table 5.4. The experimental result shows that J48 classifier achieves 91.05%accuracy rate, the Random forest achieves 91.83%accuracy rate, the Naïve Bayes achieves 89.88%accuracy rate and the BayesNet classifier achieves 89.88%accuracy rate. As we can see from the table the accuracy of the classifiers increase in significant amount except the BayesNet classifier. The accuracy of the classifiers increase because when we see the permission combinations generated in Table 5.3 above there are dangerous permissions included which are used by malware applications to do their dirty work. For example the android.permission.RECEIVE_BOOT_COMPLETED and android.permission.CALL_PHONE are some of the permissions which are included during this experiment and they are requested in higher proportion than their benign counterparts. Thus they can signify the property of malware applications when these permissions are included in our permission combinations. The performance of the machine learning classification models in terms of their True Positive Rate (TPR) and True Negative Rate are also shown in Table 5.4.

Algorithm	Correctly classified	Incorrectly classified	TPR	TNR	Confusion matrix
J48	234 / 91.05%	23 / 8.95%	0.49	0.98	m n <- - - classified as 18 19 m=malware 4 216 n=normal
Randomforest	236 / 91.83%	21 / 8.17%	0.57	0.98	m n <- - - classified as 21 16 m=malware 5 215 n=normal
Naïve Bayes	231 / 89.88%	26 / 10.12%	0.54	0.96	m n <- - - classified as 20 17 m=malware 9 211 n=normal
BayesNet	231 / 89.88%	26 / 10.12%	0.54	0.96	m n <- - - classified as 20 17 m=malware 9 211 n=normal

Table 5-4 Permission combination based classifier results for experiment 2

Experiment 3: Minimum support value of 0.03 and minimum threshold for difference (minsuppdiff) of 0.10

No	Generated permission combinations
1	[android.permission.CALL_PHONE, android.permission.INTERNET, android.permission.READ_CONTACTS, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
2	[android.permission.ACCESS_NETWORK_STATE, android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.RECEIVE_SMS, android.permission.SEND_SMS, android.permission.WRITE_SMS]
3	[android.permission.ACCESS_COARSE_LOCATION, android.permission.ACCESS_FINE_LOCATION, android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE]
4	[android.permission.ACCESS_NETWORK_STATE, android.permission.INTERNET, android.permission.READ_CONTACTS, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.RECEIVE_SMS, android.permission.SEND_SMS]
5	[android.permission.INTERNET, android.permission.READ_CONTACTS, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.SEND_SMS, android.permission.WRITE_EXTERNAL_STORAGE, com.android.launcher.permission.INSTALL_SHORTCUT]
6	[android.permission.ACCESS_NETWORK_STATE, android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.RECEIVE_BOOT_COMPLETED, android.permission.RECEIVE_SMS, android.permission.SEND_SMS]
7	[android.permission.ACCESS_NETWORK_STATE, android.permission.ACCESS_WIFI_STATE, android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.RECEIVE_SMS, android.permission.SEND_SMS]
8	[android.permission.ACCESS_COARSE_LOCATION, android.permission.ACCESS_FINE_LOCATION, android.permission.INTERNET, android.permission.READ_PHONE_STATE, com.android.browser.permission.READ_HISTORY_BOOKMARKS, com.android.browser.permission.WRITE_HISTORY_BOOKMARKS, com.android.launcher.permission.INSTALL_SHORTCUT]

9	[android.permission.INTERNET, android.permission.READ_PHONE_STATE, com.android.launcher.permission.INSTALL_SHORTCUT, com.android.launcher.permission.READ_SETTINGS, com.android.launcher.permission.UNINSTALL_SHORTCUT, com.htc.launcher.permission.READ_SETTINGS, com.motorola.launcher.permission.INSTALL_SHORTCUT]
10	[android.permission.INTERNET, android.permission.READ_CONTACTS, android.permission.READ_PHONE_STATE, android.permission.SEND_SMS, android.permission.SET_WALLPAPER, android.permission.WRITE_EXTERNAL_STORAGE, com.android.launcher.permission.INSTALL_SHORTCUT]
11	[android.permission.CALL_PHONE, android.permission.INTERNET, android.permission.READ_CONTACTS, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.SEND_SMS, android.permission.WRITE_CONTACTS]
12	[android.permission.INTERNET, android.permission.READ_PHONE_STATE, android.permission.READ_SMS, android.permission.SEND_SMS, android.permission.SET_WALLPAPER, android.permission.WRITE_EXTERNAL_STORAGE, com.android.launcher.permission.INSTALL_SHORTCUT]
13	[com.android.launcher.permission.INSTALL_SHORTCUT, com.android.launcher.permission.READ_SETTINGS, com.android.launcher.permission.UNINSTALL_SHORTCUT, com.lge.launcher.permission.INSTALL_SHORTCUT, com.lge.launcher.permission.READ_SETTINGS, com.motorola.dlauncher.permission.INSTALL_SHORTCUT, com.motorola.dlauncher.permission.READ_SETTINGS]

Table 5-5 Permission combinations generated using experiment 3

The results of the classifiers for experiment 3 are shown in Table 5.6. The experimental result shows that J48 classifier achieves 93.39% accuracy rate, the Random forest achieves 94.94% accuracy rate, the Naïve Bayes achieves 91.05% accuracy rate and the BayesNet classifier achieves 91.44% accuracy rate. From the table we can see that we have achieved good accuracy for each of the classifiers where Randomforest being the most accurate one with almost 95% detection accuracy. The increase in detection rate is due to the inclusion of permission combinations which can identify malwares more specifically. If we see experiment 1 and experiment 2 there is no third party permissions included but when analyzing some of the malware families manually we have seen that the third party permissions have their own value to identify malware applications. For example one of the sample

(SHA1:af140ab1gd04bd9e52d8c5f2ff6440f3l9ebc8qr) malware has android.permission. ACCESS _NETWORK_STATE, android.permission.INTERNET, android.permission.READ_PHONE_STATE and android.permission.WRITE_EXTERNAL_STORAGE from the default android permissions but it also includes excessive amount of third party permissions. Thus if we include third party permissions we can increase the detection rate of the classifier and the true negative and true positive rate. We have used the permission combinations generated in this experiment for our final evaluation of the proposed framework.

