A MALWARE ANALYSIS AND DETECTION SYSTEM FOR MOBILE DEVICES

ALI FEIZOLLAH

THESIS SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
UNIVERSITY OF MALAYA
KUALA LUMPUR

2017

UNIVERSITY OF MALAYA

ORIGINAL LITERARY WORK DECLARATION

Name of Candidate: Ali Feizollah

Matric No: WHA140017

Name of Degree: Doctor of Philosophy

Title of Project Paper/Research Report/Dissertation/Thesis ("this Work"):

A Malware Analysis and Detection System for Mobile Devices

Field of Study:

Network Security, Malware Detection

I do solemnly and sincerely declare that:

(1) I am the sole author/writer of this Work;
(2) This Work is original;
(3) Any use of any work in which copyright exists was done by way of fair dealing and for permitted purposes and any excerpt or extract from, or reference to or reproduction of any copyright work has been disclosed expressly and sufficiently and the title of the Work and its authorship have been acknowledged in this Work;
(4) I do not have any actual knowledge nor do I ought reasonably to know that the making of this work constitutes an infringement of any copyright work;
(5) I hereby assign all and every rights in the copyright to this Work to the University of Malaya ("UM"), who henceforth shall be owner of the copyright in this Work and that any reproduction or use in any form or by any means whatsoever is prohibited without the written consent of UM having been first had and obtained;
(6) I am fully aware that if in the course of making this Work I have infringed any copyright whether intentionally or otherwise, I may be subject to legal action or any other action as may be determined by UM.

Candidate's Signature Date:

Subscribed and solemnly declared before,

Witness's Signature Date:

Name:

Designation:

ABSTRACT

Smartphones, tablets, and other mobile devices have quickly become ubiquitous due to their highly personal and powerful attributes. Android has been the most popular mobile operating system. Such popularity, however, also extends to attackers. The amount of Android malware has risen steeply during the last few years, making it the most targeted mobile operating system. Although there have been important advances made on malware analysis and detection in traditional PCs during recent decades, adopting and adapting those methods to mobile devices poses a considerable challenge. Power consumption is one major constraint that makes traditional detection methods impractical for mobile devices, while cloud-based techniques raise many privacy concerns. This study examines the problem of Android malware, and aims to develop and implement new approaches to help users confront such threats more effectively, considering the limitations of these devices. First, we present a comprehensive analysis on the development of mobile malware, specifically Android, over recent years, as well as the most useful and salient analysis and detection methods for Android malware. We also discuss a compilation of available tools for Android malware analysis. Secondly, we propose a number of new and distinctive Android malware analysis and detection methods. More specifically, we introduce **AndroDialysis**, which is a static analysis method. Recent research has focused on analysing Android Intent in the XML file. We propose a new method of analysing Android Intent in Java code, which includes implicit intent and explicit intent. We used a Drebin data sample, which is a collection of 5,560 applications, as well as clean data sample containing 1,846 applications. The results show a detection rate of 91% using Android Intent against 83% using Android permission. We also introduce a dynamic analysis method, **AndroPsychology,** in order to analyse the network communications of Android applications. We extracted 30 different features from network traffic. We then used feature selection algorithms and deep learning algorithms to build a detection model.

The results show that network traffic is an appropriate candidate for Android malware detection. Finally, we assembled AndroDialysis and AndroPsychology in order to build a comprehensive analysis and detection system for Android, called **DroidProtect**. Unlike current systems that either perform analyses on the device or send the whole application to a server for analyses, our system has the distinction of extracting features on the device and analysing them on the Google App Engine servers using an offloading technique. Our extensive experiments show that the energy consumption of the proposed system is less than currently available systems.

ABSTRAK

Telefon pintar, tablet dan peranti mudah alih berada dimana-mana sahaja dengan begitu cepat disebabkan oleh sifatnya yang sangat peribadi dan berkuasa. Sehingga 2016, Android merupakan sistem operasi mudah alih yang paling popular di kalangan pengguna. Populariti itu meliputi penyerang juga. Bilangan perisian hasad Android telah melonjak dalam beberapa tahun kebelakangan ini, menjadikannya sistem operasi mudah alih itu yang paling disasarkan. Walaupun kepentigan kemajuan telah dibuat bagi analisis pada perisian hasad dan pengesanan dalam tradisional komputer peribadi dalam tempoh sedekad yang lalu, mengguna pakai dan menyesuaikan analisis untuk peranti mudah alih merupakan satu masalah yang mencabar. Penggunaan kuasa adalah salah satu kekangan utama yang menyebabkan kaedah pengesanan tradisional tidak praktikal untuk dilaksanakan pada peranti mudah alih, manakala teknik berasaskan awan menimbulkan banyak kebimbangan privasi. Kajian ini mengkaji masalah perisian hasad Android, yang bertujuan untuk membangunkan dan melaksanakan pendekatan baru untuk lebih membantu pengguna bagi menghadapi ancaman tersebut, dengan mempertimbangkan had peranti mudah alih. Pertama, kami membentangkan analisis komprehensif mengenai evolusi perisian hasad mudah alih, khususnya Android, sejak beberapa tahun lepas, serta kaedah yang paling berguna dan penting bagi kaedah analisis dan pengesanan dalam pengesanan perisian hasad Android. Kedua, kami mencadangkan beberapa kaedah analisis dan pengesanan terbaru bagi perisian hasad Android. Lebih khusus lagi, kita memperkenalkan AndroDialysis yang merupakan kaedah analisis static. Kerja penyelidikan yang terbaru telah memberi tumpuan kepada menganalisis tujuan Android dalam fail XML. Kami mencadangkan kaedah terbaru menganalisis tujuan Android didalam kod Java, dimana termasuk niat tersirat dan niat yang jelas. Selepas mengekstrak tujuan, model pengesanan dibina menggunakan algoritma Bayesian Network. Kami menggunakan sampel data Drebin iaitu terdapat 5,560 koleksi applikasi terdiri daripada

179 keluarga perisian yang berbeza, serta sampel data bersih yang mengandungi 1,846 applikasi. Keputusan menunjukkan kadar pengesanan sebanyak 91% dengan menggunakan tujuan Android terhadap 83% yang menggunakan kebenaran aplikasi Android. Kami juga memperkenalkan kaedah analisis dinamik, AndroPsychology, untuk menganalisis komunikasi rangkaian bagi aplikasi Android. Kaedah ini memberi tumpuan kepada komunikasi rangkaian yang dijana oleh aplikasi Android. Kami mengekstrak 30 ciri yang berbeza daripada rangkaian trafik. Kemudian, kami menggunakan algoritma pemilihan ciri dan algoritma pembelajaran mesin, untuk membina sebuah model pengesanan. Keputusan menunjukkan bahawa rangkaian trafik adalah calon yang sesuai untuk pengesanan perisian hasad Android. Akhir sekali, kami mengabungkan AndroDialysis dan AndroPsychology untuk membina sistem analisis dan pengesanan yang komprehensif untuk Android, yang dipanggil DroidProtect. Berbeza dengan sistem semasa yang melaksanakan analisis pada peranti atau menghantar keseluruhan aplikasi kepada pelayan untuk dianalisis, sistem kami membawa sesuatu yang baru dalam mengekstrak ciri pada peranti, dan menganalisis aplikasi pada pelayan Engine Google App menggunakan teknik pemunggahan. Tidak perlu dikatakan bahawa eksperimen kami yang meluas menunjukkan penggunaan sistem tenaga adalah kurang pada sistem yang dicadangkan berbanding dengan sistem yang sedia ada.

ACKNOWLEDGEMENTS

The past three years have so far been the most interesting, challenging, and rewarding years of my life. First of all, I would like to express my sincere gratitude to my supervisor Dr Nor Badrul Anuar Bin Juma'at for his patience and knowledge during this long journey; the journey that began at the commencement of my Master's degree. He has been a devoted mentor not only in research, but in many aspects of life. I am grateful for his tremendous academic support, and for giving me wonderful opportunities during these years.

Similar profound gratitude goes to Dr Rosli Bin Salleh, who has been a patient and dedicated mentor. His support and constant faith in my work encouraged me every day to be more diligent in my research.

I am also hugely appreciative of Dr Lorenzo Cavallaro from the Royal Holloway University of London, for accepting my collaboration offer. His professionalism and dedication have inspired me throughout our work.

Of course, this work would not be possible without the support of my beloved parents and my dear siblings. Their continuous support has given me strength to finish this study.

Above all, I want to thank God for all the blessings he has bestowed upon me. His benevolence and grace enabled me to accomplish this study.

"*One does not discover new lands without consenting*
to lose sight of the shore for a very long time."
Andre Gide

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

AIDL	:	Android Interface Definition Language
API	:	Application Program Interface
APK	:	Android Application Package
ARFF	:	Attribute-Relation File Format
C&C	:	Command & Control
CFG	:	Control Flow Graphs
CPU	:	Central Processing Unit
DNN	:	Deep Neural Network
DVM	:	Dalvik Virtual Machine
GPS	:	Global Positioning System
GUI	:	Graphical User Interface
HTTP	:	Hypertext Transfer Protocol
IDE	:	Integrated Development Environment
IMEI	:	International Mobile Equipment Identity
IP	:	Internet Protocol
ISA	:	Iterative Sequence Alignment
JVM	:	Java Virtual Machine
LAC	:	Lazy Associative Classification
LSTM	:	Long short-term memory
MMS	:	Multimedia Messaging Service
OS	:	Operating System
PC	:	Personal Computer
PCAP	:	Packet Capture
RNN	:	Recurrent Neural Network

SDK	:	Software Development Kit
SMS	:	Short Message Service
SOD	:	State of Discharge
SQL	:	Structured Query Language
SVM	:	Support Vector Machine
TCP	:	Transmission Control Protocol
URL	:	Uniform Resource Locator
USB	:	Universal Serial Bus
VM	:	Virtual Machine
VoIP	:	Voice over IP
XML	:	eXtensible Markup Language
XSS	:	Cross-site Scripting

LIST OF APPENDICES

CHAPTER 1: INTRODUCTION

1.1 Background Information

Smartphones have emerged as popular portable devices with increasingly powerful computing, networking and sensing capabilities, and they are now far more powerful than the early PCs. In addition, their popularity has been repeatedly corroborated by recent surveys (Gartner, 2017). Unlike PCs, the portability of mobile devices makes them attractive to users. In addition, their small size in relation to PCs plays an important role in increasing their popularity. Furthermore, users are becoming increasingly interested in Rich Mobile Applications (RMA), such as Google Maps, which deliver rich user experiences along with a high level of interaction (Knoernschild, 2010).

The popularity of such devices is clearly increasing, despite the current limitations of mobile devices such as battery life (B. X. Chen & Bilton, 2014). Gartner, an American information technology research and advisory firm, reported that the total shipment of mobile devices increased in 2013 by 5.9% and reached 2.35 billion units compared to the previous year (Gartner, 2013). Shipments of mobile devices increased by six percent in the third quarter of 2016 compared to 2015 (Gartner, 2016). On the other hand, the shipment of PCs declined by 4.3 percent to 63 million units in 2017 compared to 61 million units in 2016 (Gartner, 2017). Gartner also reported that the shipment of PCs declined by 5.7 percent in the third quarter of 2016 to roughly 68.9 million units. According to the report, PC shipment has decreased for eight quarters in a row (Ram, 2016). In terms of mobile device usage, Walker Sands published a report indicating that internet traffic pertaining to mobile devices has increased. Based on the report, 51.3% of all web traffic came from mobile devices compared to 48.7% of visits from PCs (StarCounter, 2016).

There are numerous mobile operating systems in the market, namely Android, iOS, Windows Phone and BlackBerry. Android has generally dominated the mobile device industry. Based on a report, a total of 261.1 million devices were shipped in the third quarter of 2013, and 81.3% of those shipped devices were operating the Android system (CNET, 2013). It has also been reported that Android had 88% of the worldwide market share of mobile operating systems in the third quarter of 2016 (Gartner, 2016).

Such popularity poses serious security and privacy threats, and widens the potential for various other malicious activities. The number of Android attacks is steadily increasing. Based on a report from F-Secure, Android was subject to 79% of all malware in 2012 compared to 66.7% in 2011 and just 11.25% in 2010 (F-Secure, 2013). Similarly, Symantec has said that the amount of Android malware increased almost four times between June 2012 and June 2013 (Symantec, 2013). In addition, during the period April 2013 to June 2013 there was a dramatic increase of almost 200% in Android malware. Fortinet (Fortinet, 2014), a world leader in high performance network security, announced that between January 1, 2013 and December 31, 2013, they discovered over 1,800 new distinct families of malware, the majority of which was Android malware. In February 2014, Symantec stated that an average of 272 new malware and five new malware families are discovered every month, targeting specifically the Android operating system (Symantec, 2014a).

The reason for such an enormous increase in Android malware lies in the fact that Android is an open source operating system, and the application market for Android, known as Google Play, is not monitored meticulously in terms of security (Teufl et al., 2013). Moreover, there are also unofficial Android markets, for example SlideME, in which security issues are simply not taken seriously. Furthermore, as already mentioned, the

market share of Android is high. Consequently, attackers target Android in order to gain more benefits compared to other operating systems.

1.2 Motivation

This dissertation is motivated by the following open research issues: *there is more mobile malware than before, and it is becoming more sophisticated.*

a) There is sustained growth in the number of mobile devices sold, as well as in mobile malware. Based on a report by Gartner, sales of mobile devices increased by 4.3% in the second quarter of 2016 compared to the same period in 2015. The Android operating system in particular had 86.2% of the market share in 2016 compared to 82.2% in 2015 (Gartner, 2016). A similar trend is seen for mobile malware. The first half of 2016 saw a sharp rise in mobile malware; it almost doubled compared to the same period in 2015 (Nokia, 2016).

b) There is also an increase in the sophistication of mobile malware. As malware detection methods evolve, attackers use new techniques to evade these methods. Android.Obad is most complex malware discovered to the date, and it was dubbed villain of the year of 2013 (Kaspersky, 2013). It uses heavy encryption in its code. In February 2016, Kaspersky Lab reported the discovery of Acecard, one of the most dangerous types of malware. In March 2016, they announced that they had discovered Triada, described as a complex, stealthy, and professionally written malware. It is capable of making any application an agent for performing malicious activities (Kaspersky, 2016b).

These issues call for new and distinctive detection methods. Google, as the owner of Android, has taken security precautions in order to tackle mobile malware. In 2012, it introduced Bouncer, a system that vets applications prior to publishing on Google Play. Google announced that they scan six billion applications per day. It is not feasible, however, to introduce very strict rules, as they affect privacy issues.

Thus, the situation leads to an urgent need for new detection techniques. However, this poses major challenges. One issue is the limited resources of devices, such as battery life. Many applications consume too much power, resulting in limitations. This situation challenges us to develop new methods, with power consumption as an important factor.

1.3 Problem Statement

Since the introduction of the Android operating system, its popularity has increased, and continues to do so. Over time, attackers saw Android as a lucrative target. Thus, they developed malware for Android. The growth of Android malware has been steady, in terms of both volume and complexity.

Many researchers have addressed malware detection experimentation. However, attackers have always tried to find a way to evade new detection methods. It is necessary to develop new analysis and detection methods in order to detect malicious activities. Furthermore, as already mentioned, Google introduced a system called Bouncer to analyse applications before publishing them in Google Play store (Google, 2016). However, this system has proved to be ineffective, since malware are still seen inside the store (Kaspersky, 2016a).

Moreover, despite recent advances in processing power and memory, battery life remains a limitation in mobile devices. Many applications, including current detection methods, consume too much power. (Polakis et al., 2015) conducted an experiment in which the power consumption of malware detection applications was measured. They calculated the energy consumed by the device display and the CPU. Figure 1.1 shows the average energy consumption of the applications, namely AVG, Dr.Web, Sophos, Avast, Norton, and NQ. It is worth mentioning that the authors were unable to measure the energy consumption of the display for the NQ security application, which is the reason it is not present in sub-figure a.

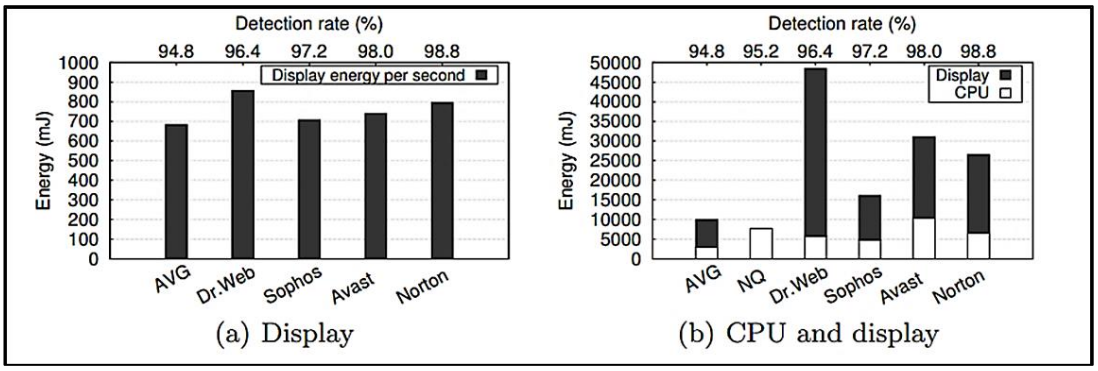


Figure 1.1. Average Energy Consumed Per Second During 'On Demand' Scan (Polakis et al., 2015)

We have calculated the energy consumption of YouTube during 10 minutes of usage. Table 1.1 shows the comparison between NQ and YouTube applications, considering the lowest amount in sub-figure b to be around 6,000 for the NQ application.