Algorithm	Correctly classified	Incorrectly Classified	TPR	TNR	Confusion matrix
J48	240 / 93.39%	17/ 6.61%	0.62	0.99	m n <- - - classified as 23 14 m=malware 3 217 n=normal
Randomforest	244 / 94.94%	13 / 5.06%	0.73	0.99	m n <- - - classified as 27 10 m=malware 3 217 n=normal
Naïve Bayes	234 / 91.05%	23 / 8.95%	0.57	0.97	m n <- - - classified as 21 16 m=malware 7 213 n=normal
BayesNet	235/ 91.44%	22 / 8.56%	0.54	0.98	m n <- - - classified as 20 17 m=malware 5215 n=normal

Table 5-6 Permission combination based classifier results for experiment 3

During analysis of the permission of benign and malicious applications we also try to see on individual permissions. Table 5.7 illustrates the top 10 permissions used in both datasets. As we can see from the table in both datasets the android.permission.INTERNET is the most common permission. The android.permission.READ_PHONE_STATE which is used to collect the phone information such as IMEI and the phone number is used in more percent in malware dataset but it also have higher percentage in benign datasets too. Another permission which is used in malware dataset mostly is android.permission.SEND_SMS (47%) when we see its percentage in

benign set it is below 4%. The android.permission.READ_SMS is the next dangerous permission which is used in 39% of the malware application but it is used only in 4% of the benign applications. These two permissions are used by malware applications to send SMS messages to Premium numbers.

There are also other permissions such as android.permission.RECEIVE_SMS, android.permission.READ_CONTACTS, android.permission.RECEIVE_BOOT_COMPLETED which are used mostly in malware applications. If an application requests for a single permission only it will not harm the system but if it request for combination of such dangerous permissions then there is a possibility that the application will harm the system. That is why in this thesis work we focus to analyze the combination of permissions which have been used by malwares frequently.

Benign applications		Malware applications	
android.permission.INTERNET	96%	android.permission.INTERNET	91%
android.permission.ACCESS_NETWORK_STATE	93%	android.permission.READ_PHONE_STATE	67%
android.permission.WRITE_EXTERNAL_STORAGE	70%	android.permission.WRITE_EXTERNAL_STORAGE	48%
android.permission.READ_PHONE_STATE	53%	android.permission.SEND_SMS	47%
android.permission.WAKE_LOCK	41%	android.permission.ACCESS_NETWORK_STATE	46%
android.permission.ACCESS_WIFI_STATE	40%	android.permission.READ_SMS	39%
android.permission.ACCESS_COARSE_LOCATION	31%	android.permission.RECEIVE_SMS	35%
android.permission.ACCESS_FINE_LOCATION	30%	android.permission.READ_CONTACTS	34%
android.permission.GET_ACCOUNTS	26%	android.permission.RECEIVE_BOOT_COMPLETED	33%
android.permission.RECEIVE_BOOT_COMPLETED	25%	android.permission.ACCESS_WIFI_STATE, android.permission.WAKE_LOCK	26%

Table 5-7 Permissions requested by both benign and malware applications and their percentage

5.2 Analyzing the Intent information

Benign applications		Malware applications	
android.intent.action.MAIN	99%	android.intent.action.MAIN	95%
android.intent.action.BOOT_COMPLETED	20%	android.intent.action.BOOT_COMPLETED	40%
android.intent.action.VIEW	24%	android.intent.action.PHONE_STATE	10%
android.intent.action.PACKAGE_ADDED	9%	android.intent.action.USER_PRESENT	6%
android.intent.action.SEARCH	8%	android.intent.action.NEW_OUTGOING_CALL	5%
android.intent.action.PACKAGE_REMOVED	7%	android.intent.action.SIG_STR	4%
android.intent.action.SEND	7%	android.intent.action.VIEW	3%
android.intent.action.PACKAGE_REPLACED	7%	android.intent.action.PACKAGE_ADDED	3%
android.intent.action.PHONE_STATE	4%	android.intent.action.SET_WALLPAPER	2%
android.intent.action.USER_PRESENT	3%	android.intent.action.PACKAGE_REMOVED	2%

Table 5-8 Top 10 Intents used by both benign and malware applications

From Table 5.8 above we can see that some of the intents are used in higher percentage than their benign counterparts. The intent action android.intent.action.BOOT_COMPLETED is used two times more in malware applications than benign applications. The android.intent.action.PHONE_STATE and android.intent.action.USER_PRESENT are also used in higher percentage value in malware than benign applications. In this thesis work we have followed the same approach used for the permission feature analysis. Thus we have selected the most frequently used intent action combinations in malware datasets to detect malicious android applications. We have used the same Apriori algorithm to generate the most repeatedly used intent action combinations in both datasets and then we find for an interesting combination by using the threshold value for difference. The most commonly used intent action combinations in malware datasets are shown in Table 5.9.

No	Generated Intent action combinations
1	android.intent.action.ACTION_POWER_CONNECTED, android.intent.action.BOOT_COMPLETED, android.intent.action.INPUT_METHOD_CHANGED, android.intent.action.UMS_CONNECTED, android.intent.action.UMS_DISCONNECTED
2	android.intent.action.BOOT_COMPLETED, android.intent.action.DATE_CHANGED, android.intent.action.MAIN, android.intent.action.NEW_OUTGOING_CALL, android.intent.action.PHONE_STATE
3	android.intent.action.BOOT_COMPLETED, android.intent.action.MAIN, android.intent.action.NEW_OUTGOING_CALL, android.intent.action.PHONE_STATE, android.intent.action.USER_PRESENT
4	android.intent.action.BOOT_COMPLETED, android.intent.action.INPUT_METHOD_CHANGED, android.intent.action.UMS_CONNECTED, android.intent.action.UMS_DISCONNECTED, android.intent.action.USER_PRESENT
5	android.intent.action.ACTION_POWER_CONNECTED, android.intent.action.BOOT_COMPLETED, android.intent.action.INPUT_METHOD_CHANGED, android.intent.action.UMS_DISCONNECTED, android.intent.action.USER_PRESENT
6	android.intent.action.ACTION_POWER_CONNECTED, android.intent.action.BOOT_COMPLETED, android.intent.action.UMS_CONNECTED, android.intent.action.UMS_DISCONNECTED, android.intent.action.USER_PRESENT
7	android.intent.action.BATTERY_CHANGED, android.intent.action.BOOT_COMPLETED, android.intent.action.MAIN, android.intent.action.PHONE_STATE, android.intent.action.USER_PRESENT

Table 5-9 Intent action combinations generated

During experimental analysis of the Intent actions we use the same threshold values used in experiment 3 for both minimum support value and difference threshold value. We perform the experimental analysis for the intent actions using the same user input values used during the permission combination analysis. The best detection accuracy was achieved when we use the threshold values used in experiment 3. By including the intent actions generated above we can achieve a better detection rate for the classifiers. The experimental result shows that J48 classifier achieves 93.77% accuracy rate, the Random forest achieves 96.50% accuracy rate, the

Naïve Bayes achieves 91.83% accuracy rate and the BayesNet classifier achieves 91.44% accuracy rate. As we can see from Table 5.10 we have achieved almost 97% accuracy using the Randomforest classifier and we can identify most of the malware instances except seven of them. To identify the malware instances which cannot be detected using either their permission combinations or intent actions we analyze their network characteristics.

Algorithm	Correctly classified	Incorrectly Classified	TPR	TNR	Confusion matrix
J48	241 / 93.77%	16/ 6.23%	0.68	0.98	m n <- - - classified as 25 12 m=malware 4 216 n=normal
Randomforest	248 / 96.50%	9 / 3.50%	0.81	0.99	m n <- - - classified as 30 7 m=malware 2 218 n=normal
Naïve Bayes	236 / 91.83%	21 / 8.17%	0.59	0.97	m n <- - - classified as 22 15 m=malware 6 214 n=normal
BayesNet	235/ 91.44%	22 / 8.56%	0.54	0.98	m n <- - - classified as 20 17 m=malware 5 215 n=normal

Table 5-10 Classifier results using permission and intent action combinations

5.3 Analyzing the network behavior of Apps

The permission based malware detection method will be avoided by some malwares. For example if a malware application has the internet permission only then it can download malicious code from the internet and perform its dirty work. Thus monitoring the network behavior of such malwares and others will be used to identify them.

To analyze the network behavior of the apps we monitor and collect the network features listed on section 4.1.1 when the application is running on the emulator for 15 up to 20 minutes. The

ADB Monkey tool is used to interact with the applications during the experiment. The Monkey tool has no actual difference compared to human interaction to activate malicious activity of an application. The following figure shows the network behavior of most network intensive applications monitored during our experimental analysis and a malware sample application. Even though the figures are drawn by using the average received bytes vs. average sent bytes, we can see that different applications have different network behavior and this is very helpful to identify whether a certain application is malware or benign based on its network characteristics.

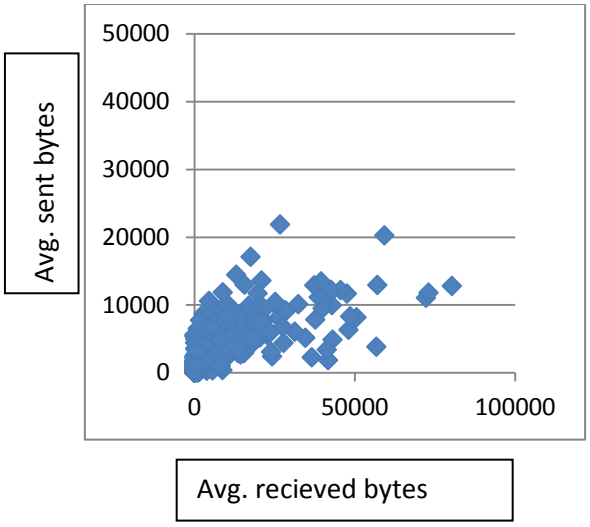
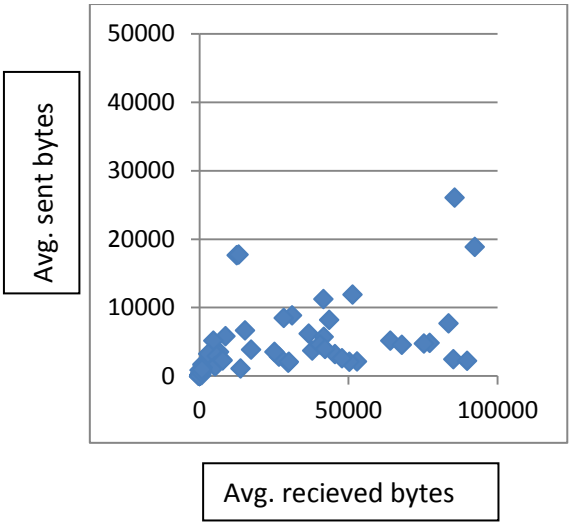
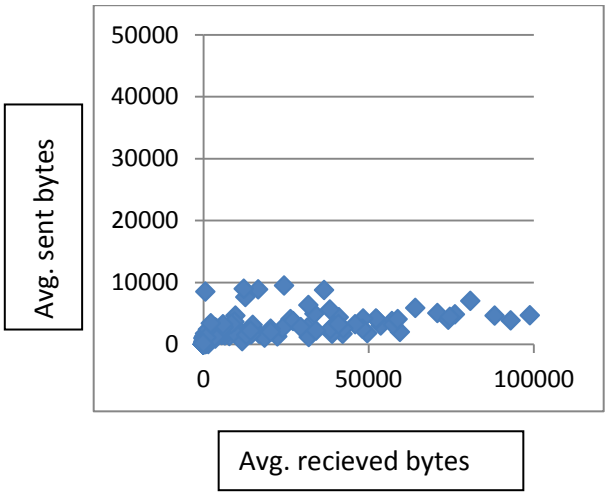


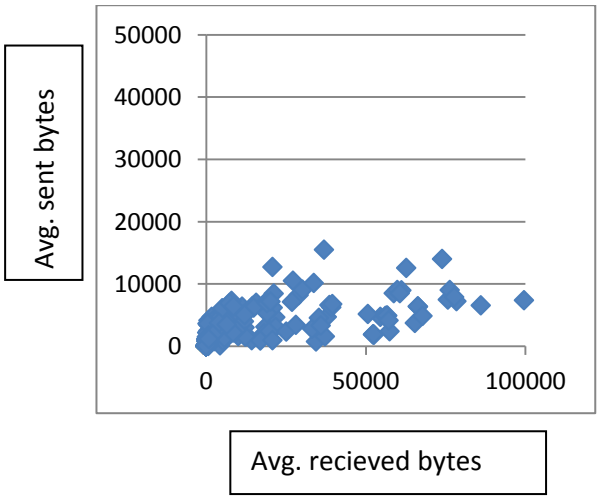
Figure 5-2(a) Network behavior of Facebook