Table 1.1. Energy Consumption of Two Applications during 10 Minutes of Usage

Application	Energy Consumption in Joules
YouTube	551.59
NQ	3,600

We calculated the energy consumption of the NQ application for 10 minutes as follows. The 6,000 millijoules mentioned is for one second, and the YouTube consumption of 551.59 Joules is for 10 minutes. If we multiply 6,000 by 600 (to get 10 minutes of usage), and divide it by 1000 (for a millijoule to joule conversion), the result is 3,600 Joules in 10 minutes. It is clear that the NQ application consumes approximately 6.52 times more energy than the YouTube application. It is worth noting that the calculations were made for the lowest level of energy consumption of the malware detection applications. Others will consume much more energy than the NQ application.

This dissertation therefore deals with the problem of the implementation of mobile malware analysis and detection methods on Android devices. It focuses specifically on the limitations of the battery life of such devices.

1.4 Aims and Objectives

The aim of this study is to propose a new framework for analysing and detecting Android malware, focusing on minimising the energy consumption of the proposed solution. In order to achieve this aim, several issues need to be thoroughly examined, analysed, and evaluated. They are:

a) To study the development and current state of Android malware as well as current analysis and detection methods.

b) To design and propose a new framework for Android malware analysis and detection.

c) To evaluate the proposed framework in terms of detection accuracy by using real-world malware.

d) To implement the proposed framework and measure energy consumption of the application, comparing it with similar products.

Due to the overwhelming amount of Android malware, this work centres on the Android operating system. However, the general principle and proposed architecture is applicable to other mobile devices.

The above objectives are dealt with in the following chapters, the structure of which is presented in the next section.

1.5 Thesis Structure

Chapter 2 presents an overview of the development of Android malware since its appearance. It then discusses Android architecture in detail. This section helps to understand various parts of the operating system used in malware detection. The characteristics of Android malware are discussed in the next section. Discussing malware traits helps to develop detection methods. We treat in some depth malware analysis

methods, which in turn helps to address the question of **what to analyse.** This entails examination of a selection of mobile features; feature selection is an important part of any experiment. The next section addresses the question of **how to analyse.** Analysis methods are categorized into three groups: static, dynamic, and hybrid. Each category is explored comprehensively by providing definitions and examining related research works. The next section addresses the question of **how to detect.** Malware detection methods are discussed, describing their benefits and disadvantages. The final section of this chapter relates to the question of **where to detect.** It discusses the point of detection, which is the location in which malware detection is used.

Chapter 3 is called DroidLab. It investigates different tools used in Android malware analysis and detection. The chapter has three sections. The first section concerns static analysis tools that inspect Android installation files and extract various components. The second section deals with dynamic analysis tools for analysing the behaviours of Android applications. The third section discusses the available tools used in machine learning approaches, while the fourth section discusses those used to measure the energy consumption of mobile applications.

Chapter 4 outlines the proposed malware analysis and detection system for Android devices. It discusses various parts of the system along with their functions. Process flow and data flow are discussed, using numerous diagrams. In addition, methods and services used in the system are explained.

Chapter 5 evaluates the proposed system by performing four different experiments. The first one relates to static analysis. It explores the use of Android Intent and shows that it is a rich and undervalued component for malware analysis. The results from Android Intent are presented and compared to those from Android permission, which is a well-known component in Android malware analysis. The second and third experiments are

related to dynamic analysis. They explain the rationale behind choosing network traffic as a selected dynamic feature. The second experiment chooses the best network-related features by using four feature selection algorithms. The results are presented and analysed at the end of this evaluation. The third experiment uses an advanced deep learning algorithm to detect malware. The fundamentals of such an algorithm are explained, along with the detection results. The final experiment serves the objective of this study by measuring the energy consumption of the proposed system. The results are then compared to similar systems.

Chapter 6 presents a prototype system that includes all the elements of the proposed framework. First, the development process is described, which includes the technical preparation of the prototype. Following this, the various parts of the system are illustrated in the form of screenshots.

Finally, Chapter 7 concludes this work by discussing its contributions, limitations, and offering suggestions for future work.

In addition, there are number of appendices included at the end of this study. They include a list of reviewed work from the literature, a list of malware families in the MalGenome data sample, and list of publications derived from this research work.

CHAPTER 2: MOBILE MALWARE EVOLUTION, CHARACTERISTICS AND DETECTION METHODS

Mobile malware has witnessed many changes since its first appearance. They include simple annoyance malware up to the most sophisticated. The objective of this chapter is first to walk through the development of mobile malware in order to establish a context for this study. Android architecture and its security features are also explained in detail. We then discuss and evaluate some of the most useful and salient research work, nominate available gaps in the literature, and clarify the problem addressed in this study.

2.1 Mobile Malware Evolution

The history of mobile malware goes back to 2004. A coder named Vallez developed a proof-of-concept malware known as Cabir for the Symbian operating system. Soon afterwards, malicious coders developed malware based on Cabir (TrendMicro, 2012). In the same year, attackers made use of Cabir code to develop Qdial, a malware that sends a short messaging service (SMS) to premium numbers. This caused users to receive unexpectedly expensive phone bills. Also in November of the same year, Skulls malware infected mobile devices. It altered files on devices, causing applications to stop functioning, replacing their icons with a skull and crossbones.

By 2005, mobile malware had begun to steal users' information. Pbstealer was a malware that collected the address books from devices and transmitted them to a nearby Bluetooth-enabled device. Considering that some entries in the address book may have contained usernames and passwords, such types of malware brought a new kind of danger to mobile devices (TrendMicro, 2012). At the time, malware tended to spread via Bluetooth, since devices were not equipped with Wi-Fi chips. In this context, another major development in malware was the use of multimedia messaging services (MMS) as a way of spreading

the malware. Commwarrior was one of the first malware to use this method (Adeel & Tokarchuk, 2011).

By 2009 the growth of mobile malware was steadily rising. In addition to the Symbian operating system, attackers developed malware in Java language. This was because of the introduction of a Java-based mobile operating system, which gave attackers more options for infecting a broader range of devices.

The introduction of two new mobile operating systems radically changed the spectrum of mobile malware in 2010. Gartner reported that the sale of mobile devices had increased by 72% compared to 2009 (Gartner, 2011). Attackers saw this steep increase as an opportunity to develop new malware based on the newly introduced operating systems, namely Google's Android and Apple's iOS. By 2011 it was reported that Android had obtained almost 50% of the worldwide market share of mobile operating systems (MashableAsia, 2011).

Attackers followed the same malicious behaviour as Symbian malware. DROIDSMS was the first malware for Android, and was first detected in August 2010. It sent SMS messages to premium numbers (TrendMicro, 2010a). However, the capabilities of mobile devices at that time offered new opportunities for attackers. In the same year, a modified version of DROIDSMS was detected as a disguised version of the Tap Snake game. It collected the GPS location of the victim's device and transmitted it to the attacker over the Hypertext Transfer Protocol (HTTP) connection (TrendMicro, 2010b).

Android malware growth sharply increased in the years following 2010. According to a report from F-Secure, Android accounted for 79% of malware in 2012, up from 66.7% in 2011 and from just 11.25% in 2010 (Amos et al., 2013). Additionally, Android malware

continued to become more sophisticated. Android.Obad is the most complex malware discovered to date; it was dubbed villain of the year 2013 (Kaspersky, 2013).

Kaspersky lab announced that they had discovered 2,961,727 malicious installation packages and 884,774 new malicious mobile programs in 2015, a threefold increase from the previous year. Figure 2.1 shows the geographical distribution of Android malware in 2015.

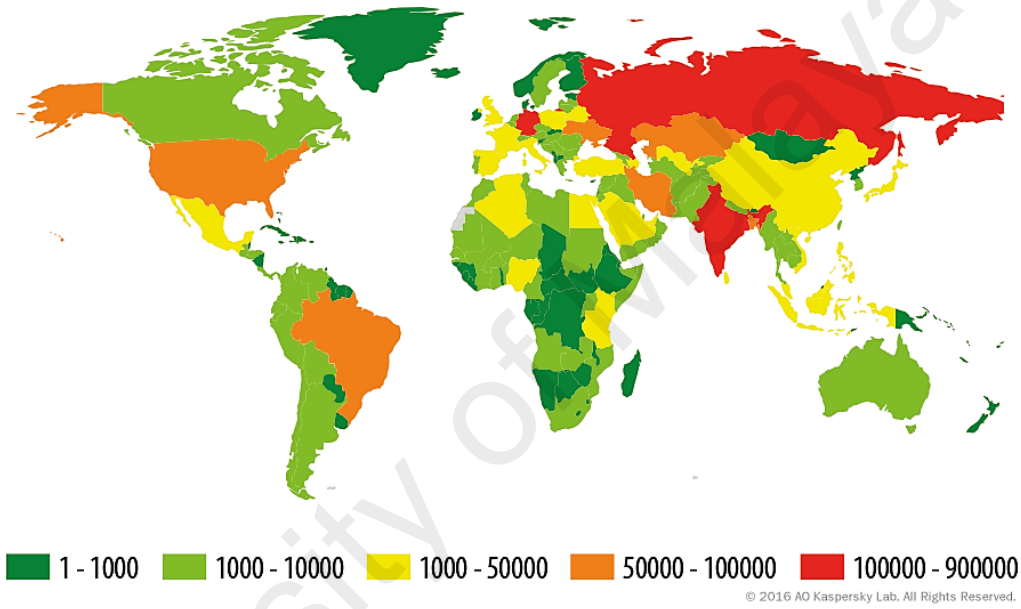


Figure 2.1. The geography of mobile malware by the number of attacked users in 2015 (Kaspersky, 2016a)

The 10 countries with the highest number of victims in 2015 are tabulated in Table 2.1. China is ranked first with 37%; this means that 37% of users of mobile security products in China encountered a mobile threat at least once during the year. The reason for this is that many unofficial application markets are popular, and users tend to download applications from such sources. Accordingly, attackers publish their malicious application in third-party markets, where security monitoring is not very rigorous.

Table 2.1. Top 10 countries by percentage of attacked users in 2015

Rank	Country	Attacked Users	Rank	Country	Attacked Users
1	China	37%	6	Vietnam	22%
2	Nigeria	37%	7	Iran	21%
3	Syria	26%	8	Russia	21%
4	Malaysia	24%	9	Indonesia	19%
5	Ivory Coast	23%	10	Ukraine	19%

The propagation strategy developed alongside malware itself. Prior to Android, attackers relied on SMS, MMS and Bluetooth to infect more devices. Following the introduction of Android, attackers tried to spread their malicious applications through Google Play. Android users use the official application market, known as Google Play, to download applications. However, some users choose to download applications from third-party markets, such as SlideME.

The propagation strategy gained popularity, as in March 2011 it was discovered that 50 applications inside Google Play were infected with DroidDream malware. This malware steals the IMEI and ISMI numbers of devices along with other personal information (AndroidPolice, 2011). Google introduced Bouncer in 2012 in response to rapidly growing Android malware inside Google Play. This is a security mechanism that vets applications before publishing to the market. Google announced that they check over six billion applications per day in order to prevent malicious applications from being published (Google, 2016). Despite such efforts, in early October 2015 Kaspersky came across several malware in the official Google Play market that stole victims' usernames and passwords. About a month later a new modification of the same malware was unearthed, which was also distributed via Google Play. Attackers published this malware 10 times on the official market under different names over a period of several months. The number of downloads for all versions was estimated at between 100,000 and 500,000 (Kaspersky, 2016a).

2.2 Android Operating System

This section describes Android architecture and examines the Android installation package. It sheds light on the foundations of the Android operating system. It also discusses available Android security mechanisms.

2.2.1 Android Operating System Architecture

Android is based on the Linux 2.6 kernel. The kernel is the first layer on top of the hardware that interacts with the device's hardware. Figure 2.2 shows the Android architecture.

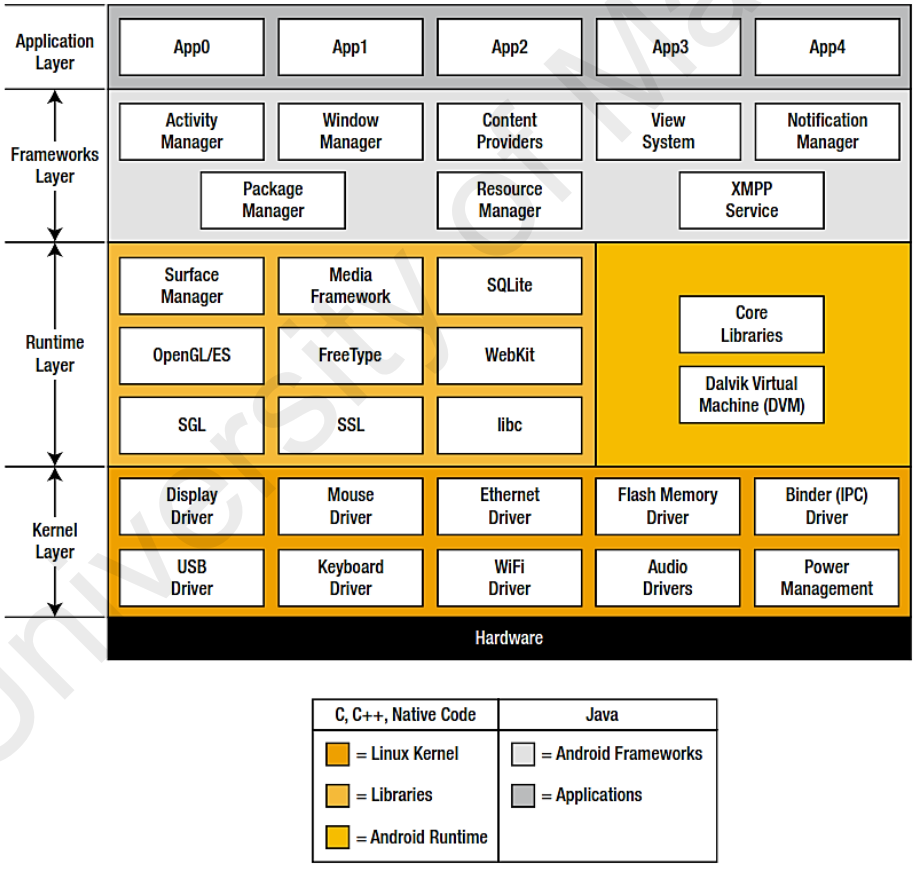


Figure 2.2. The Android Architecture (Gunasekera, 2012)

The kernel layer is responsible for directly interacting with hardware and performing different tasks such as display, USB, Wi-Fi, audio, etc. **The runtime layer** is comprised of library components written in C/C++ language. Android developers access libraries

through the Java application program interface (API) in order to use them in their applications.

Additionally, this layer consists of the Dalvik Virtual Machine (DVM), in which system and third-party applications are executed. The Dalvik was written by Dan Bornstein, who named it after a small village in Iceland. The Dalvik was written because mobile devices have limited resources (although memory and CPU power have increased over the years, battery limitations remain a challenge). It allows Android to run applications efficiently considering the limitations of the device.

Android applications are written in Java language that creates class and jar files. Upon compiling written applications, Java files are converted to Dalvik format and stored in DEX file used by the DVM to run applications. Figure 2.3 shows the conversion from Java to Dalvik format.