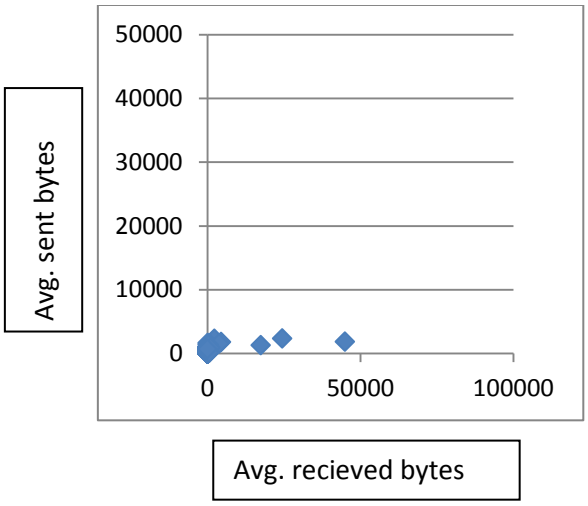
(b) Network behavior of Gmail



(c) Network behavior of Twitter



(d) Network behavior of operamini



(e) Network behavior of Basebridge sample malware

To save resource on the device we want to analyze the network behavior of applications which use the internet. Therefore we have analyzed the network behavior of applications which request the android.permission.INTERNET permission in their manifest fileand we also monitor applications which do not request any permission at all since such applications will use zero permissions flaws to use the internet permission provided for other applications. To analyze the effectiveness of the network behavior we run 100 malware and 400 benign applications on the emulator and collect the network features using the API provided by the Android SDK when each of these applications was run. Then we train the machine learning algorithms used in this thesis work using the extracted network features. Then we use 60 malware and 200 benign applications to evaluate the machine learning algorithms that we train before. The results of the evaluation are shown in Table 5.11. The experimental result shows that J48 classifier achieves 81.54% accuracy rate, the Random forest achieves 86.15% accuracy rate, the Naïve Bayes achieves 72.31% accuracy rate and the BayesNet classifier achieves 77.69% accuracy rate.

Algorithm	Correctly classified	Incorrectly classified	TPR	TNR	Confusion Matrix
J48	212/ 81.54%	35/ 18.46%	0.65	0.865	m n <- - - classified as 3921 m=malware 27 173 n=normal
Randomforest	224/86.15%	36/13.85%	0.73	0.90	m n <- - - classified as 44 16 m=malware 20 180 n=normal
Naïve Bayes	188/72.31%	72/27.69%	0.52	0.79	m n <- - - classified as 31 29 m=malware 43 157 n=normal
BayesNet	202/77.69%	58/22.31%	0.58	0.84	m n <- - - classified as 35 25 m=malware 33 167 n=normal

Table 5-11 Classifier results for network behavior analysis

5.4 Evaluation of Proposed Framework using combined feature set

In this section we will evaluate the proposed framework using the collected feature sets which are analyzed individually in the previous sections. During this experimental analysis our dataset consists of all the features including the permission combinations, the intent information and the network behavior of applications. We have randomly selected 1557 benign applications and 154 malware applications to train the classifiers and we use 669 benign and 65 malware applications to test the trained model classifiers. We have monitored the combined features when the application was running on the Android emulator. The classifier results obtained are shown in Table 5.12.The experimental result shows that J48 classifier achieves 94.96% accuracy rate, the Random forest achieves 96.73% accuracy rate, the Naïve Bayes achieves 92.37% accuracy rate and the BayesNet classifier achieves 94.14% accuracy rate.

Algorithm	Correctly classified	Incorrectly classified	TPR	TNR	Confusion Matrix
J48	697/ 94.96%	37/ 5.04%	0.83	0.96	m n <- - - classified as 54 11 m=malware 26 643 n=normal
Randomforest	710/96.73%	24/3.27%	0.89	0.97	m n <- - - classified as 58 7 m=malware 17 652 n=normal
Naïve Bayes	678/92.37%	56/7.63%	0.72	0.94	m n <- - - classified as 47 18 m=malware 38 631 n=normal
BayesNet	691/94.14%	43/5.86%	0.80	0.96	m n <- - - classified as 52 13 m=malware 30 639 n=normal

Table 5-12 Classifier results for the combined feature set

5.5 Performance overhead analysis

The users of smartphones, especially in Android, are unwilling to use security analysis applications which can degrade the performance of their smartphones. It is therefore necessary that the security tools developed for smartphones should not lower the computational capabilities of the devices.

The main goal of our thesis is to implement a lightweight security auditing tool for android devices. Thus our proposed solution should not impact the device such that the user is aware of such performance degradations. To measure the performance overhead of the solution we use the Task manager application and we measure the CPU, memory consumption as well as the Battery exhaustion period with and without running the proposed solution. We use Samsung GT-S5300, with OS Android Gingerbread version 2.3.6, and Linux kernel version 2.6.35.7 to perform the performance analysis. The service running at background and collecting the features periodically requires an average of 3.7-5.6% of CPU overhead and of 3-4% of RAM space. The device which

was used during our performance overhead analysis has a total of 289 MB of RAM space. The effect of our proposed solution on the battery is analyzed by comparing the battery level difference with and without running the periodic service using the battery monitor of Android settings. The analysis result shows that only 2% of battery level degradation with measurement interval of 20 minutes. During our battery level measurement the discharge rate of the battery was bad that is why it discharges faster.

Chapter 6

Conclusions and Recommendations

Today smartphones are becoming more popular and cheaper and there are different smartphone manufacturers and users of smartphones have increased so greatly. At the same time, attacks for smartphones become increasingly dangerous since they contain personal information including digital images, personal address book and personal documents and performing telephony services such as sending SMS messages to premium rate numbers have economic benefit for attackers. Recently, Android is the most popular smartphone operating system, which is free, open source, and based on embedded Linux. Android platform provides a lot of easily used programming interfaces. Currently, how to detect malware and prevent Android devices from being attacked becomes an area of research. Traditional malware detection methods proposed based on PC architecture is not very applicable to lower computing capability and power-limited smartphones. Thus a lightweight malware detection mechanism suitable for smartphones is desirable.

In this thesis we design and implement a lightweight security auditing tool for android devices. The framework is developed using the APIs provided by the android SDK. It collects features from the Android system which are accessible at the application level and can best describe the behavior of the system and newly installed applications and uses machine learning algorithms for detection of malicious activities.

Our experimental results indicate that our developed framework has better accuracy and low rate of false positive and false negative using the features collected at the application level: permission combinations used by the application, intent actions used by applications for their activation and the network behavior of the applications. From the experiments we realize that Android permission combination analysis, network traffic monitoring and the intent information analysis can provide effective method to determine the behavior of malicious activities on android applications.

Compared to Andromaly [1], our thesis work uses a smaller number of features, and has been tested on real malware, and extract additional features which best describe the android malware

and design new method of monitoring in some features which are also used in [1], and shows better performance in terms of detection. After the learning phase, the false positive rate of our thesis work is 0.01, whereas that of [1] is 0.12. The detection rate of our thesis work is 96% only using permission feature, while that of [1] is 86%.

As a future work, adding more features which can increase the detection accuracy of our framework and analyzing the applications which are not correctly classified to minimize the number of false positives and false negatives. When we monitor features we have to take into account that as we monitor more features we are consuming high amount of resources from the device.

Bibliography

[1] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., Weisee, Y. (2012)"Andromaly": a behavioral malware detection framework for android devices Journal Intelligent Systems, 2012.

[2] Over 1 billion Android-based smart phones to ship in 2017. Canalys, http://www.canalys.com/newsroom/over-1-billion-android-based-smart-phones-ship-2017 Feb.2013.

[3] Ramon Llamas, Ryan Reith, and Michael Shirer. Apple Cedes Market Share in Smartphone Operating System Market as Android Surges and Windows Phone Gains, According to IDC. http://www.idc.com/getdoc.jsp?containerId=prUS24257413, Jul. 2013.

[4] Kindsight Security Labs Malware Report - Q2 2013. Alcatel-Lucent, http://www.alcatel-lucent.com/solutions/kindsight-security.Access date: Jul.2013.

[5] FortiGuard Midyear Threat Report. Fortinet, http:// www.fortinet.com/resource_center / whitepapers / quarterly- threat-landscape-report-q213.html. Access date: Oct. 2013.

[6] Third Annual Mobile Threats Report. Juniper Networks, http://newsroom.juniper.net/press-releases/juniper-networks-finds-mobile-threats-continue-ram-nyse-jnpr-1029552. Access date: Dec. 2013.

[7] TrendLabs 2Q 2013 Security Roundup. Trend Micro,http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-2q-2013-trendlabs-security-roundup.pdf.Access date: Mar. 2013.

[8] Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S. Google Android: A State of the Art Review of Security Mechanisms, Nov. 2009.

[9] Eran Kalige and Darrel Burkey. A Case Study of Eurograbber: How 36 Million Euros was Stolen via Malware, Dec. 2012.

[10] Christina Warren. Google Play Hits 1 Million Apps. http://mashable.com/2013/07/24/google-play-1-million,Access date: Jun. 2013.

[11] Etienne Payet and Fausto Spoto.Static analysis of Android programs. Information and Software Technology, Oct. 2012.

[12] Xuxian Jiang. An Evaluation of the Application ("App") Verification Service in Android 4.2. http://www.cs.ncsu.edu/faculty/jiang/appverify, Access date: Feb. 2013.

[13] Jon Oberheide and Charlie Miller.Dissecting the Android Bouncer, Oct.2012.

[14]Thomas Blasing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak.An Android Application Sandbox System for Suspicious Software Detection.In Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE), Oct. 2010.

[15]Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro.A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors.In Proceedings of the 6th European Workshop on System Security (EUROSEC), Apr. 2013.

[16]William Enck, Peter Gilbert, Byung-Gon Chunn, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Oct. 2010.

[17]Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In Proceedings of the 3rd ACM conference on Data and Application Security and Privacy (CODASPY), Feb. 2013.

[18] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-Sandbox: Having a Deeper Look into Android Applications. In Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC), Mar. 2013.

[19]Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In Proceedings of the 21st USENIX Security Symposium, Aug. 2012.

[20] Stefan Brahler. Analysis of the Android Architecture, Access date: Oct. 2013.

[21] David Ehringer. The Dalvik Virtual Machine Architecture, Access date: Mar. 2013.

[22] Victor van der Veen. Dynamic Analysis of Android Malware, Access date: Aug. 2013.

[23]Dan Bornstein. Dalvik VM Internals. Google I/O, May 2008.

[24] http://developer.android.com/guide/components/fundamentals.html, Access date: May. 2013

[25]Grayson Milbourne and Armando Orozco. Android Malware Exposed. An In-depth Look at the Evolution of Android Malware, Aug. 2012.

[26] http://source.android.com/devices/tech/security/#android-platform-security-architecture, Access date: Jun. 2013.

[27]Google. Android security overview. http://source.android.com/tech/security/index.html, Access date: Jun. 2013.

[28] Google Security and permissions.http://developer.android.com/guide/topics/security/

security.html, Access date: Sep. 2012.

[29] Google. <permission> http://developer.android.com/guide/topics/manifest/permission-element.html, Access date: Sep. 2012.

[30] Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These aren't the permissions you are looking for. In BlackHat, 2010. http://dtors.files.wordpress.com/2010/09/blackhat-2010-final.pdf, Access date: Feb. 2013.

[31] Google. Android market developer program policies. http://www.android.com/us/developer-content-policy.html, Access date: Mar. 2013.

[32] Hiroshi Lockheimer. Android and security.http://googlemobile.blogspot. com/2012/02/ android-and-security.html, Access date: May. 2013.

[33] Iker Burguera Hidalgo. Behavior-based malware detection system for the Android platform, 2011.

[34]By Fengmin Gong, Chief Scientist, Mcafee Network, and Security Technologies. Deciphering detection techniques : Part II anomaly-based intrusion detection, Mar. 2003.