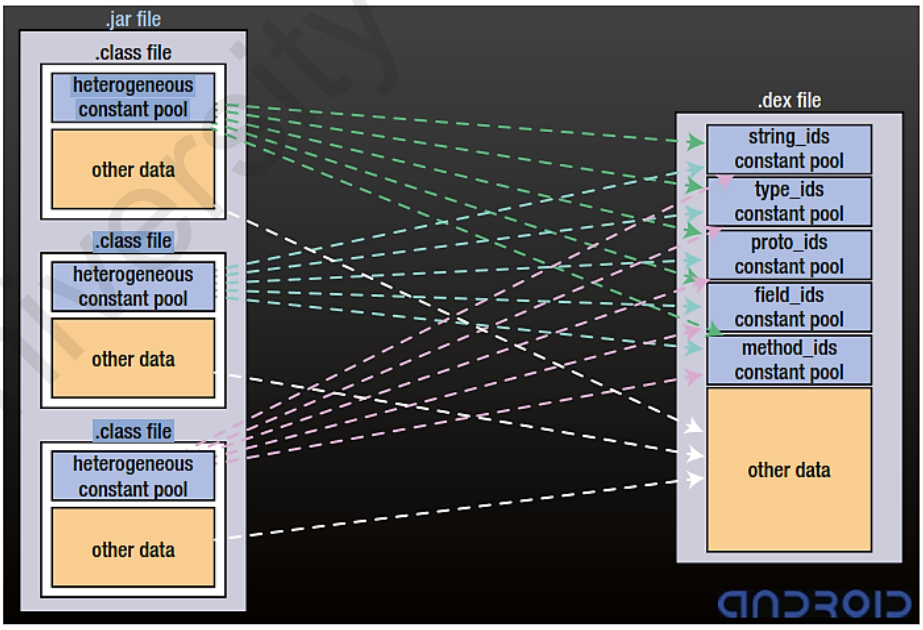


Figure 2.3. Conversion of Java to Dalvik Format (Gunasekera, 2012)

Noticeably, the constants in each class file are combined into a shared pool of constants, and other data sections are assembled into one section in the DEX file. Not only does this

conversion make applications run faster on devices, but it also reduces the size of the DEX file.

The framework layer consists of many APIs, giving developers access to building blocks of applications (e.g. buttons, text boxes, notification area, etc.). The APIs in runtime layer give developers access to fundamental actions that require interaction with the kernel layer and the hardware. However, APIs in the framework layer are used for many application components. Finally, **the application layer** is the layer that users interact with. The messaging applications, contacts, games, third-party applications are located in this Android layer, which is the layer closest to users, taking input to applications and providing output to users (Gunasekera, 2012).

2.2.2 Android Application Package Structure

As discussed earlier, Android applications are written in Java language and then compiled into a DEX file. This process is shown in Figure 2.4.

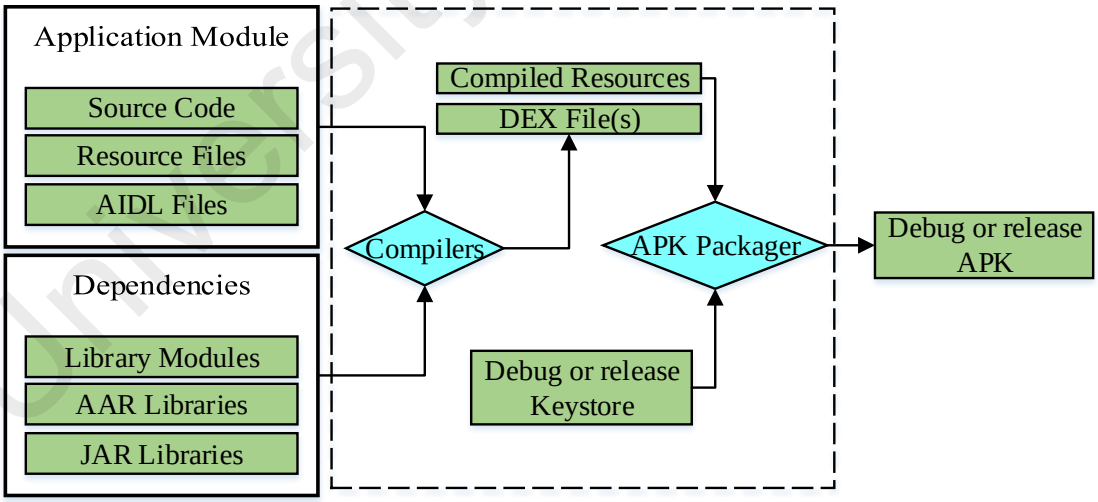


Figure 2.4. The Build Process of Android APK File[1]

The process of packaging an Android Application Package (APK) file starts with compiling the source code, resource files (pictures, icons, sound files etc.), and Android

[1] https://developer.android.com/studio/build/index.html

Interface Definition Language (AIDL) files, along with any dependencies that the code may have used (including libraries and JAR files). It is worth mentioning that AIDL allows developers to define the programming interface that both the client and service agree upon in order to communicate with each other using inter-process communication (IPC). The output of this compilation is a DEX file. The process could result in more than one DEX file. The total number of references that can be invoked by the code within a single DEX file is 65,536. Exceeding this number results in the creation of a second DEX file, which is why it is mentioned as DEX file(s) in Figure 2.4.

The next step is to prepare the debug or release the keystore. Android requires that all APKs are digitally signed with a certificate before they can be installed. A keystore is a binary file that contains one or more private keys. When debugging applications, developers need to sign their APK with a debug certificate; the final version of an application is signed with the release keystore. Lastly, APK packager uses the DEX file and the keystore to produce the APK file.

The generated APK file has many components (including a DEX file). It is used to install applications on Android devices. Part of the malware analysis and detection method is based on APK files. It is thus helpful to understand its structure. It is an archive file that can be opened with the WinZip program. The components of an APK file are as follows:

a) AndroidManifest.xml: An XML file holding meta information on an application, such as descriptions and security permissions. Prior to installation of an Android application, the application provides prospective users with a list of permissions that are available in the file.

b) Classes.dex: This contains the source code of an application written in Java and compiled for Android that the machine converts it to a special file format with a DEX extension.

c) Resources: This entails all the resources the application needs to run, such as pictures used in the application, the layout of the application, its appearance to a user, the use of a database, as well as data stored in the database.

2.2.3 Android Security Features

Since the Android operating system runs on top of the Linux 2.6 kernel, it inherits its security structure from Linux, and adds some modifications to suite mobile devices. In this section, many security components of Android are discussed in order to better understand current research.

Android applications run inside a virtual machine. They are unable to see other applications. The DVM was presented in Figure 2.2 as part of the runtime layer of the Android system. Figure 2.5 shows the concept of the DVM from a different perspective and in more detail.

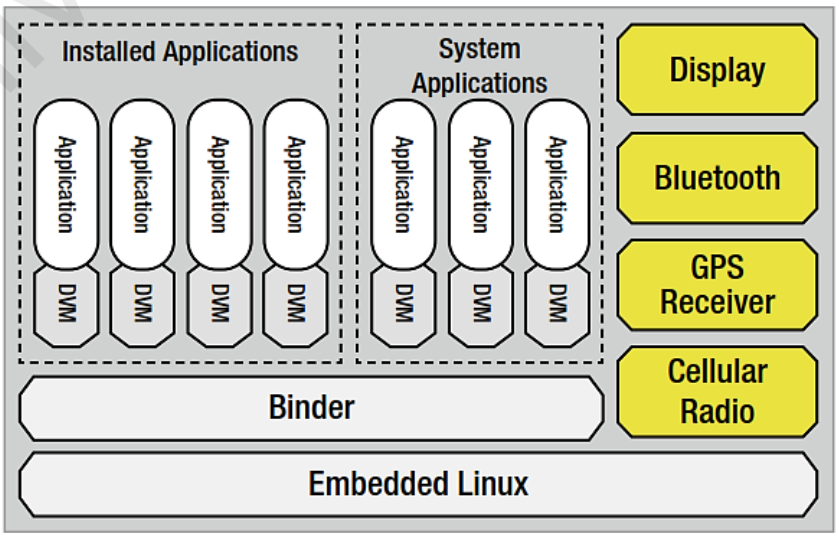


Figure 2.5. The Dalvik Virtual Machine (DVM) in Android Architecture (Gunasekera, 2012)

The Android applications (system or third-party applications) have their own virtual machine. Since starting a virtual machine from scratch is time consuming, resulting in delays in the functionality of applications, Android relies on a pre-loaded virtual machine. A process known as Zygote is responsible for starting up an application using a pre-loaded virtual machine, and initializing core library classes required by that application (Armando et al., 2012).

However, upon launching, each application has some very basic access to various system components. In case it should require additional resources, it requests permission for that resource. The Android permission is a security feature derived from Linux. The Android checks to see if an application has been granted proper permission before performing an activity (e.g. permission for using a camera, accessing a users' location, making a call) (Felt et al., 2011).

Intent is a complex messaging system in the Android platform, and is considered a security mechanism for hindering applications from gaining access to other applications or system functions directly (e.g. sending an SMS, making a phone call, opening a link in a browser, etc.). This is a way of controlling what applications can do once they are installed in Android (Aftab & Karim, 2014). Android permission and Android Intent work closely together to provide security. As an example, Android applications ask permission to make a phone call. They then use Intent to actually make the phone call. Therefore, Android checks to see if applications have specific permissions to use Intent.

2.3 Mobile Malware Characteristics

In this section, we discuss the various types of Android malware and their characteristics. We also discuss the type of malware that this work focuses on, which clarifies the target of this work.

Before categorizing mobile malware, a definition of mobile malware will be provided. Techopedia defines mobile malware as follows: “Mobile malware is malicious software that is specifically built to attack mobile phone or smartphone systems. These types of malware rely on exploits of particular operating systems (OS) and mobile phone software technology, and represent a significant portion of malware attacks in today’s computing world, where mobile phones are increasingly common” (Techopedia, 2016).

Webopedia defines mobile malware as “Malicious software ("malware") that is designed specifically to target a mobile device system, such as a tablet or smartphone to damage or disrupt the device. Most mobile malware is designed to disable a mobile device, allow a malicious user to remotely control the device or to steal personal information stored on the device” (Webopedia, 2016).

Based on the two mentioned definitions, we deal with malware that exploits mobile devices to steal personal information. There is a variety of attacks particular to Android, ranging from adware to the most sophisticated and dangerous kind. The purpose of adware is to advertise a product or a website; it is harmless but annoying. The most dangerous and sophisticated malware is capable of accessing personal data on the device as well as hijacking the mobile device itself. We have categorized mobile malware based on their behaviours and characteristics as follows.

2.3.1 Adware

Although some Android applications are free, they show advertisements while operating. Sometimes the advertising is aggressive and annoys users. Apart from pushing advertisements in devices without the user’s consent, they are able to change internet browser settings, showing icons on the home section of devices, and in minor cases collecting user information.

Android Dowgin is an example of an adware that installs itself on an Android device as a bundle with other applications. It then displays advertisements in the notification area of the device and is not easily removed. It is estimated that between 10,000 to 50,000 users are infected with this adware (AVG.ThreatLabs, 2013). It has been spreading since July 2013 and continues to proliferate (Eset, 2013). The alarming issue is that, as of December 2013, some of the more prominent antivirus software such as Symantec, TrendMicro, and McAfee were not able to detect it (Virustotal, 2013).

2.3.2 Trojan and Bots

Trojan is a seemingly clean application containing a malicious code. Once it is installed onto mobile devices, the malicious part is activated. It then performs various malicious activities including corrupting the operating system, collecting personal information, gaining root access, and sending user information to attackers.

A botnet comprises a network of infected devices scattered geographically that is used to attack other systems for malicious purposes. The botnet is under the command of a hacker. The hacker is able to command the bots, also known as zombies, to attack a specific victim. An infected device communicates with the hacker through a rendezvous point called the command and control (C&C) server.

The reason for putting Trojan and bots in the same category is the aggressive nature of Android malware. Trojan and bots share the same characteristics. They start by representing themselves as a normal, clean application. Upon installation, however, they show their true nature by performing malicious activity. This trait categorizes them as Trojan. Following this, they contact their master through the C&C server and report their activity or receive commands to perform further damage to the device, which defines them as bots.

Security analysts discovered, for instance, an infected version of the Angry Birds Space application in April 2012. It functions like a normal application without suspicious symptoms. However, it uses a software trick known as GingerBreak to acquire root access that allows it to do tasks outside of its privilege. It secretly downloads malicious codes from a server and opens a back door for attackers, upon which the device eventually joins the botnet (Sophos, 2013). Another example is the ZeroAccess botnet that adds approximately 100,000 new infections weekly. It receives a considerable sum of money from its clients each week in order to generate new associated infections. It had an 88.65% share of the botnet dominance in 2013 (Fortinet, 2014).

Xbot was discovered in February 2016. This is a cocktail of different types of Android malware. It starts by infecting a device as a Trojan. It then collects banking and credit card information as the users enter their credentials. It acts as a bot by contacting the C&C server and passing the collected information on to the attacker. The attacker has the ability to lock and encrypt files on the device and SD card, and then demands 100 USD ransom from the victim. Researchers have unearthed 22 applications infected with Xbot, some of which target Australian banks (C. Zheng et al., 2016).

The Android attackers sometimes have financial encouragements and have recently also become more aggressive (Symantec, 2014a). Upon installation, some applications send expensive SMS messages to premium numbers without the users' knowledge, and this reflects itself in the user's bills. Such applications have been on the rise for years. A report published in 2013 shows that some attackers earn up to 12,000 USD per month via such malware (The.Register, 2013). Based on a report by Sophos, a malicious version of the popular Angry Bird game secretly sends premium SMS for 15 GBP per message. Each time the user starts the application, it sends a premium SMS. It is estimated that 1,391 devices are infected with this malware, and it has been estimated that developers of this

malicious application have earned 27,850 GBP through sending SMS messages to premium numbers (Sophos, 2012).

Recently attackers have adopted a new approach towards infecting mobile devices. Thus far, attackers had been dependant on tempting users to download their malicious applications, after which the application performs malicious activities without the users' knowledge. It has been observed that PCs have been used as a conduit for Android devices, which are called hybrid threats (Symantec, 2014a). Trojan Droidpak uses hybrid threats to infect mobile devices. It first gains access to a personal computer and, based on that, a malicious APK file downloads itself. When the user connects an Android device to the computer, the malicious file attempts to install itself on the device. After successful installation it attempts to convince the user to download and install an infected version of a Korean banking application (Symantec, 2014a).

Based on a report from Kaspersky, Trojan for mobile devices constitutes 49% of Android malware (Kaspersky, 2012). Additionally, in terms of malware dangerousness, trojans and bots are more dangerous than other categories of malware. Such families include Obad, Shedun, Godless, Hummingbad, and Gunpoder (Milin-Ashmore, 2016). We therefore focus on the analysis and detection of this category in this study, which covers the majority of the Android malware spectrum.

2.3.3 Ransomware

This type of malicious application is new to the mobile malware ecosystem. Ransomware takes mobile devices hostage and demands ransom. Android.Simplocker was the first Android ransomware, and was detected in 2013. Symantec found a fake security application called Android Defender that encrypts files, locks the device, and renders it useless. It demands ransom in order to unlock the device (Symantec, 2014b). To increase

the victim's fear, this variant of malware uses the front camera to display the victim's photo (ESET, 2016).

Lock-screen ransomware and crypto-ransomware are two categories of this type of malware. The lock-screen method hijacks resources and locks the device, hindering the user from using it. The crypto-ransomware hijacks files by using encryption. In both methods the attacker demands ransom in order to unlock or decrypt the device (ESET, 2016).

MacAfee reported an increase of 26% in the amount of ransomware in the last quarter of 2015 (MacAfee, 2016). This type of malware is new; it has been estimated to increase over time and spread to Android-based smartwatches. Smartwatches introduced new types of smart devices that connect to mobile devices. They offer new opportunities for attackers to spread their malicious applications (Symantec, 2015).

2.4 Mobile Malware Analysis and Detection Methods

The previous sections of this chapter formed a basis for reviewing Android malware analysis and detection methods. The scope of this study demands that we examine the current literature from four different perspectives corresponding to each section. They are as follows: **A)** features to analyse (Section 2.4.1), **B)** how to analyse the selected features (Section 2.4.2), **C)** how to detect mobile malware using the analysed features (Section 2.4.3), and **D)** where to detect mobile malware (Section 2.4.4). The full list of reviewed works is available in Appendix A.

2.4.1 Feature Selection in Mobile Malware Detection

Numerous studies have developed methods to thwart attacks on mobile devices. In order to develop an effective detection system, a subset of features from hundreds of available features must be chosen. This section investigates the different features available for

analysis. Android applications consist of various elements such as permissions, Java code, certification, the behaviour of the application on the device, and its behaviour on the network. Selecting the most useful subset of features from a massive number of available features changes the result of the whole experiment (Guyon & Elisseeff, 2003). Some of the benefits of feature selection are as follows:

a) Feature selection makes it possible to reduce the dimensionality of the datasets, because with less data it is possible to easily visualize the trend in data (Liu & Motoda, 2007).

b) Datasets involve analysing vast amounts of data; therefore, reducing them to a useful subset not only saves the time and cost of experiments, but also minimises the time required for real-world implementation (Liu & Motoda, 2007). Furthermore, selecting a useful subset of the features considerably reduces the runtime of the machine learning algorithms during the training phase.

c) Feature selection removes noisy and irrelevant data from datasets, leading to more accurate results from machine learning algorithms (Jensen & Shen, 2008).

We conducted two experiments in order to examine the effect of features on results. We collected the network traffic of over 800 Android applications, including normal and malicious, from the MalGenome (Yajin & Xuxian, 2012) data sample. The dataset consists of ten network traffic features, out of which we selected five features for each experiment. The dataset comprises of 504,148 records. The K-nearest neighbour classifier with three neighbours was used. Table 2.2 shows the results of the experiments.