[35] Intrusion Detection System, IDS. http://www.sans.org/reading_room/whitepapers/detection/intrusion-detection-systems-definition_challenges_343, Access date: Jul. 2013.

[36] Malware evolution. http://pages.cs.wisc.edu/~pb/comsnets09.pdf, Access date: Jul. 2013.

[37] Malware economic damage in 2007. http://www.computereconomics.com/page.cfm? = Malware%20Repor,Access date: Jun. 2013.

[38] Nwokedi Idika and Aditya P Mathur. A survey of malware detection techniques. Purdue University, 2007.

[39] Iseclab. International Secure Systems Laboratory. http://www.iseclab.org/, Access date: Aug. 2013.

[40] Manuel Egele. A survey on automated dynamic malware analysis techniques and tools vienna university of technology, 2011.

[41] G A Jacoby and Nathaniel J Davis Iv. Battery-based intrusion detection. Design, page 224, 2005.

[42]Timothy K. Buennemeyer, Theresa M. Nelson, Lee M. Clagett, John P. Dunning, Randy C. Marchany, and Joseph G. Tront. Mobile device profiling and intrusion detection using smart batteries. In Proceedings of the Proceedings of the 41st Annual Hawaii International Conference on System Sciences, HICSS '08, 2008. IEEE Computer Society.

[43] Jerry Cheng, Starsky H Y Wong, Hao Yang, and Songwu Lu. SmartSiren: virus detection and alert for smartphones, ACM, 2007.

[44] Aubrey-Derrick Schmidt, Jan Hendrik Clausen, Ahmet Camtepe, and Sahin Albayrak. Detecting symbian os malware through static function call analysis. 2009 4th International Conference on Malicious and Unwanted Software MALWARE, 2009.

[45]Samsung HTC Smartphone vendor companies market share. http://www.eweek.com/c/a/ Mobile- and -Wireless/Android-Helps-Samsung-HTC-Double-Market-Share-IDC-792965, Access date: Jan. 2013.

[46]Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Jan Clausen, AhmetCamtepe, and Sahin Albayrak. Enhancing security of linux-based androiddevices, 2008.

[47]Aubrey-Derrick Schmidt, Rainer Bye, Hans-Gunther Schmidt, Jan Clausen, Osman Kiraz, Kamer A. Yüksel, Seyit A. Camtepe, and Sahin Albayrak. Static analysis of executables for collaborative malware detection on android. In Proceedings of the 2009 IEEE international conference on Communications, ICC '09, 2009. IEEE Press.

[48] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In Proceeding of the 6th international conference on Mobile systems, applications, and services, MobiSys '08, New York, NY, USA, 2008.

[49] Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. Inf. Secur. Tech. Rep. Feb. 2009.

[50]Ashkan Shari Shamili, Christian Bauckhage, and Tansu Alpcan. Malware detection on mobile devices using distributed machine learning. In Proceedings of the 2010 20th International Conference on Pattern Recognition, ICPR '10, Washington, DC, USA, 2010. IEEE Computer Society.

[51] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, Herbert Bos, and Universiteit Amsterdam. Paranoid android: Zero-day protection for smartphones using the csvunl, 2010.

[52] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In Proceedings of the 26th Annual Computer Security Applications Conference, New York, NY, USA, 2010. ACM.

[53]Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google Android: A Comprehensive Security Assessment. IEEE Security & Privacy, Apr. 2010.

[54] Eric Chien. Motivations of Recent Android Malware. Symantec Security Response, Oct. 2011.

[55]William Enck. Defending Users against Smartphone Apps: Techniques and Future Directions. In Proceedings of the 7th International Conference on Information Systems Security (ICISS), Dec. 2011.

[56] Michael Becher, Felix C. Freiling, Johannes Hoffmand, Thorsten Holz, Sebastian Uellenbeck, and Christopher Wolf. Mobile Security Catching Up? Revealing the Nuts and Bolts of the Security of Mobile Devices. In Proceedings of the 32nd Annual IEEE Symposium on Security and Privacy (S&P), May. 2011.

[57]Timothy Vidas, Daniel Votipka, and Nicolas Christin. All Your Droid Are Belong to Us: A Survey of Current Android Attacks. In Proceedings of the 5th USENIX Workshop on Offensive Technologies (WOOT), Aug. 2011.

[58] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steven Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In Proceedings of the 1st Annual ACM CCS workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), Oct. 2011.

[59] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In Proceedings of the 33rd Annual IEEE Symposium on Security and Privacy (S&P), May 2012.

[60]50 Malware applications found on Android Official Market. http://m.guardian.co.uk/ technology/blog/2011/mar/02/android-market-apps malware?cat= technology&type= article, Access date: Nov. 2013.

[61] Angry Birds Bonus Level. J. Oberheide. http://m.guardian.co.uk/technology/blog/ 2011 / mar/02/ android-market-apps-malware?cat=technology&type=article, Access date: Nov. 2013.

[62]Netqin, mobile security service provider. http://www.netqin.com/en/, Access date: Dec. 2013.

[63] Steamy Window Malware. http://www.netqin.com/en/, Access date: Dec. 2013.

[64]Thomas Bl, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, Sahin Albayrak, and Technische Universit. An android application sandbox system for suspicious software detection. Techniques, 2010.

[65] Zarni Aung, Win Zaw. Permission-Based Android Malware Detection. INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH VOLUME 2, ISSUE 3, MARCH 2013.

[66] W. Enck, M. Ongtang, and P. McDaniel. On LightweightMobile Phone Application Certification. In Proceedings ofthe 16th ACM Conference on Computer and Communications Security, CCS '09, 2009.

[67] PENG Guojun, SHAO Yuru, WANG Taige, ZHAN Xian, ZHANG Huanguo. Research on Android Malware Detection and Interception Based on Behavior Monitoring. Wuhan University Journal of Natural Sciences 2012, Vol.17 No.5.

[68] Burguera L, Urko Z, Simin N.Crowdroid: behavior-based malware detection system for Android [C] //Proc 1st ACM Workshop on Security and Privacy in Smartphones and Mo-bile Devices. New York: ACM Press, 2011: 15-26.

[69] A. Desnos and G. Gueguen. Android: From reversing to decompilation. In Proceedings of Black Hat Abu Dhabi, 2011.

[70] Mila. Contagio mobile. http://contagiominidump.blogspot.com/,Access date: Feb. 2012.