Table 2.2. Results of the Experiments

	Experiment 1	Experiment 2
Features	frame.len, frame.number, frame.time_delta, frame.time_relative, tcp.srcport	tcp.dstport, tcp.window_size value, tcp.seq, ip.src, ip.dst
True Positive Rate (TPR)	98.63%	99.98%
False Positive Rate (FPR)	1.37%	0.02%

As Table 2.2 illustrates, different features yield different results, despite the fact that the data collection process and the used classifier are the same for both experiments. Thus, the effect of feature selection is conspicuous. In addition, selection of the most useful features is an important and challenging task.

We studied 100 of the most salient related research works with respect to feature selection in mobile malware detection. We categorize available features into four groups, namely static features, dynamic features, hybrid features, and application metadata.

2.4.1.1 Static Features

Static features include features available in the APK file such as Androidmanifest.xml files and Java code files. Out of 100 papers reviewed, 45 papers used static features to conduct their experiments. Among static features, researchers used permission in 36% of the papers, more than other static features. Selection of Java code came second in 29% of the papers. The following sections discuss static features in details.

(a) ***Android Permission***

We know that the Android operating system has a Linux core, from which it inherits important parts of the Linux security architecture. Prior to installation of an application, the Android provides a list of requested permissions to the user. Upon the permissions

being granted, the application installs itself on the device. There are 130 official Android permissions (Moonsamy et al., 2013b). Google categorizes them into four groups, namely, normal, dangerous, signature, and signatureOrSystem (Google, 2014). Researchers take different approaches in analysing Android permissions. Some use permissions to evaluate applications and rank them based on possible risks (Au et al., 2012; Grace, Zhou, Zhang, et al., 2012; Pandita et al., 2013; Peng et al., 2012). Numerous studies simply extract permissions and utilize machine learning to detect malicious applications (Aung & Zaw, 2013; Samra et al., 2013; Borja Sanz, Santos, Laorden, Ugarte-Pedrero, Bringas, et al., 2013; Suleiman Y Yerima et al., 2014), to name a few.

Researchers argue that merely analysing the requested permissions is not sufficient for detecting malicious applications (C. Y. Huang et al., 2013; Moonsamy et al., 2013b). They analyse the used permissions in addition to the requested permissions in order to detect malware. Malicious applications tend to request more permissions than they need, which is a way of identifying them. AppGuard has gone one step further and has extended Android's permission system to alleviate current vulnerabilities (Backes et al., 2013). The approach is claimed to be a practical extension for the Android permission system, as it is possible to use it on devices without any modification or root access.

Why is Android permission the most used static feature? As mentioned earlier, the Android operating system has Linux architecture. Permission is the first barrier to attackers. Even though the Java code contains malicious code, some of API calls in the code need permission to be invoked (D.-J. Wu et al., 2012b). Permission-protected API calls are part of the security features of the Android operating system. For example, before sending a message or accessing the camera, Android checks if the application has permission to do so (Felt et al., 2011). Based on that scenario, researchers focus on permission features to detect malware based on the demanded permissions.

(b) ***Android Java Code***

Developers write the Java code, which is the main part of Android application files, and subsequently compile them to a special format called Dalvik that is proprietary to the Android operating system. Researchers have used various analysis approaches on Java code. Some researchers use API calls to detect malware (Deshotels et al., 2014a; Grace, Zhou, Wang, et al., 2012; V. Rastogi et al., 2014; S. Y. Yerima et al., 2013; M. Zheng et al., 2013b). Every Android application needs to have API calls to interact with the device. As an example, there are API calls to the telephony manager of the operating system to retrieve phone ID and subscriber ID. API calls in a method are sequential. Researchers consider such a sequence as a signature that is unique to that application. However, changing the sequence of the API calls is a strategy called code obfuscation that is used by attackers to bypass the detection process. Analysing control flow of the Java code is another approach adopted by researchers (Crussell et al., 2012; Suarez-Tangil et al., 2014; Xu et al., 2013). Attackers can change the sequence of API calls or rename the calls to evade the detection system. However, the flow of the Java code does not change and researchers use it to develop stronger detection systems.

(c) ***Other Static Features***

Besides permissions and Java code, some researchers analyse several other static features as follows.

1) **Intent**: As discussed in Section 2.2.3, Intent is one of the security features in Android. Application developers use Intent in Java code and XML file. It is used in Java code to perform actions. Moreover, it is one of the elements described in Androidmanifest.xml file. It is declaration of capability to perform an action. For instance, when an application is able to open a text file, it declares it in the XML file. This way, the Android knows what application to use to open a text file.

Researchers have been using Intent for malware detection, since attackers command malware to collect private data, which requires the presence of intentions in the Android Intent.

In DroidMat, various features from an Android file including intents are extracted and analysed (D.-J. Wu et al., 2012b). The authors utilized several machine learning algorithms such as k-means, K-Nearest Neighbours, and naïve bayes, to develop malware detection systems. The evaluation of the DroidMat showed an improvement over similar systems in that time.

The A3 system was published that considers several features including intents in the Android installation file (Luoshi et al., 2013). It then constructs a call graph that represents the flow of the Java code execution. Afterwards, it uses A^* algorithm to determine the shortest path that subsequently shows the behaviour of malware.

DREBIN presents a broad static analysis (Arp et al., 2014). The approach collects static features of Android installation files including intents. The authors used a support vector machine (SVM) for detection purposes. The results of the experiment showed that DREBIN detected 94% of the malware, with a low rate of false alarms.

So far, research has focused on analysing Intent in the XML file. Therefore, the Intent in Java code is an undervalued feature. As a result, we choose it for our experiments (more detail on the rationale behind in Section 5.2).

2) **Network address**: Attackers instruct malware to contact them and report their status or to send users' personal data. To do so, attackers embed the address of the C&C server in the malicious code of the malware. Researchers look for the network or the IP address of the C&C server in the code of the Android installation file. Luoshi et al. and Arp et al. incorporated the network address as one of the static features in their systems

(Arp et al., 2014; Luoshi et al., 2013). However, attackers started to encrypt the address of C&C servers to evade detection methods, for example in the DroidKungFu malware family.

3) **Strings**: Sanz et al. stated that one of the widely used techniques in classic malware detection is analysing strings available in the file (Borja Sanz et al., 2014). They applied the same technique for Android malware by extracting every printable string in the Android file, such as menus in applications or server addresses with which the application connects. The authors used the Vector Space Model (VSM) (Baeza-Yates et al., 1999) to represent the strings as vectors in multidimensional space. Afterwards, the authors used distance measures, such as Manhattan distance, Euclidean distance and Cosine similarity to calculate the anomaly of the data. The authors evaluated the results with 666 samples of Android applications. They achieved an accuracy of 83.51% and a true positive rate (TPR) of 94% in the experiments.

4) **Hardware components**: DREBIN used hardware components as a static feature (Arp et al., 2014). As part of Androidmanifest.xml file, applications request combinations of hardware that they need in order to function, for example the camera or GPS. Combinations of requested hardware imply harmfulness of the application, for example, 3G and GPS access implies a malware that reports the location of users to attackers.

2.4.1.2 Dynamic Features

We define dynamic features as behaviour of applications in interaction with operating system or network connectivity. There are two main types of dynamic features: system calls and network traffic. Every application demands resources and services from the operating system by issuing system calls, such as read, write and open.

Network traffic is another dynamic feature used by researchers. Applications tend to connect to a network to send and receive data, receive updates, or maliciously leak

personal data to attackers. Monitoring network traffic of mobile devices is a way of catching a culprit in the act.

Based on our analysis, 42 out of 100 papers used dynamic features. Twenty-two papers used system calls as their dynamic feature and 10 papers used network traffic. The remaining 10 papers selected other dynamic features, such as system components or user interaction.

(a) ***Android System Call***

There are more than 250 system calls in a Linux kernel, which includes Android (Burguera et al., 2011). Analysing system calls leads to anomaly detection in the application's behaviour (Feizollah et al., 2013). Applications use system calls to perform specific tasks such as read, write and open, since they cannot directly interact with the Android operating system. Upon issuing a system call in user mode, the Android operating system switches to kernel mode to perform the required task. System call is the most selected feature among the dynamic features, constituting more than half of the reviewed papers. Research works such as (Burguera et al., 2011; Khune & Thangakumar, 2012; Su et al., 2012; L. K. Yan & Yin, 2012; Zhao et al., 2011b) captured and analysed system calls to detect malicious applications.

(b) ***Android Network Traffic***

The majority of applications (normal or malicious) require network connectivity. MalGenome authors stated that 93% of their collected Android malware samples need a network connection in order to connect to attackers (Yajin & Xuxian, 2012). Additionally, a research work was published in 2012 in which they analysed permissions of Android files (Sarma et al., 2012). They examined over 150,000 applications and found that 68.50% of normal applications require network access, while 93.38% of malicious applications do.

Similarly, Sanz et al. analysed permissions of 2,000 applications (Borja Sanz, Santos, Laorden, Ugarte-Pedrero, Bringas, et al., 2013). Based on their analysis, over 93% of malicious applications requested network connectivity. It is evident that the majority of applications request network access, particularly the malicious ones. Therefore, it behoves researchers to focus on analysing network traffic for effective Android malware detection.

Despite the effectiveness of the network traffic feature in mobile malware detection, it has not attracted researchers' attention as much as the other dynamic features. Utilizing network traffic imposes the challenge of dealing with massive numbers of network records, possibly as many as a million, in the dataset. Furthermore, analysing collected network traffic requires profound understanding of network architecture. Thus, we select this feature for our experiment (details in Section 5.3).

(c) ***Other Dynamic Features***

In addition to system calls and network traffic, researchers have been using other dynamic features. They are as follows:

1) **System components**: Mobile devices have similar components to PCs, such as CPU and memory. Some researchers investigated detection of Android malware by using system components. In MADAM, the authors analysed CPU usage, free memory, and running processes of mobile devices that are considered the kernel level of the operating system (Dini et al., 2012). In addition, it examined user/application level features, such as Bluetooth and Wi-Fi status of the device. The collected data were used to train the K-Nearest Neighbours algorithm.

STREAM was introduced in 2013 for the Android operating system (Amos et al., 2013). It collects data regarding system components like cpuUser, cpuIdle, cpuSystem,

cpuOther, memActive, and memMapped. It subsequently uses machine learning algorithms to train the system in order to detect Android malware. Other works that also use system components as dynamic features are (Hoffmann et al., 2013; Hyo-Sik & Mi-Jung, 2013).

2) **User interaction**: Users are potential victims of malicious applications. Analysing users' interaction with applications is one of the possible solutions in malware detection. PuppetDroid captures users' interaction with applications (Gianazza et al., 2014). The authors consider features such as pushing a button and navigating through pages as user interaction, and evaluated the system with 15 Android applications. The goal is that after capturing user interactions related to a malware, the system looks for similar user interaction to detect malicious applications. Dynodroid is another system that was developed based on user interaction analysis (Machiry et al., 2013). It collects users' activities, such as tapping the screen, long pressing and dragging. The evaluation of Dynodroid involves analysing 50 Android applications. The results found bugs in Android applications.

2.4.1.3 Hybrid Features

Hybrid features are the most comprehensive; they consist of static and dynamic features features. They involve vetting Android application installation files as well as analysing the behaviour of the application in runtime. Blasing et al. developed AASandbox, which analyses static and dynamic features (Blasing et al., 2010). It extracts permissions and Java code from the APK file and uses them as static features. It then installs the application, logs system calls, and uses it as dynamic feature. Authors of ProfileDroid examined Androidmanifest.xml and Java code as static features (Wei et al., 2012). They chose user interaction, system calls and network traffic as dynamic features. Similar

works that chose hybrid features include (Eder et al., 2013; Spreitzenbarth et al., 2013; Xu et al., 2013; Zhou et al., 2013).

2.4.1.4 Android Applications Metadata

A few researchers opted to utilize Android applications' metadata for malware detection. Metadata are defined as the information users see prior to download and installation of applications, such as the description, the requested permissions, their rating, and information regarding the developer. The applications' metadata cannot be categorized as static or dynamic features as they have nothing to do with the applications themselves.

WHYPER collected permissions requested by applications in the market and used Natural Language Processing (NLP) to look for sentences that justify the need for the requested permissions (Pandita et al., 2013). It achieved 82.8% precision for three permissions (address book, calendar and record audio) that protect sensitive and personal data.

Similarly, Teufl et al. used a sophisticated knowledge discovery process and lean statistical methods to analyse the metadata gathered from Google Play (Teufl et al., 2013). The authors argue that metadata analysis should complement static or dynamic analysis. They collected data including the last time modified, category, price, description, permissions, rating, and number of downloads. The authors mentioned that the following data could also be used as metadata: creator ID, contact email, contact website, promotional video, number of screenshots, promo texts, recent changes, ID, package name, installation size, version, application type, ratings count, and application title. The approach also used machine learning algorithms. Definitions of some of the aforementioned metadata are shown below.

a) Last time modified: Applications in Google Play go through changes and updates. The date of last modification is a metadata.

b) Category: Google Play categorizes applications based on their types, such as games, applications and book. Each game type is further subcategorized as action, adventure, arcade, and board.

c) Description: Developers provide a brief description to describe the main functionality of their applications.

d) Permissions: Upon opting to install an application, it prompts the user with a list of permissions that the application requires to function properly.

e) Rating: Users rate applications based on their experience with it. It is helpful for new users to decide whether to download the application.

f) Creator ID: Every developer has an ID in Google Play. They use their ID to publish applications. When detecting a malicious application, Google is able to identify the developer and terminate their ID.

2.4.2 Malware Analysis

This subsection is dedicated to discussing analysis methods. Malware analysis is the process of analysing a sample of a malicious application or a malware family in order to find a pattern and trait. Such traits are then used for detection methods. There are three types of Android malware analysis: static, dynamic, and hybrid. For each type we intend to examine the current research works, and point out their weaknesses as well as their strengths.

2.4.2.1 Static Analysis

Static analysis is the process of dissection and examination of an Android installation file known as APK to detect suspicious applications. For instance, Huang et al. conducted a study in which the requested and required permissions were inspected to detect malicious applications (C. Y. Huang et al., 2013). The requested permissions are presented to users upon installation, whereas required permissions are those that are actually used in the

applications' code. They used four algorithms including AdaBoost, Naïve Bayes, Decision Tree, and SVM to evaluate the performance. The evaluation shows that their system detects 81% of malicious applications.

Seo et al. took a different approach by proposing a system to detect mobile threats to homeland security via static analysis (Seo et al., 2014). They developed a tool called DroidAnalyzer that inspects applications to detect potential threats by looking for usage of risky APIs and keywords in the Java code such as IMEI leakage, phone number leakage, su command, reboot command, etc. The DroidAnalyzer was evaluated by analysing applications in various categories such as banking, flight booking and tracking, and home and office monitoring applications.

Chen et al. used a NiCad clone detector in their experiment. It is a method to detect Near-Miss Intentional Clones (J. Chen et al., 2015). It takes a source directory as input, and finds classes of clones in the provided code. This process is implemented for one malware; the clone classes are used as a signature to find similar code in other source codes (Cordy & Roy, 2011). Their results show that this method is able to detect 95% of previously known malware in their dataset. However, it is not useful for a new variant of malware, as this method relies on a signature.

APPraiser is a system that differentiates between malicious and legitimate versions of similar applications (Ishii et al., 2016). It first extracts similar applications using the appearance analysis. It then extracts relatives, using several intrinsic fingerprints such as developer identities and application package names. Finally, it classifies clones using the code difference analysis and antivirus checkers.

However, the problem with static analysis is that some malware families such as DroidKungFuUpdate stealthily download malicious codes (Yajin & Xuxian, 2012),

which is known as dynamically-loaded code method. Thus, the malicious code is undetectable via static analysis. Similarly, permission-based analysis is less effective regarding malware such as Basebridge, which hides an updated version within the original application, and as a result slips into a mobile device without the user's knowledge and bypasses the permission system (C. Y. Huang et al., 2013). Static analysis is simple to implement, but it produces less information, thus limiting the extraction of possible features from malware activities. In addition, attackers use various methods such as code obfuscation to evade detection through static analysis. Code obfuscation is a method used to bypass static analysis, and it is defined as the act of changing the code so that it is difficult to understand and disassemble, but performs as the original code (Ishii et al., 2016).

Java reflection is another method used by attackers to evade detection. It is defined as modifying or examining the run time behaviour of a class. Reflection for Android apps can also be used to access all of an API library's hidden and private classes, methods, and fields. Android malware such as Android.Obad and FakeInstaller (F. Ruiz, 2012) call their methods indirectly through reflection, and the real method name is kept encrypted. Moreover, the name of the target method is unknown prior to execution of the applications. Thus, by converting any method call to a reflective call with the same function, it becomes difficult for static analysis to discover exactly which method was called.

2.4.2.2 Dynamic Analysis

Dynamic analysis fixes the problem of obfuscation in static analysis by identifying malware based on their behaviour. It is done by running applications on mobile devices or emulators, and observing their behaviours and interactions with the Android operating system. For one, Crowdroid collects the device's kernel system calls and sends them to a

remote server for processing (Burguera et al., 2011). The collected data are classified using K-Means algorithm. Self-written malware used to evaluate Crowdroid achieved 100% detection rate. Additionally, two real malware families used or evaluation achieved 85% and 100% respectively. By using system call as one of the features, malware can be detected based on similar behaviours and patterns. However, the evaluation method is not realistic, as it only used self-implemented malware and two real malware families. Furthermore, Crowdroid needs a constant network connection to send the collected data to the server for processing, which consumes bandwidth, and poses a challenge in case the connectivity is lost.

Andromaly monitored different system values such as CPU consumption, number of network packages, number of running processes, and battery level (Shabtai et al., 2012). The framework adopted feature selection methods such as chi-square, fisher score and information gain to enhance the detection accuracy. It then identified the best classification method out of six classifiers, namely Decision Tree, Naïve Bayes, Bayesian Network, K-Means, histogram, and logistic regression using the Andromaly framework. As a result, Andromaly managed to achieve a 99.9% accuracy rate with the Decision Tree classifier and Information Gain as feature selection method. Although this framework achieved great accuracy, the authors used self-written malware to evaluate their system, which could have produced unrealistic results.

The MADAM was introduced that is a multi-level detector prototype based on dynamic analysis by combining two system call levels, kernel and user level (Dini et al., 2012). With twelve system calls as the main features together with the K-Nearest Neighbours (KNN) classifier, they successfully obtained a 93% accuracy rate for ten malware. Although this approach is promising, it is incapable of detecting malware that avoids the

system call with root permission, for example SMS malware that is invisible in the kernel level (PandaSecurity, 2011).

Su et al. presented a smartphone dual defence protection framework, which has two phases, a verification service and a network monitoring tool. The first, the verification service phase, applies system call statistics to differentiate between malicious and normal codes in an application. The second phase monitors any possible malicious codes identified in the first phase. The two simulated classifiers to evaluate the proposed framework are Decision Tree and Random Forest. The former classifier achieved 96.67% and the latter 91% detection accuracy (Su et al., 2012).

Although dynamic analysis rectifies weaknesses of static analysis, it has code coverage problem. While running applications, there is no guarantee that the execution path in Java code stimulates and triggers malicious behaviour of malware, which is defined as code coverage. Some of the recent research works began to address code coverage (Gianazza et al., 2014; Ho et al., 2014), however, they were unable to fully solve the problem.

2.4.2.3 Hybrid Analysis

Hybrid analysis is the optimum approach for malware analysis as it uses both static and dynamic analyses. Hybrid solutions could therefore combine static and dynamic analyses so that their added strengths mitigate both weaknesses.

For example, Zhou et al. proposed DroidRanger that uses hybrid analysis. It uses static analysis to extract permissions, and matches applications' permission-based footprint with malware-specific footprint signatures (Y. Zhou et al., 2012). The researchers also proposed a heuristics-based filtering scheme that inspects applications for suspicious behaviour such as dynamically loaded code. The suspicious applications are observed using dynamic analysis to confirm whether they are malicious or not. In case malware is

detected, the DroidRanger generates its signature and adds it to the database. The authors evaluated DroidRanger by downloading applications from five different Android markets, and used their system to detect malware. The results show that they detected 171 malicious applications and two zero-day malware.

Mobile-Sandbox also combines static and dynamic analyses (Spreitzenbarth et al., 2013). For static analysis, it first analyses the permissions requested by an application. Then, it converts the application's Dalvik bytecode to smali code using baksmali5 and looks for dangerous methods, statically coded URL strings, and calls to encryption libraries. Information on timers and broadcasts as event triggers are also collected in order to improve code coverage during its dynamic analysis stage. In its dynamic analysis, Mobile-Sandbox logs runtime information at the three following levels: (i) Dalvik level monitoring using TaintDroid and a customized version of DroidBox; (ii) native code monitoring using ltrace; and (iii) network traffic monitoring. External events, such as incoming calls or SMS messages are simulated to trigger potentially malicious behaviour.

The publication of TaintDroid brought new a perspective to the research community by paying attention to a privacy leakage in the Android system and Android malware (Enck et al., 2010). AppSealer performs static taint tracking on an Android application and then follows the app along the respective propagation paths to monitor actual leaks at runtime, effectively ruling out false positives introduced by the static analysis (M. Zhang & Yin, 2014). It then fixes component-hijacking vulnerabilities at runtime if sensitive data reach a sink. This approach, however, cannot find leaks missed by the static analysis and thus inherits the problem of reflective method calls.

The Harvester tool can reduce obfuscation generated by encrypted strings and reflective methods with its hybrid methods (Rasthofer et al., 2015). It first uses static analysis by pinpointing sensitive variables in the code, and uses a program slicing method to separate

parts of the code involved in calculating the variable of interest. Using dynamic analysis, it executes the sliced piece of code and monitors its behaviour. The authors of Harvester claim that it usually extracts target phone number, body of SMS messages, decryption keys, or URL that are called inside the applications. They also mention that it is rigid against code obfuscation and dynamically loaded code.

AppDoctor also follows a similar approach by slicing Android applications to find user interactions that lead to application crashes (Hu et al., 2014). Although Harvester's hybrid slice-and-run principle is similar to AppDoctor, AppDoctor executes the complete derived user interface actions, while Harvester's slices only contain code contributing to the concrete value of interest.

Andro-Dumpsys uses hybrid analysis for malware detection (Jang et al., 2016). Andro-Dumpsys is based on similarity matching of malware creator-centric and malware-centric information. Andro-Dumpsys detects and classifies malware samples into similar behaviour groups by exploiting their footprints, which are equivalent to unique behaviour characteristics. The client application sends the package name or APK to the server. The server extracts memory dump, serial number of certificate, suspicious API sequence, permission distribution, intent, system command, and existence of forged files. It then uses machine learning for detection. The experimental results demonstrate that Andro-Dumpsys is scalable and performs well in detecting malware and classifying malware families with low false positives and false negatives, and is capable of finding zero-day threats.

Therefore, due to the comprehensiveness of hybrid analysis, we choose to follow this approach. It combines static and dynamic analysis to overcome their respective weaknesses, which results in a robust analysis method.

2.4.3 Mobile Malware Detection

This subsection discusses types of detection methods. After analysing malware families, their characteristics and behaviours are used for detection purpose. There are misuse-based detection and anomaly-based detection. For each type, we investigate several related research works and analyse them in terms of their weaknesses and strengths.

2.4.3.1 Misuse-based Detection

With the aim of confronting malware, mobile devices have adopted traditional approaches such as antivirus in PCs. The misuse-based methods are also known as signature-based methods, and are mainly used by antivirus software that relies on detecting malware based on unique signatures. This tactic is not as efficient (Sohr et al., 2011) against mobile device malware, as it requires continuous signature database updating, and mobile malware is constantly modified to circumvent various detection methods. For instance, the first version of DroidKungFu surfaced in June 2011, and seemed one of the most sophisticated kinds of malware at that time. Shortly afterwards, the second and third versions appeared in July and August. The fourth version was detected in October and the fifth soon after that. The variants tend to utilize assorted encryption keys to protect themselves. Such malware adaptation indicates hackers' insistent attempts to bypass detection, as (Yajin & Xuxian, 2012) demonstrated that traditional antivirus software is able to detect malware up to 79.6%.

The signature-based approach can be further categorized into behaviour-based signature and static-based signature.

(a) ***Behaviour-based signature***

AntiMalDroid generates a behaviour signature by running applications and monitoring their behaviour (Zhao et al., 2011b). The authors define behaviour as intent issued and accessing system resources. AntiMalDroid creates a signature database from the analysed

applications. In order to categorize an application as normal or malicious, AntiMalDroid runs the application and generates its behaviour signature. It then compares the signature to the database to check whether it matches any known signature. The evaluation included 100 normal applications and 2 types of malware, and was tested by 200 applications. The results show a detection rate between 90% and 93%.

SimBehavior is a similar system that is based on behaviour signature (H. Lu et al., 2014). The authors argue that the commonly used system call dependency method is too complex for mobile devices, and propose a lightweight method based on a resource differentiation scheme, which is abbreviated as DiffHandle. Malware makes sequences of system calls by using obfuscation technologies. Similarly, it prevents detection methods from gaining common behaviours in samples of the malware family. Thus, the authors presented an Iterative Sequence Alignment (ISA) method to defeat disorders introduced by malware obfuscation. After DiffHandle generalizes resources that system calls operate on, and ISA gains common system calls from these generalized but disordered system call sequences of the same family, SimBehavior obtains handle dependencies and order restrictions between common system calls by mapping these system calls into original system call sequences. Finally, these common system calls handle dependencies and order restrictions between these system calls' makeup of the DiffHandle-signature of the same family. The evaluation was performed on 331 malware families, categorized into eight families. SimBehavior achieved an average detection rate of 92.4%.

ALTERDROID is a dynamic analysis approach for detecting hidden or obfuscated malware components that are distributed as parts of an app package. The key idea behind ALTERDROID is analysing the behavioural differences between the original app and a number of automatically generated versions of it, where a number of modifications have been carefully injected. Observable differences in terms of activities that appear or vanish

in the modified app are recorded, and the resulting differential signature is analysed through a pattern-matching process driven by rules that relate different types of hidden functionalities with patterns found in the signature. The extensive experimental results obtained by testing ALTERDROID over relevant apps and malware samples support the quality and viability of this system (Suarez-Tangil et al., 2016).

(b) ***Static-based Signature***

AndroSimilar uses static-based signature by examining the Java code of the Android applications (Faruki et al., 2013). It extracts statistical features to detect malicious applications. The authors claim that their proposed method is effective against code obfuscation and repackaging, widely used techniques to evade signature-based detection methods. To calculate features that remain persistent among related samples, normalized entropy features are assigned with precedence by associating normalized bytes based on empirical observations. This rank is a measure of the occurrence of features obtained by reading the byte content. A feature whose likelihood of occurrence is lowest receives a low rank. Certain features having a very high or low score receive a null score to avoid the introduction of weak features during attribute selection. The evaluation shows 98.89% detection rate.

DroidAnalytics is an Android malware detection system based on static-based signature, which automatically collects, extracts and analyses signatures of Android application files (M. Zheng et al., 2013b). It extracts methods and classes from the application's Java code and employs them as signatures. Subsequently, the generated signatures are used to detect malicious applications. Nonetheless, the aforementioned method is useful only for known malware, whereas with the advent of new threats, the same process must be performed and the generated signature has to be added to the database.

Overall, the misuse-based approach is precise for known malware; however, it is unable to detect zero-day malware. A new variant of malware has a different signature to previously known malware. Thus, analysts need to generate its signature and update the database. However, by the time the database is updated, the malware has already damaged mobile devices. Due to such weaknesses, researchers turned to anomaly-based approach.

2.4.3.2 Anomaly-based Detection

Anomaly-based methods depend on classifiers to train a system to differentiate between normal and malware behaviour, which can be used to detect anomalies to discover unknown malware.

(Sangkatsanee et al., 2011) proposed an anomaly detection system that contains 12 features of network traffic, such as source and destination port, Transmission Control Protocol (TCP) flags (i.e. fin, syn, push and urgent flag), UDP and ICMP packets. The adopted classifier was a decision tree, and it successfully obtained a 99.4% accuracy rate. Previously mentioned work performed by Su et al. also followed anomaly-based detection by using machine learning classifiers (Su et al., 2012).

Sahs and Khan extracted Android file permissions and control flow graphs (CFG) (Allen, 1970). Then they used a SVM to make the system learn to distinguish between patterns of malicious applications and normal ones. With a 93% detection rate the results were accurate (Sahs & Khan, 2012).

DroidMat is another example of machine learning in malware detection (D.-J. Wu et al., 2012b). The authors used K-means to inspect an application, obtaining 87.39% detection accuracy as opposed to the misuse-based methods mentioned earlier with only 79.6%.

Shabtai et al. identified the best classification method out of six classifiers, namely Decision Tree, Naïve Bayes, Bayesian Network, K-means, histogram, and logistic

regression, using the Andromaly framework. The framework adopts feature selection methods such as chi-square, fisher score, and information gain to enhance detection accuracy. As a result, they managed to achieve a 99.9% accuracy rate with the decision tree classifier and information gain method. Although they achieved acceptable accuracy, they used self-written malware to test their framework, which could have produced unrealistic results (Shabtai et al., 2012).

RobotDroid was proposed which is based on the SVM machine learning classifier to detect unknown malware in mobile devices. The focus was on privacy information leakage and hidden payment services. They evaluated three malware types, namely Gemini, DroidDream and Plankton. As a result, this framework is limited to a few malware types, and more would be required to increase detection accuracy (Zhao et al., 2012).

DroidScreening employs an anomaly-based approach to detect Android malware (J. Yu et al., 2016). It extracts many features from Android application installation files. The features are requested permissions, existence of native code in the Java code, use of reflection in the Java code, and issued system calls. This system then applies a lazy associative classification (LAC) algorithm to the extracted features to build a detection model. The detection model is used to categorize the Android application as normal or malicious. It achieved a 97% detection rate using a dataset of 1,554 malware samples.

Thus, the anomaly-based detection techniques were chosen for the purpose of this study because they are capable of detecting anomalies based on what they learn.

2.4.4 Point of Detection

We categorize related research works based on where the detection is implemented. They are local-based and cloud-based. As resources, such as battery power, are limited in

mobile devices, this grouping compares the practicality of the proposed approaches. The strengths and weaknesses of each category are discussed in the following sections.

2.4.4.1 Local-based Detection

The local-based detection process is implemented directly on mobile devices. TaintDroid detects leakage of private data (Enck et al., 2010). It labels important data and follows them to see whether they leave the device.

The MADAM was implemented on a device. They claimed that the overall performance overhead is acceptable, with 3% memory consumption, 7% CPU overhead, and 5% battery usage (Dini et al., 2012). Similarly, Andromaly was evaluated on five different devices (Shabtai et al., 2012). Feature extraction and application of machine learning algorithms were performed on devices. Unfortunately, information regarding Andromaly's resource consumption is not available.

2.4.4.2 Cloud-based Detection

Cloud-based detection is defined as replicating a real device on the cloud, and reporting any changes of the device to the servers. Thus, the replicated device is used to monitor the real device; processing and analysis are also done on the replicated device that is on the cloud. Therefore, the real device does not take a heavy workload. Any suspicious activity is then reported to the users.

Zonouz et al. proposed a cloud-based smartphone malware detection called Secloud. It analyses malware in a real test bed with a direct network connection to the cloud. It emulates a replica of mobile devices on the cloud and keeps it synchronised by continuously reporting every changes in the real devices. Secloud also redirects the devices' network traffic to the cloud through a proxy. This way the whole analysis process takes place in the cloud (Zonouz et al., 2013).

Similarly, CloudShield was designed for peer-to-peer networks (Barbera et al., 2012). It is based on virtual copies of real devices that run on the cloud. The peer-to-peer network of clones is used to compute the best strategy to patch the smartphones in such a way that the number of devices to patch is low. The authors simulated worm attacks that affect the network load. They explored the idea of a peer-to-peer network of virtual smartphone clones running on the cloud, which can help stop worm attacks spreading from smartphone to smartphone on the mobile network. The worm propagates by using Bluetooth connections, MMS messages, phone calls, or any other means of infection available among smartphones. The final experiments show that CloudShield outperforms state-of-the art worm-containment mechanisms for mobile wireless networks.

Paranoid Android is another related work that duplicates the real device on the cloud and passes any changes of device to the cloud (Portokalidis et al., 2010). It is capable of running multiple detection methods simultaneously. The aforementioned works need a constant network connection to the cloud to report every single event and changes of devices (i.e. new application installation, application update, system calls, etc.) to the server. Such a design is not practical in a situation where network connectivity is not available. Furthermore, it consumes bandwidth that could be costly for the user; energy consumption is also another concern for this approach.

This study opts for the cloud-based approach; however, the offloading method is used. The cloning approach has various disadvantages as opposed to the offloading approach, which offers processing on the cloud without cloning mobile devices. The details of this technique are explained in Section 4.3.1.

2.5 Discussion

This section points out key aspects of Android features, analysis and detection methods, and point of detection. It also discusses the weaknesses, strengths, and potential gaps in each section.

a) **Android Features:** As discussed earlier, Android features are grouped into static, dynamic and hybrid features. Among static features, permission is the most used one. After that, Intent comes in the second place. Intent is declared by the application in XML file and in Java code. This work chose Intent in Java code over permission, since Intent potentially is a rich feature and there is a gap in literature as no other work has evaluated its importance. Therefore, the choice of Intent sounds promised.