[71] Huang CY, Tsai YT and Hsu CH, "Performance Evaluation on Permission-Based Detection for Android Malware", Proceedings of the International Computer Symposium ICS 2012, Hualien, Taiwan, pp. 111-120, 2013.

Appendix

Code segmentof the extractor unitused to retrieve features using APIs provided by the Android SDK.

```
/*****************************

Code segment to get the number of running processes

************************/

ActivityManager am = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
      List<RunningAppProcessInfo>   processes  = am.getRunningAppProcesses());

for (RunningAppProcessInfo process_running : processes) {

if (((appProcess.importance == RunningAppProcessInfo.IMPORTANCE_FOREGROUND) ||
(appProcess.importance == RunningAppProcessInfo.IMPORTANCE_VISIBLE))) {
for (String package_name : appProcess.pkgList) {
PackageManager pm;

/* ************************************************
Code segment used in the extractor unit that is used to extract the total number of
permissions requested by the running application
***************************************/

PackageInfo Info = pm.getPackageInfo( package_name, PackageManager.GET_ACTIVITIES);

      if ((Info.applicationInfo.flags ApplicationInfo.FLAG_SYSTEM) != 0) {
                  // system processes are not monitored
                        continue;
                  }
else {
PackageInfo info = pm.getPackageInfo(package_name, PackageManager.GET_PERMISSIONS);


      int permissions += info.requestedPermissions.length;



/*********************************************************
Part of the Source code for the extractor unit that is used to check whether the
running application contain any dangerous permission combination
************************************/


BufferedReader permission_combination = new BufferedReader(new InputStreamReader(
                        getAssets().open("permission.txt")));

if (info.requestedPermissions != null) {

while (permission_combination.ready()) {
```

```
       String line = permission_combination.readLine();

       String [] linearray = line.split(",");
       for (String y : linearray) {
       List<String[]>  arlist= new ArrayList<String[]> () .add(y);
                           }
     List<String[]>  perm_comb_array = new ArrayList<String[]> ().add(arlist);
                           }


for (int n = 0; n < perm_comb_array.size(); n++) {
perm_comb_array.get(n);

for (int i = 0; i < perm_comb_array.get(n).length; i++) {
for (int m = 0; m < info.requestedPermissions.length; m++) {

if (perm_comb_array.get(n)[i].equals(info.requestedPermissions[m])) {
              boolean perm_comb_found =true;
                                                        }
                                                  }
                                            }
                                            }
}


/*********************************************
Part of the source code for the extractor unit that is used to extract intent
information of the running application.
**********************/
BufferedReader intent_Info = new BufferedReader(new InputStreamReader(
                           getAssets().open("Intent.txt")));


while (intent_Info.ready()) {

       String line2 = intent_Info.readLine();

   String [] linearray2 = line.split(",");




for (String y : linearray2) {
       List<String[]>  arlist2= new ArrayList<String[]> () .add(y);

                           }
     List<String[]>  intent_array = new ArrayList<String[]> ().add(arlist2);
                           }

       for (int n = 0; n < intent_array.size(); n++) {

                    intent_array.get(n);

          for (int i = 0; i < intent_array.get(n).length; i++) {
              Intent _intent = new Intent(intent_array.get(n)[i]);
```

```
     List<ResolveInfo> resolveInfo = pm.queryBroadcastReceivers(_intent, 0);

       int numberOfapps = resolveInfo.size() - 1;

             if (numberOfapps < 1) {
// the application running does not contain such dangerous intent actions
                         continue;
           }
         else{
   for (int k = 0; k <= numberOfapps; k++) {
       ResolveInfo info = resolveInfo.get(k);
     ActivityInfo activityinfo = info.activityInfo;

       if (activityinfo.packageName.equals(package_name)) {

                   boolean intent_comb_found = true;

                                                  }
                                               }

             }
       }
       }

/*****************************************************
Code segment that is used to extract the network behavior of the running
applications.
*******************************/

int uid = Info.applicationInfo.uid;
long transmitted_packets = TrafficStats.getUidTxPackets(uid);
long transmitted_Bytes = TrafficStats.getUidTxBytes(uid);
long recieved_Packets = TrafficStats.getUidRxPackets(uid);
long recieved_Bytes = TrafficStats.getUidRxBytes(uid);


long increment_of_transmitted_Packets = transmitted_packets -
previous_tansmitted_Packets;
long increment_of_transmitted_Bytes = transmitted_Bytes - previous_tansmitted_Bytes;
long increment_of_recieved_Packets = recieved_Packets - previous_recieved_Packets;
long increment_of_recieved_Bytes =  recieved_Bytes - previous_recieved_Bytes;


 transmitted_packets = transmitted_packets;
    previous_tansmitted_Bytes = transmitted_Bytes;
 previous_recieved_Packets = recieved_Packets;
    previous_recieved_Bytes = recieved_Bytes;

         }
}
}
       }
```

```
/*********************************
Code segment used to collect the features extracted by using the extractor unit
periodically and write them in ARFF file format on external sd-card of the device.
*******************************/
ScheduledExecutorService _scheduledExecutor = Executors.newScheduledThreadPool(1);
ScheduledExecutorService = _scheduledExecutor2 = Executors.newScheduledThreadPool(1);
      _scheduledExecutor.scheduleAtFixedRate(new Runnable() {
            private int attempt = 1;

            public void run() {

           /*********************
              Method used to collect the features using the above code segments
            ******************/
                  collectfeature();

         /*********************
              Code segment used to write the data extracted and returned by the
                 Collectfeature() method to external sdcard

            ******************/


File sdCard = Environment.getExternalStorageDirectory();
File magDir = new File(sdCard.getAbsolutePath() + "/LWSAT/");
               magDir.mkdirs();

FileOutputStream fos;
         String filename = malware.arff;
         String data = collectfeature();
      try {
            File file = new File(magDir, fileName);
            fos = new FileOutputStream(file);
            fos.write( data.getBytes() );
            fos.close();
      } catch (FileNotFoundException e) {
            e.printStackTrace();
      } catch (IOException e) {
            e.printStackTrace();
      }

                  }


      }, 1, 5, TimeUnit.SECONDS);

      _scheduledExecutor2.schedule(new Runnable() {
            public void run() {

            /*******************************
             Code segment used to classify the collected feature vectors
                and alert the user
             ***************/
```

```
InputStream ins = getAssets().open("RandomForest.model");
   ObjectInputStream oins = new ObjectInputStream(ins);
   rfc = ois.readObject();
   ois.close();
  BufferedReader data = new BufferedReader(new FileReader(
                                                magDir +   "/malware.arff"));
           Aninstances newdata = new Aninstances(data);

           data.close();
           data.setClassIndex(test.numAttributes() - 1);


           double  predicted = rfc.classifyInstance(newdata.lastInstance());
           String value = newdata.classAttribute().value((int) predicted);
                 if (perdicted.equals("positive")) {

                       handler.post(new Runnable() {

                             @Override
                             public void run() {
                                   // TODO Auto-generated method stub
          Toast t = Toast.makeText(context, "malware has been detected", 5000);
                                   t.setGravity(Gravity.CENTER, 0, 0);
                                   t.show();
                             }
                       });

                 } else {
                       handler.post(new Runnable() {


                             public void run() {

                 Toast t = Toast.makeText(context, "The application you
                                               are running is safe", 5000);
                                   t.setGravity(Gravity.CENTER, 0, 0);
                                   t.show();
                             }
                       });
                 }

           }
     }, 1200, TimeUnit.SECONDS);
```