Among dynamic features, system calls and network traffic are two most important features. System call represents application's behaviour in the device, while network traffic represents behaviour of the application outside the device. The main reason for choosing network traffic for this work is that it represents malware conversation with attacker as it leaks user's data or receives command from the attacker. The other reason is that network traffic is not analysed thoroughly in literature and there is a potential to analyse it further. It is also worth noting that collecting system calls in Android requires rooted device and complicated methods as opposed to network traffic. By choosing static and dynamic features, we will have a dataset of hybrid features that is a comprehensive group of features.

b) **Mobile Malware Analysis:** It is categorized into static, dynamic, and hybrid. Static analysis is examination of Android installation file and its content. Dynamic analysis is analysis of Android application behaviour after execution. Static analysis has problem of code obfuscation, Java reflection, and dynamically-loaded code. Dynamic analysis has weakness of code coverage. Therefore, combining the two method results in more robust

analysis, which is called hybrid analysis. Among various recent works discussed in Section 2.4.2.3, DroidRanger and Mobile-Sandbox analyse permission for static analysis that is bypassed by malware such as Basebridge. TaintDroid cannot find reflective method calls. The Harvester and AppDoctor take complicated approach that are difficult to implement. Andro-Dumpsys relies on replica of the device on the cloud, which has its own disadvantages (Section 2.4.4). Overall, available Android hybrid analyses are difficult to implement for the end user, and they employ features that are bypassed by malware. Therefore, this work adopts hybrid analysis by analysing Intent in Java code for static analysis and network traffic for dynamic analysis.

c) **Mobile Malware Detection:** It is divided into misuse-based detection and anomaly-based detection. The misuse-based detection uses signature of malware to detect a particular malicious application that matches the signature. The attackers have easily bypassed this detection method by slightly changing their code. Therefore, we chose anomaly-based detection that uses machine learning to detect malware.

d) **Point of Detection:** The detection process is performed on the device or in cloud. Since running detection process on the device consumes lots of battery power, we choose to implement cloud-based detection using offloading technique (4.2.1).

e) **Energy Consumption Measurement:** It is worth noting that in current literature, it is not a custom to measure energy consumption of the proposed method. It is an important and an oversight issue. The best methods become unproductive if they drain battery of the device. It is not appealing to users. This work tends to introduce this concept to research community and hopes that future works measure energy consumption of their method as they measure accuracy.

2.6 Summary

This chapter has overviewed the evolution of mobile malware, their characteristics, the Android operating system, and its security features. Additionally, malware analysis and detection methods were reviewed. We have analysed the most related research works from four different perspectives. They are feature selection, analysis methods, detection approach, and point of detection. Apart from reviewing research works, this chapter lays the foundation of this study as we decide which features to select, what analysis methods to choose, what detection approach to choose, and where to implement the detection method. Various tools are needed to apply the selected methods on a raw dataset. In the next chapter, we focus on analysis and detection tools.

CHAPTER 3: DROIDLAB - MOBILE MALWARE ANALYSIS TOOLS

The previous chapter discussed research works related to this study. These works used various tools to analyse and conduct their experiments. Becoming familiar with the available tools profoundly extends the understanding of malware analysis techniques.

This chapter gathers several monitoring and analysis tools. These tools have been used in research works to perform experiments on malware. Structurally, this chapter consists of three sections with relevant sub-sections. We have categorized analysis tools into static tools and dynamic tools, based on their monitoring and analysis approach, in Sections 3.1 and 3.2 respectively. Section 3.3 discusses available tools in measuring energy consumption. The description and drawback of each one is mentioned, and our selected tool is specified.

3.1 Static Analysis Tools

Static analysis tools are used to analyse Android APK files. Technically, they are capable of inspecting various components of the APK file. The following sub-sections discuss several static analysis tools.

3.1.1 Androguard

Androguard (Anthony Desnos, 2010) is an interactive static analysis tool for Android applications. It is capable of dissecting the Android application into its components through its API.[2] It also allows further analysis of the binary code and access to its various parts such as class names, method names, variables, strings, etc. The Androguard API has the following features:

[2] http://doc.androguard.re/html/index.html

a) **APK**. The Android Application Package (APK) is the file type used to install Android applications. It entails several components such as AndroidManifest.xml, DEX file, and resources. The Androguard dissects the XML file and returns its elements such as activities, permissions, minimum SDK version, maximum SDK version, etc. It is also capable of accessing the binary code of the APK file.

b) **DVM**. The Dalvik Virtual Machine (DVM) is an important component of the Android operating system, responsible for running each application in its own virtual machine. This feature of Androguard allows access to the DEX file of the Android application, which contains the application code. More specifically, it retrieves Java metadata about an application, the name and size of its classes, methods, and variables among other static features from the DVM (Suarez-Tangil, 2014).

c) **Analysis**. This part of the API provides more details on the Java code. It specifies permissions that are used in Java code, rather than permissions declared by the application inside the xml file. It also identifies whether certain libraries are used in the code such as crypto, dynamic code, native code, and reflection code. Additionally, it also provides a Control Flow Graph (CFG) representation of the Dalvik code flow (Suarez-Tangil, 2014).

d) **Bytecode**. The Dalvik code executed by the DVM is a compact and efficient instruction set (numeric codes, constants, and references) that encodes executable programs into a portable language called bytecode. This bytecode is translated into native machine code at run time. This facilitates the portability of the bytecode itself across different hardware-specific platforms. However, it also makes the reverse engineering analysis of Android apps easier. This component of Androguard provides a number of methods that aid bytecode analysis (Suarez-Tangil, 2014).

3.1.2 ApkTool

ApkTool is a reverse engineering tool for Android applications (Wiśniewski, 2010). As mentioned in Section 2.2.1, Android applications are written in Java and compiled to

DEX file. ApkTool is used to reverse this process and decode applications into nearly the original form, which is smali[3] code. It is possible to modify smali code and rebuild the application to APK format. Therefore, we describe two main functions of the Apktool as follows:

a) **Decompile**. It reverses the Android application to a readable format called smali bytecode. In addition to Java code, it decompiles other APK components like the XML file, resources folder, libraries, and assets.

b) **Recompile**. After decompiling the application, the user may modify its content and then recompile the modified application using Apktool, which results in a new application. The new application may differ from the original one in functionality.

3.1.3 AXMLPrinter

The AXMLPrinter is a static analysis tool designed to merely decode the AndroidManifest.xml file in the APK package. It is useful when an analyst intends to extract some data relating to the application. This command line tool is faster as compared to Apktool, as it just decodes the xml file that contains the minimum SDK version, maximum SDK version, activities, permissions, intent-filters, etc.

3.2 Dynamic Analysis Tools

Dynamic analysis tools help monitor applications' behaviours. Based on their behaviour, it is possible to identify their characteristics and categorize them as benign or malicious. In the following, we discuss the most famous dynamic analysis tools.

[3] https://github.com/JesusFreke/smali

3.2.1 Wireshark

The Wireshark is a well-known network protocol analyser that captures network traffic and represents it in a graphical way. For each packet, various network layers such as physical layer, IP layer, network layer, and if applicable HTTP layer is shown. The Wireshark is used for network troubleshooting, analysis and communication protocol design. The output file of the Wireshark has a PCAP extension, which can be opened with other programs such as tcpdump. The displayed data can be refined with filters available in the software. In addition, it is possible to detect VoIP traffic and decode the data. Any media data such as pictures and videos from the captured traffic can be recovered and played. Various graphs and statistics that help to understand the network traffic when dealing with massive data volumes can be drawn from the software.

In this study, the Wireshark is used to filter TCP packets from numerous types of packets such as ARP, DNS, etc. Furthermore, Tshark is a command line version of the Wireshark. It is a more powerful tool than the Wireshark, since it gives a user the power to extract different network features such as packet size and connection duration from a myriad of network packets with a line of command. Windows shell scripting can also be used to automate the process of feature extraction, as applying same command to a pool of captured network traffic is time consuming.

3.2.2 DroidBox

DroidBox[4] performs dynamic analyses of Android applications. The following information is generated when an analysis is complete:

a) Hashes for the analysed package

[4] https://github.com/pjlantz/droidbox

b) Incoming/outgoing network data

c) File read and write operations

d) Started services and loaded classes through DexClassLoader

e) Information leaks via the network, file and SMS

f) Circumvented permissions

g) Cryptographic operations performed using Android API

h) Listing broadcast receivers

i) Sent SMS and phone calls

Additionally, two graphs are generated, visualizing the behaviour of the package, one showing the temporal order of operations and the other a treemap (Shneiderman & Wattenberg, 2001) that can be used to check similarity between analysed packages.

3.2.3 TaintDroid

We discussed TaintDroid in Section 2.4.2.3. It marks sensitive data in applications' code and tracks them while the applications are running. Basically, it tracks how applications use sensitive information, which is acquired by integrating TaintDroid into the Android platform at a low level. It also shows the information flow inside Android applications. Figure 3.1 depicts TaintDroid architecture as illustrated by Enck et al. (Enck et al., 2010).

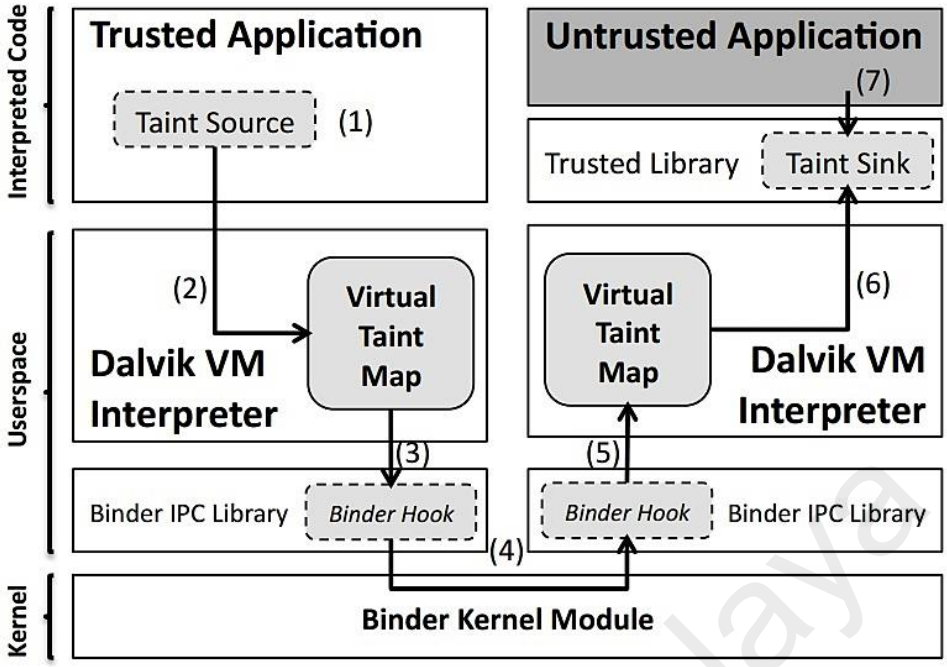


Figure 3.1. TaintDroid Architecture as Depicted in (Enck et al., 2010)

TaintDroid source code is available at the author's site[5] for several versions of Android e.g. Android 2.1 and Android 2.3.

3.3 Machine Learning Tools

We chose to follow the anomaly-based detection method, as discussed in Section 2.4.3. To do so, the following tools are used in this work and discussed in the following sections.

3.3.1 WEKA

Weka is a flightless bird with an inquisitive nature that is found only on the islands of New Zealand. Waikato Environment for Knowledge Analysis (WEKA) is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to a dataset or called from your own Java code. Weka contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization. It is also well suited for developing new machine learning schemes. Weka is an exquisite

[5] http://appanalysis.org

choice as it provides a graphical user interface and is easy to work with and understand (Hall et al., 2009).

3.3.2 TensorFlow

TensorFlow is an open source library for machine learning, including deep learning. It was developed by Google and released in November 2015. It is used in various Google services such as speech recognition, Gmail, Google Photos, and searches. TensorFlow can be used to build deep learning systems for any task. A machine learning system can be represented in TensorFlow using the data flow graph. This is a directed graph where the nodes contain computations and the edges are the flow of tensors through this graph. Tensors are mathematical objects that can be described using an n-dimensional array. They are the primary data type in TensorFlow: they are used to store data, which can be transformed by operations. These operations describe the actual functionality of the computation, and an instantiation of an operation corresponds to a node in the data flow graph. The input and output of operations are zero or more tensors. To create a machine learning system in TensorFlow, it needs to be expressed as a data flow graph. This data flow graph can then be interacted with in a session. The most important function of a session is to run the data flow graph. When running a computation, a dictionary of inputs is fed to the graph. The graph then executes the operations, and outputs the result of the final operation (van Niedek, 2016). TensorFlow has been used in various research fields such as the prediction of financial markets (Vahala), learning structured representations for geometry (B. Kim, 2016), and phonetic classification (van Niedek, 2016). It is worth mentioning that TensorFlow is capable of running in mobile devices[6].

[6] https://www.tensorflow.org/mobile/

3.4 Energy Consumption Profilers

The rapid advancements of the communication and computing capabilities of mobile devices have led to faster depletion of batteries. Since 1999, energy profilers have therefore been receiving attention. An energy profiler is defined as a system that monitors and characterizes the energy consumption of a device (Tarkoma et al., 2014). Although Android has built-in energy APIs, they typically allow applications to query and subscribe to coarse-grained information, such as battery voltage, battery health, battery capacity, and temperature. Thus, various energy consumption profilers have been proposed to access fine-grained and per-application information. Table 3.1 lists well-known systems in chronological order.

Table 3.1. A List of Energy Consumption Profilers

Name	Year	Purpose
PowerScope	1999	Energy profiling of device and processes
Nokia Energy Profiler	2006–2007	On-device standalone profiler
PowerTutor	2009	Hybrid profiler based on PowerBooter
eProf	2012	Fine-grained power model to identify energy bugs in applications
AppScope	2012	Fine-grained energy profiler for applications based on DevScope

The PowerScope (Flinn & Satyanarayanan, 1999) is an example of early energy profiler developed in 1999. It analyses the energy consumption for each process in the operating system. The Nokia Energy Profiler (Creus & Kuulusa, 2007) is another system implemented in 2006-2007. It was developed for the Symbian[7] Series 60 devices to determine the power consumption. The PowerTutor (L. Zhang et al., 2010) is an Android application that shows energy use in a similar way to Android's built-in API, but with

[7] http://series60.kiev.ua

breakdowns per resource, such as CPU, Wi-Fi, and screen. The PowerTutor does not consider the effects of running multiple applications simultaneously but rather estimates the energy consumption for each application separately. The eProf (Pathak et al., 2012) is used to identify energy bugs in applications. It is useful for application developers to debug their product from the energy consumption point of view.

The energy profiler of our choice for this study is AppScope (Yoon et al., 2012) due to its accuracy, and the fact that it provides fine-grained information about each application, its processes, and resource consumption (i.e. CPU, display, etc.). Alternatively, PowerTutor is also considered for this study. Despite working well and producing accurate results, AppScope is designed to work on a specific old device. In addition, some applications do not work on old devices. Therefore, PowerTutor is our alternate option for measuring energy consumption. The following sections provide more details on AppScope and PowerTutor.

3.4.1 AppScope

AppScope is an energy profiler that monitors kernel activity of Android devices. It collects usage information from the monitored device and estimates consumption of each running application using an energy model provided by DevScope (Jung et al., 2012). AppScope displays the categorized amount of energy consumed by an application, where each category is associated with a component of the device (CPU, Wi-Fi, cellular, etc.). AppScope uses an event-driven monitoring method that uses little power and provides high accuracy. In fact, its authors report that AppScope incurs approximately 35mW and 2.1% in power consumption and CPU utilization overhead, respectively. AppScope provides information about the power consumed by different applications running in the device. Additionally, it also offers information about the energy consumed by each individual process executed by every app.

3.4.2 PowerTutor

PowerTutor was developed by the University of Michigan Ph.D. students Mark Gordon, Lide Zhang, and Birjodh Tiwana. It is an application that displays the power consumed by major system components such as CPU, network interface, display, GPS receiver, and different applications. PowerTutor uses a power consumption model built by direct measurements during careful control of device power management states. This model generally provides power consumption estimates within 5% of actual values. A configurable display for power consumption history is provided. It also provides users with a text-file based output containing detailed results. PowerTutor can also be used to monitor the power consumption of a specific application (Z Yang, 2012).

3.5 Summary

This chapter discussed several tools available for Android malware analysis. We categorized them into static analysis tools and dynamic analysis tools. Moreover, it explored different tools for applying machine learning algorithms, and for measuring the energy consumption of mobile applications.

Having reviewed the related research works and analysis tools, it is time to discuss our proposed system. The next chapter describes the proposed framework for this study. Various components of the framework are discussed along with techniques and services used to develop the framework.

CHAPTER 4: MOBILE MALWARE ANALYSIS AND DETECTION: THE FRAMEWORK

The previous chapter explored research works related to this study, and identified their weaknesses and strengths. Additionally, overviews of various tools used in malware analysis helped to explain the methodology of the analysis methods. This chapter details the proposed framework, aiming at minimising energy consumption of the analysis and detection processes while achieving high detection accuracy. The following sections discuss the architecture of the proposed system, along with its different modules and the rationale behind them.

4.1 The DroidProtect Traits

The proposed architecture has the following advantages over the current methods.

1) **Intelligent:** This methodology employs machine learning to detect malware, whereas the current methods are based on a signature database that needs to be updated constantly.

2) **Hybrid Analysis:** Our method monitors static and dynamic features of the device. The static analysis is done on Android Intent, which manifests real intentions of applications (details in Section 5.2). The dynamic analysis part monitors network traffic of Android devices and examines the traffic to detect anomalies. This way, we have a higher chance of detecting malware, given the fact that over 90% of malware request network connectivity to connect to a malicious server and receive spiteful commands (Feizollah et al., 2015).

3) **Lightweight:** the proposed method uses an offloading technique, where monitoring, capturing, feature extraction is done on the devices, and the features are sent to a remote server. The heavy workload of the extensive detection process (using machine learning) is performed on servers.

4) **Scalable:** As mentioned earlier, a machine learning model is used to classify an application as malicious or clean. The system administrator is able to extend the dataset by adding more applications, and to update the model by re-training the machine learning algorithm. Thus, the result is a robust and more powerful malware detection model for the Android operating system.

5) **Offloading:** The proposed architecture uses offloading technique to upload only important features from a device to servers. Offloading has been used in various research fields and proved to be efficient. However, this is the first time that we propose to use it for mobile malware detection.

6) **Feature Engineered:** This work meticulously examines available features in Android malware detection and selects the most effective ones. In fact, the choice of implicit and explicit Intents is a novel one that has not been used before. It is also efficient and effective compared to Android permission that has been widely used (Section 5.2.1). The network traffic feature is also chosen carefully to make sure that it is effective in malware detection. The use of TCP and HTTP protocols to analyse mobile malware is unprecedented in this work (Section 5.3.3).

4.2 The Architecture

As mentioned earlier, with the prevalence of mobile devices, security threats are growing in number and seriousness. Among the mobile operating systems, Google's Android has been attacked more than others. From April 2013 until June 2013, the number of malware for Android doubled. Such growth prompted the antivirus industry to respond to contain the multiplication of malware. Their response was similar to that to malware in PCs, namely by developing antivirus applications for Android devices. However, the characteristics of Android devices are different to those of PCs. The Android operating system has different approaches for controlling the system's resources, hardware and application resources. PCs treat programs as trusted ones, giving them access to various

parts of the system, whereas Android devices limit access of each application to its own data. Therefore, antivirus applications have difficulty accessing other applications' directories. Moreover, resources (i.e. CPU, memory, battery) on Android devices are limited. To adapt antivirus programs on PCs to Android devices, we propose **DroidProtect**. This system attacks malware by performing static and dynamic analysis and extracting selected features on the device. The heavy process of identifying a malware is done on servers and the response is sent back to the device. This way, the energy consumption of the devices is reduced. Figure 4.1 depicts the architecture of the DroidProtect.

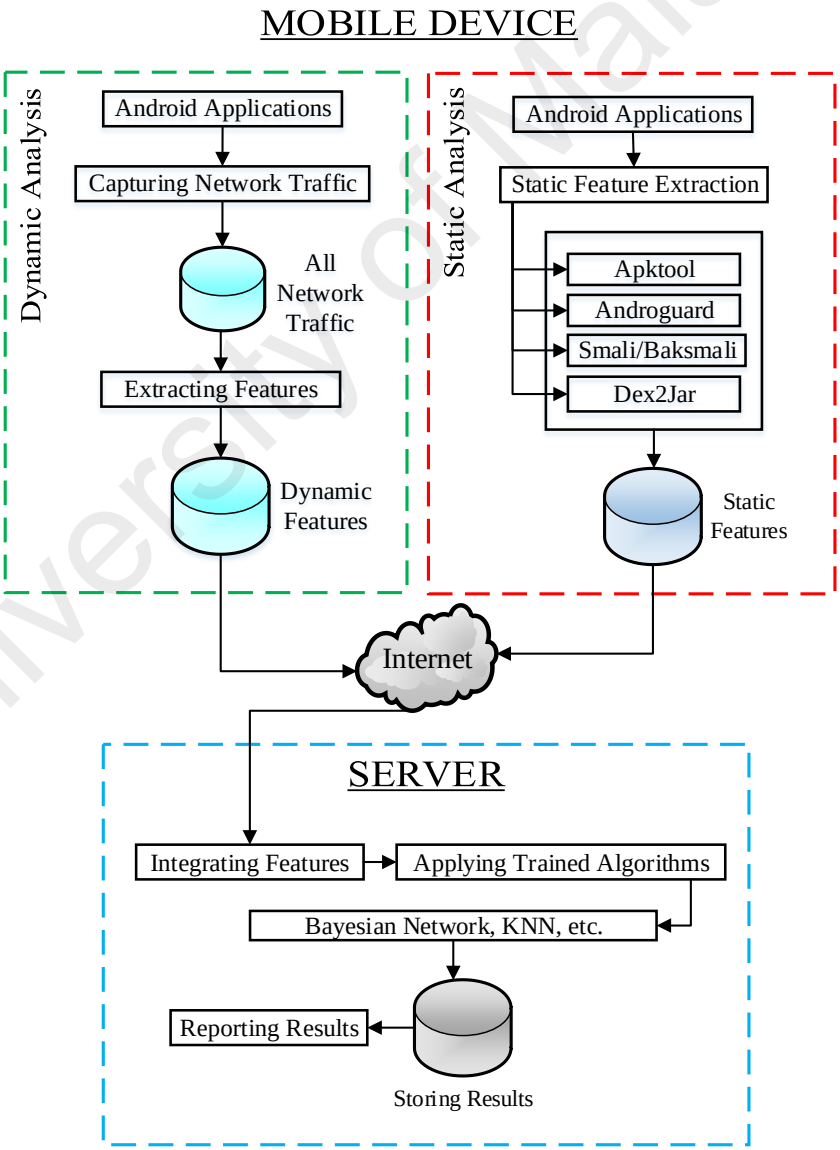


Figure 4.1. Architecture of the DroidProtect

The above figure consists of three modules: static analysis module, dynamic analysis module and server module.

The static analysis module is responsible for analysing the APK file of applications and extracting related features. The extracted features are sent to the server for the malware detection process. The dynamic analysis module collects network traffic of the device and extracts network-related features. Similar to the static analysis module, the extracted features are sent to the server for malware detection purposes. All the extracted features are received and integrated in the server modules.

The extracted static and dynamic features are received from the mobile device. They are thoroughly analysed to determine the cleanness of the device. The process is performed by feeding the data to the machine learning model. The model is prepared offline by a system administrator. A data sample consisting of thousands of malware and clean applications is selected for this process. Various features are extracted and the final dataset is fed to machine learning algorithms. Based on their performance and accuracy, the best algorithm is selected. At the end of this process, a model is produced. The model is then used in the server module to determine the cleanness of new data received from the mobile device. The process of producing the model and their effectiveness is discussed in detail in the next chapter. At the end, the results of the experiments are sent back to the device, and presented to the user.

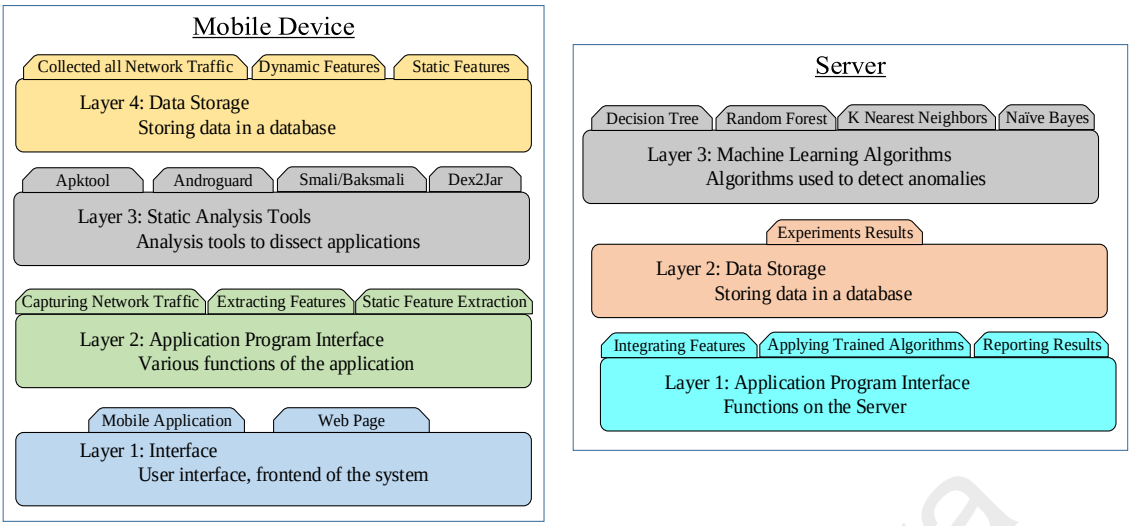


Figure 4.2. Layer Architecture of the DroidProtect

Figure 4.2 shows the architecture in a layer structure. Each layer represents a function in the respective modules. For each layer, a specific action or tool is mentioned. For instance, the API layer in the mobile device performs network traffic capturing and features extraction for static and dynamic data. It is also necessary to show the flow of the process in DroidProtect. Figure 4.3 shows layers interactions and the process flow.

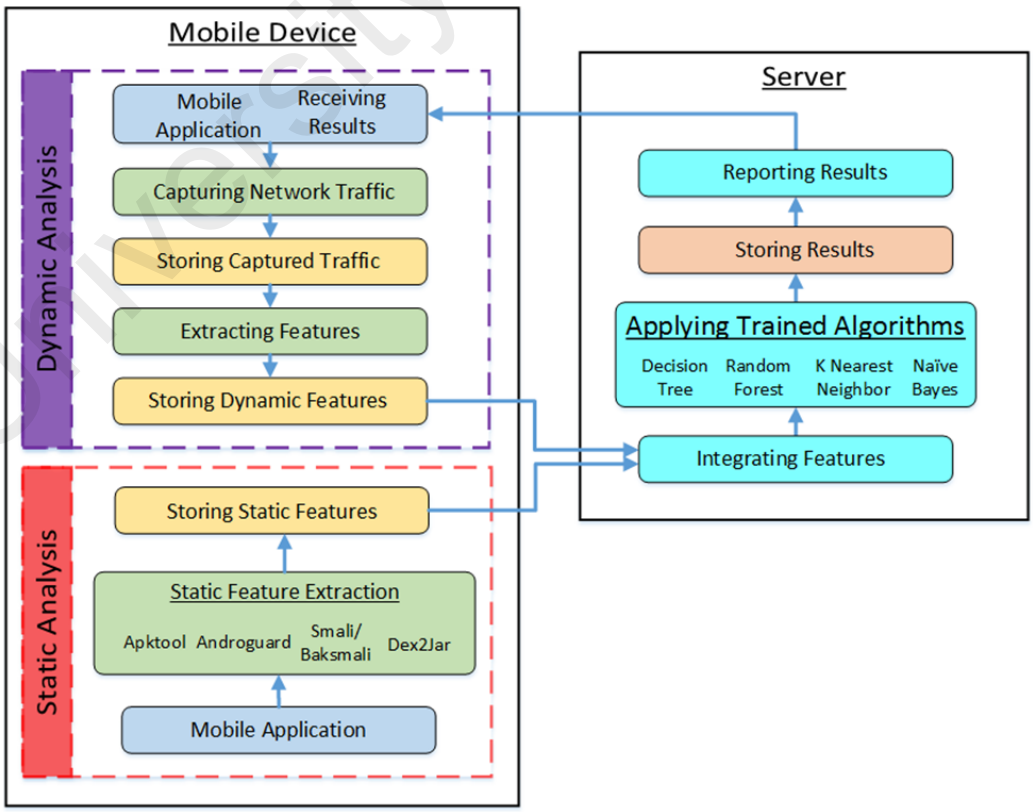


Figure 4.3. Layers Interactions

4.3 The Used Methods and Services

In this section, various methods and services used in the proposed architecture are discussed.

4.3.1 Computation Offloading

Mobile devices suffer from limited battery life, thus running resource-intensive applications is a gruelling task. To overcome this situation, an offloading technique was introduced where the heavy computation task runs in a different environment than the mobile device. Computation offloading is also different from the migration model used in multiprocessor systems and grid computing, where a process may be migrated for load balancing (Powell & Miller, 1983). The key difference is that computation offloading migrates programs to servers outside of the users' immediate computing environment; process migration for grid computing typically occurs from one computer to another within the same computing environment (Kumar et al., 2013).

This method is helpful in our architecture in which the collected features (static and dynamic) are sent to servers for analysis. This way, mobile devices save on the battery usage otherwise required to perform a heavy workload.

4.3.2 Machine Learning Tools

Machine learning classifiers have for several years helped in developing intelligent systems by training machines on how to make decisions. With a dataset labelled as input, machine learning constructs a model that is applicable to new data to identify pattern similarities. Numerous studies with significant detection results have adopted a similar approach, with the intention of detecting intrusions effectively (Feizollah et al., 2013; Narudin et al., 2016; Sangkatsanee et al., 2011; Zhao et al., 2012).

4.4 Summary

This chapter proposed a framework (DroidProtect) for mobile malware analysis and detection. Its many components were explained via architecture figures and layer interactions. Additionally, the methods and services used were discussed in order to better understand this framework. Moreover, the characteristics of DroidProtect were explained to show its contributions.

Proposing a framework requires validation to show that it is capable of fulfilling the mentioned objectives. The next chapter serves this purpose by performing four comprehensive experiments on the DroidProtect.

CHAPTER 5: EVALUATION OF THE MOBILE MALWARE ANALYSIS AND DETECTION FRAMEWORK

The previous chapter proposed a novel framework for mobile malware analysis and detection. The novelty of the framework is to extract features on the device and send them to a server for processing. This way, the heavy process of analysis and detection is performed on the server rather than on the device. Furthermore, it is unnecessary to duplicate the real device on the cloud and synchronise every change, which raises security concerns.

The objective of this chapter is to evaluate the proposed framework in terms of feasibility, soundness, and validity. The evaluation is carried out to verify how well the system fulfils the intended objectives. In order to perform a comprehensive evaluation of the system, appropriate evaluation criteria that address system performance issues are used. This chapter performs extensive experiments related to various parts of the framework to achieve final results.

Structurally, this chapter starts by describing the data samples used in this study, as it is a vital part of every experiment to include valid and trustworthy data. The rest of this chapter includes four experiments pertaining to static analysis, dynamic analysis, and energy consumption of the framework. Experiment one investigates the static analysis section. It analyses Android Intent (explicit and implicit) as a new feature in Android malware detection. It fills the gap in recent research works (Section 5.2.1) by examining the effectiveness of this feature. The same experiment is also performed on Android permission under the same experimental conditions. At the end, the results of the two experiments are compared. The second and the third experiments (Section 5.3) relate to dynamic analysis, specifically the network traffic of Android applications. In the two experiments, the best network-related features are selected by using the described

algorithms, and the best classifier is chosen according to the results, which are compared to the recent research works to magnify the contributions of this study. The fourth (Section 5.4) experiment aims to satisfy the problem statement of this study by calculating the energy consumption of the proposed framework under various conditions. Additionally, the results are compared to other available solutions in this domain.

5.1 Dataset Description

Every experiment requires a dataset based on which the authors evaluate their proposed system. Android malware is a relatively new research area. The first Android malware was discovered in 2010 (Lookout, 2010). Initially, researchers did not have a solid and standard dataset of samples to work with. Instead, they tended to write their own malware and assessed their system on self-written malware (Chekina et al., 2012; Shabtai, 2010). Other researchers tried to collect samples through some websites that shared Android malware samples, such as Contagio.[8] Therefore, the weakness was the limitation of malware samples that in turn made the evaluation of their system unreliable. This section discusses details of the most widely used Android malware data samples.

5.1.1 MalGenome

The MalGenome data sample includes 1,260 Android malwares in 49 different families (Yajin & Xuxian, 2012). A malware family is a collection of malware presenting similar a behaviour. This collection was gathered between August 2010 and October 2011 by the North Carolina State University. The authors analysed the data samples and found that around one third (36.7%) of the collected malware samples leverage root-level exploits to fully compromise Android security, posing the highest level of threats to users' security and privacy. Additionally, more than 90% of malware turn the compromised devices into

[8] http://contagiominidump.blogspot.com

a botnet controlled through network or short messages. Among the 49 malware families, 28 (with 571 or 45.3% samples) of them have the built-in support of sending out messages (to premium-rate numbers) or making phone calls without user awareness. They mentioned that 27 malware families (with 644 or 51.1% samples) are harvesting users' information, including user accounts and short messages stored on the devices (Appendix B). Table 5.1 tabulates malware families available in MalGenome along with number of samples per family and their discovery date.