Shell script code used to automate the feature extraction process from the Android emulator

```
/*****************************************
```

Shell script code used to creat, start and stop the emulator and to install the feature extractor application

```
**********/

#!/bin/bash

function emulator_creater {

android.bat create avd -n emulator-5554 \

      -t 1  --abi armeabi \

      --force

mksdcard -l e 512M C:\users\admin\desktop\sdcard.img

}

function Start_emulator {

   emulator-arm -port 5554-avd emulator-5554\

       -sdcard C:\users\admin\desktop\sdcard.img\

      -noaudio -wipe-data   -no-boot-anim \

      -no-snapshot -http-proxy 0.0.0.0:0 &

adb -s emulator-5554 wait-for-device

adb -s emulator-5554 install   /Audittool/bin/FETUL.apk

}


function Stop_emulator {

           /cygdrive/c/Windows/system32/taskkill.exe /F /IM emulator-arm.exe

   android.bat delete avd -n emulator-5554

}

function monitor_benign {

   ls Training/N*apk | while read APK; do

Start_emulator
```

```
        monkeytool $APK
    done
}

function monitor_malicious{
    ls Training/M*apk | while read APK; do
Start_emulator
        monkeytool $APK
    done
}
emulator_creater
monitor_benign
monitor_malicious
     data_collector
Stop_emulator
exit 0
```

##

Shell script code used to start the monkeytool and to start and stop the extractor application and collect the data from the application

##

```
#!/bin/bash
function monkeytool {
adb -s emulator-5554 install $1
    start_audittool $1
    pkg=`aapt dump badging $1 |
        grep -o "package: name='[^']*'" | cut -f2 -d \'`
```

```
adb -s emulator-5554shell monkey -p $pkg  --pct-syskeys 0 --pct-anyevent 0  -s `date +%s`  10000
   stop_auditool $1
 adb -s emulator-5554  shell pm disable $pkg
    adb -s emulator-5554 uninstall $pkg
   adb -s emulator-5554shell pm enable fetulhak.abd
}
function start_audittool {
adb -s emulator-5554shell am start -a android.intent.action.MAIN \
       -n fetulhak.abd/.MainActivity
   if [[ $1 == Training/M* ]]; then
       Lable=positive
 else
       Lable=negative
   fi
adb -s emulator-5554 shell "echo $Lable> /mnt/sdcard/LWSAT/classvalue"
}
function stop_audittool {
adb -s emulator-5554pull /mnt/sdcard/LWSAT/malware.arff  /Training/$1.arff
  adb -s emulator-5554 shell pm disable fetulhak.abd
}
function data_collector{
   sed '1,/@data/!d' `find . -name '*.arff' | head -1` >instance.txt
   for feature in $(find . -name '*.arff'); do
       sed '1,/@data/d' $feature > instance2.txt
       sed '1d' instance2.txt >> instance.txt
```

```
        done
    mv tmpTotal.txt /dataset.arff
}
########################################
Script code used to develop model classifier and evaluate it using the WEKA tool
########################################
#!/bin/sh
   Ref1=`cygpath -wp /cygdrive/c/users/admin/desktop/WekaMod.jar`
   Ref2="java -Xmx1g -cp $Ref1"
function develop_Model {

    $Ref2 weka.classifiers.bayes.BayesNet -t /data.arff -d BayesNet.model \
            "-D -Q weka.classifiers.bayes.net.search.local.K2 -- -P 1 -S BAYES
       -E weka.classifiers.bayes.net.estimate.SimpleEstimator
       -- -A 0.5"
    $Ref2 weka.classifiers.trees.RandomForest -t /data.arff -d RandomForest.model "-I 10 -K 0 -S 1"
    $Ref2 weka.classifiers.bayes.NaiveBayes -t /data.arff -d NaiveBayes.model
    $Ref2 weka.classifiers.trees.J48 -t /data.arff -d J48.model "-C 0.25 -M 2"
}
function evaluate_Model {
#Cross validation
time $Ref2 weka.classifiers.bayes.BayesNet  -t data.arff   &> evaluate_BayesNet.txt
time $Ref2 weka.classifiers.bayes.NaiveBayes  -t data.arff   &> evaluate_NaiveBayes.txt
time $Ref2 weka.classifiers.trees.J48  -t data.arff   &> evaluate_J48.txt
time $Ref2 weka.classifiers.trees.RandomForest  -t data.arff   &> evaluate_RandomForest.txt
```

```
#evaluation on testing set data
 time $Ref2 weka.classifiers.bayes.BayesNet  -l /BayesNet.model \
      -T data2.arff &> testevaluation_BayesNet.txt
time $Ref2 weka.classifiers.bayes.NaiveBayes -l /NaiveBayes.model \
      -T data2.arff &> testevaluation_NaiveBayes.txt
time $Ref2 weka.classifiers.trees.J48 -l /J48.model \
      -T data2.arff &> testevaluation_J48.txt
time $Ref2 weka.classifiers.trees.RandomForest -l /RandomForest.model \
      -T data2.arff &> testevaluation_RandomForest.txt
}
```