Table 5.1. Malware Families in MalGenome Data Sample

Malware Family	No. of samples	Discovery Month	Malware Family	No. of samples	Discovery Month
ADRD	22	2011-02	GingerMaster	4	2011-08
AnserverBot	187	2011-09	GoldDream	47	2011-07
Asroot	8	2011-09	Gone60	9	2011-09
Basebridge	122	2011-06	GPSSMSSpy	6	2010-08
BeanBot	8	2011-10	HippoSMS	4	2011-07
BgServ	9	2011-03	Jifake	1	2011-10
CoinPirate	1	2011-07	jSMSHider	16	2011-06
CruseWin	2	2011-07	Kmin	52	2011-10
DogWars	1	2011-08	Lovetrap	1	2011-07
DroidCoupon	1	2011-09	NickyBot	1	2011-08
DroidDeluxe	1	2011-09	Nickyspy	2	2011-07
DroidDream	16	2011-03	Pjapps	58	2011-02
DroidDreamLight	46	2011-05	Plankton	11	2011-06
DroidKungFu1	34	2011-06	RogueLemon	2	2011-10
DroidKungFu2	30	2011-07	RogueSPPush	9	2011-08
DroidKungFu3	309	2011-08	SMSReplicator	1	2010-11
DroidKungFu4	96	2011-10	SndApps	10	2011-07
DroidKungFuSapp	3	2011-10	Spitmo	1	2011-09
DroidKungFuUpdate	1	2011-10	TapSnake	2	2010-08
Endofday	1	2011-05	Walkinwat	1	2011-03
FakeNetflix	1	2011-10	YZHC	22	2011-06
FakePlayer	6	2010-08	zHash	11	2011-03
GamblerSMS	1	2011-07	Zitmo	1	2011-07
Geinimi	69	2010-13	zSone	12	2011-05
GGTracker	1	2011-06			

5.1.2 Drebin

Based on the nature of malware, they change shape and infecting technique to evade detection. Therefore, it behoves researchers to update the data samples to develop systems that are more effective. By introducing Drebin in 2014, this need was fulfilled. The Drebin data sample was published in 2014 by Arp et al. (Arp et al., 2014). It is a collection of 5,560 Android malware categorized into 179 different families. It was collected between August 2010 and October 2012. The authors scanned the Drebin with antivirus applications. They report that while the best scanners detected over 90% of the malware, others detected less than 10% of the data sample. The Drebin was well-accepted among researchers (Dash et al., 2016; Varsha et al., 2016). Upon requesting the data sample, we acquired it for this study.

5.1.3 AndroZoo

Unlike the mentioned data samples, AndroZoo is a growing collection of Android applications from several sources, including the official Google Play. As of writing this thesis, AndroZoo contains more than five million Android applications. Not only does this data sample accommodate Android malware, but it contains benign applications as well (Allix et al., 2016). Crawling various sources started in late 2011 and has continued ever since.

The 14 sources include Google Play, Anzhi, AppChina, 1mobile, AnnGeeks, Slideme, HiApk, ProAndroid, etc. The AndroZoo sends all the downloaded applications to the VirusTotal for scanning. The number of antivirus software that detect an application as malicious is stored in the metadata file, as vt_detection. The metadata file is available on the AndroZoo website[9] and is updated regularly. As a result, if vt_detection is zero, then

[9] http://androzoo.uni.lu

the application is clean. Otherwise, it is considered as malware. This feature allows researchers to use AndroZoo not only as a malware repository, but also as a clean application repository.

5.1.4 Malware Repositories

Apart from the aforementioned data samples, other data samples are also available. IccRE (Icc Repository) is a collection of 445 Android malware that leak privacy data through inter-component communication (Li et al., 2015). The authors performed inter-component analysis to detect privacy leaks between components of Android applications. Their system reached a precision of 95%. They decided to share the data sample with the research community on a request basis.

VirusShare[10] is another repository of malware samples that provides security researchers, incident responders, and forensic analysts access to samples of malicious code. Access to the site is granted by invitation only. To request, researchers need to email the administrator and explain their intention of accessing the repository. It contains Android malware samples as well as Microsoft Windows. The Android section has two sets of data samples. One contains 11,080 malware with size of 5.18 GB, and the other contains 24,317 samples with a size of 47.64 GB.

5.2 Static-related Analysis

As mentioned in Section 2.4.2.1, static analysis is the process of analysing the Android applications' installation file, APK. In this section, we elaborate on the details of our novel static analysis method. During the process of evaluating recent research works, we noticed that Android Intent, more specifically explicit Intent and implicit Intent, is a

[10] https://virusshare.com

semantically rich element in the APK files, and has the potential to be a candidate feature in Android malware detection. Furthermore, to the best of our knowledge, there has been no attempt at comprehensive analysis of Android Intent, which was our motivation to conduct this experiment.

5.2.1 Experiment 1: Evaluating Effectiveness of Android Intent in Malware Detection

The objective of this experiment is to propose Android Intent as a feature for malware detection, and to evaluate its effectiveness by comparing the results to Android permission. In order to achieve that, the following sub-sections focus on more detail in Android Intent, and justify the reason for considering it a feature in Android malware detection. Next, the specification of the chosen algorithm is described. In the evaluation, a Drebin data sample and clean applications from AndroZoo repository are used, and the experiment is conducted on Android Intent and Android permission. Subsequently the results are compared to reach the conclusion.

5.2.1.1 Android Intent

Intent is a complex messaging system in the Android platform, and is considered as a security mechanism to hinder applications from gaining direct access to other applications. Applications must have specific permissions to use Intent. This is a way of controlling what applications can do once they are installed in Android. An Intent-filter (defined in AndroidManifest.xml file) announces the type of Intent the application is capable of receiving.

Applications use Intents for intra-application and inter-application communications. Intra-application communication takes place between activities inside an application. An Android application consists of many activities, each referring to buttons, labels, and texts available on a single page of the application, with which the user interacts. When

interacting with the application, the user moves from activity to activity (i.e. from page to page). Android Intents assist developers in performing interactions among the activities. Furthermore, Intents are used in pushing data from one activity to another, carrying the results at the end of any particular activity (Aftab & Karim, 2014).

Inter-application communication is achieved when applications send messages or data to other applications through Intent. The applications should also be able to receive data from other applications. To receive Intents, applications must define what type of Intent they accept in the Intent section of the AndroidManifest.xml file, as intent-filter. Many past studies (Chakradeo et al., 2013; Feng et al., 2014; Luoshi et al., 2013) refer to this type of Intent. The actual communication between two applications occurs through the Binder, which handles all inter-process communications. The Binder provides the features for binding functions and data between one execution environment and another, as each Android application runs in its own Dalvik environment. The Intent mechanism is considered higher than Binder, hence, it is built on top of Binder.

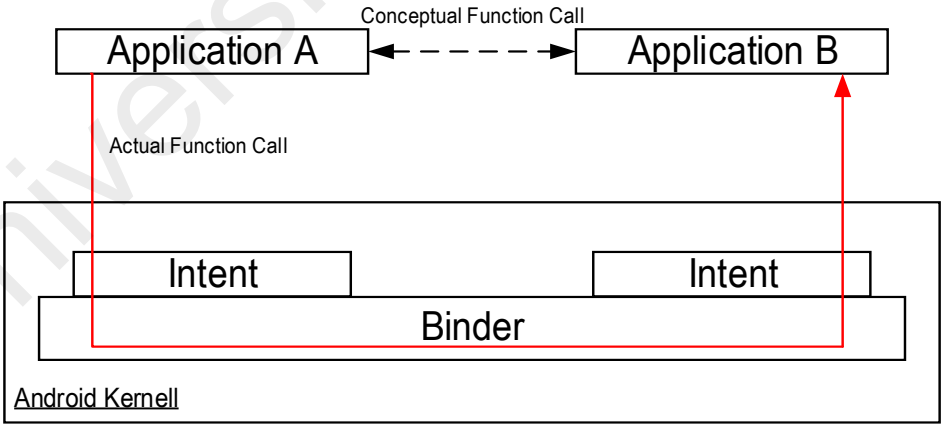


Figure 5.1. Inter-application Communication Using Android Intent and Binder

Figure 5.1 shows the architecture of inter-application communication. The Binder driver manages part of the address space of each application and makes it as read-only; all writing is done by the kernel section of Android. When application A sends a message to application B, the kernel allocates some space in the destination application's memory,

and copies the message directly from the sending application. It then queues a short message to the receiving application, telling it the location of the received message. The recipient can then access that message directly because it is in its own memory space. When application B has finished processing the message, it notifies the Binder driver to mark the memory as free (Hellman, 2013).

There are two types of Intent: explicit and implicit. When developers know exactly what component to use to perform a specific action, they use explicit Intent. This component can be any activity, service, or broadcast receiver. Explicit Intent is used for intra-application and inter-application communications, and developers use this type of Intent to navigate from an activity to another activity inside applications, as well as to exchange messages between applications. For instance, there are some applications that are used for browsing, such as the default browser on the device or Google Chrome. Developers use explicit Intent to request Android to open a link specifically using Google Chrome. Implicit Intent is used by developers when asking Android to open a link, but without specifying the exact target application. In response, Android offers a list of all applications capable of opening a link to the user. Such a list is populated based on the intent-filter section of AndroidManifest.xml files. Table 5.2 shows a sample code of explicit and implicit Intents.

Table 5.2. Sample Code Snippet of Explicit and Implicit Intents

Explicit Intent	String url=”www.yahoo.com”;
	Intent explicit=new Intent(Intent.ACTION_VIEW);
	explicit.setData(Uri.parse(url));
	explicit.setPackage(”com.android.chrome”);
	startActivity(explicit);

Implicit Intent	String url=”www.yahoo.com”;
	Intent implicit=new Intent(Intent.ACTION_VIEW);
	implicit.setData(Uri.parse(url));
	startActivity(implicit);

Table 5.2 shows that implicit Intent uses Intent.ACTION_VIEW to open the specified URL. However, explicit Intent states the exact component's name (in this case com.android.chrome) to open the URL. In our study, our aim is to extract both implicit and explicit Intents and conduct a comprehensive evaluation of their effectiveness in malware detection.

Intents have three components: action, category, and data. The action component describes what kind of action is to be executed by the Intent such as MAIN, CALL, BATTERY LOW, SCREEN ON, and EDIT. Intents specify the category they belong to, such as LAUNCHER, BROWSABLE and GADGET. The data components provide the necessary data to the action component. For instance, the CALL action requires a phone number, and the EDIT action needs a document or HTTP URL to complete.

5.2.1.2 Data Collection and Analysis

For our experiment, we used real-world applications that include both clean and infected applications. We gathered clean applications from Google Play and scanned them with VirusTotal[11] to ensure the cleanness of the applications. The applications collected include both free and paid types, as ProfileDroid (Wei et al., 2012) mentioned that paid applications behave differently from free ones, and it is important to include all such applications. Google Play applications were categorized into 27 main application categories, and the games category had 17 sub-categories. We gathered samples from 24 main application categories (including the game category itself) and 17 games sub-categories to cover a wide variety of applications, as shown in Table 5.3.

[11] www.virustotal.com

Table 5.3. Categories of Gathered Applications

Books & References	Medical	Tools	Games - adventure
Business	Weather	Games - action	Games - strategy
Comics	Travel	Games - card	Games- simulation
Communication	Photography	Games - casino	Games – family
Education	Productivity	Games - casual	Games – racing
Entertainment	Shopping	Games - educational	Games – sports
Finance	Social	Games - music	Games – arcade
Health & Fitness	Sports	Games - puzzle	
Music & Audio	Media & Video	Games -role playing	
News & Magazines	Transportation	Games - word	
Personalization	Live Wallpaper	Games - board	

The clean dataset contains 1,846 applications. Additionally, we used DREBIN (Arp et al., 2014) as infected dataset. It is a collection of 5,560 applications from 179 different malware families. We used our Python code to extract permission and Intent from applications in our dataset. The top 10 permissions of both clean and infected applications are shown in Table 5.4. Google classifies Android permissions into four groups, namely normal, dangerous, signature, and signatureOrSystem (Google, 2014).

Table 5.4. Top 10 Permissions in Clean and Infected Applications

Clean Applications		Infected Applications	
Permissions	Frequency	Permissions	Frequency
INTERNET	98%	INTERNET	98%
ACCESS_NETWORK_STATE	89%	READ_PHONE_STATE	89%
WRITE_EXTERNAL_STORAGE	83%	WRITE_EXTERNAL_STORAGE	67%
WAKE_LOCK	53%	SEND_SMS	54%
READ_PHONE_STATE	52%	RECEIVE_SMS	38%
ACCESS_WIFI_STATE	48%	WAKE_LOCK	38%
GET_ACCOUNTS	42%	READ_SMS	37%
VIBRATE	41%	ACCESS_COARSE_LOCATION	32%
BILLING	39%	ACCESS_FINE_LOCATION	30%
ACCESS_COARSE_LOCATION	24%	READ_CONTACTS	23%

Table 5.4 also shows that five permissions are common, as highlighted, between clean and infected applications, such as, INTERNET, WRITE_EXTERNAL_STORAGE, WAKE_LOCK, ACCESS_COARSE_LOCATION, and READ_PHONE_STATE. However, these applications have five different permissions among the top 10 permissions. Infected applications request SEND_SMS, RECEIVE_SMS and READ SMS permissions, which are classified as dangerous. In fact, WRITE_SMS, which is also dangerous, should be included in the list of the top frequent permissions. It is ranked 11th in our dataset, and it is requested by 22% of infected applications. Therefore, it is evident that infected applications request four SMS-related permissions to have full access to SMS functionality of the devices. In our experiment, 30% of infected applications requested the ACCESS_FINE_LOCATION permission to access the precise location, and 33% of them requested the ACCESS_COARSE_LOCATION permission, which is a common permission, to access a proximate location. In general, the viciousness of infected applications can be gauged through permissions. We also extracted Intent of applications, as shown in Table 5.5, which shows top 10 Intents used in clean and infected applications. It is worth noting that the VIEW Intent was removed from the top 10 Intents, as it is used in all clean and infected applications.

Table 5.5. Top 10 Intents in Clean and Infected Applications

Clean Applications		Infected Applications	
Intents	Frequency	Intents	Frequency
SEND_MULTIPLE	45%	BOOT_COMPLETED	56%
SCREEN_OFF	23%	SENDTO	45%
USER_PRESENT	18%	DIAL	42%
SEARCH	17%	SCREEN_OFF	37%
PICK	10%	TEXT	28%
DIAL	9.5%	SEND	27%
GET_CONTENT	9%	USER_PRESENT	22%
EDIT	8.7%	PACKAGE_ADDED	21%
MEDIA_MOUNTED	8%	SCREEN_ON	18%
BATTERY_CHANGED	7%	CALL	10%

Malicious applications wait for BOOT_COMPLETED to start their malicious activity. CALL and DIAL are used for making phone calls. CALL requires CALL_PHONE permission, whereas DIAL does not require such permission. As presented in Table 5.5, DIAL is used more than CALL, which allows the malicious application to make a premium phone call without the user's knowledge.

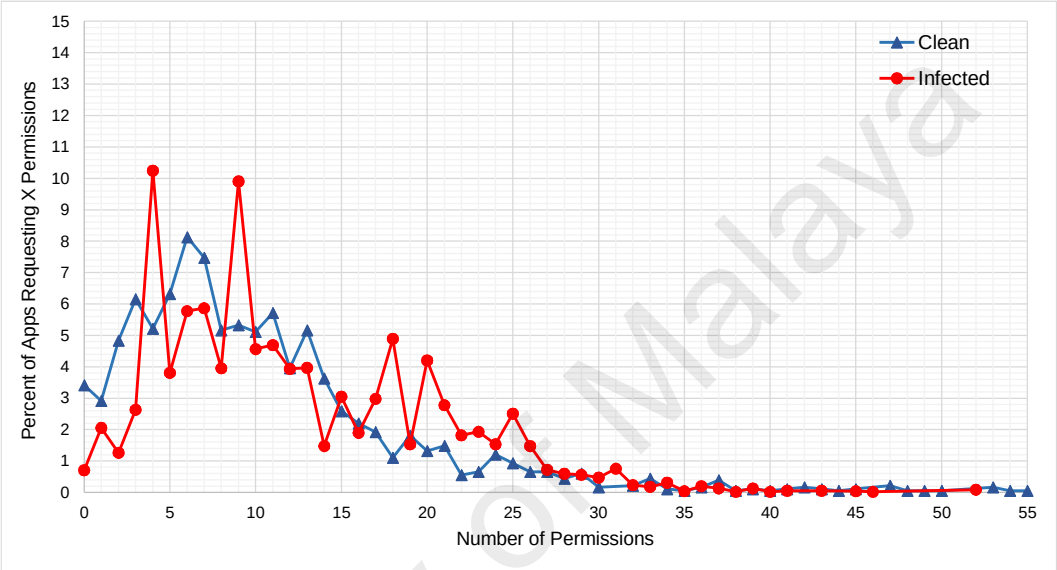


Figure 5.2. Percent of Applications That Request Specific Number of Permissions

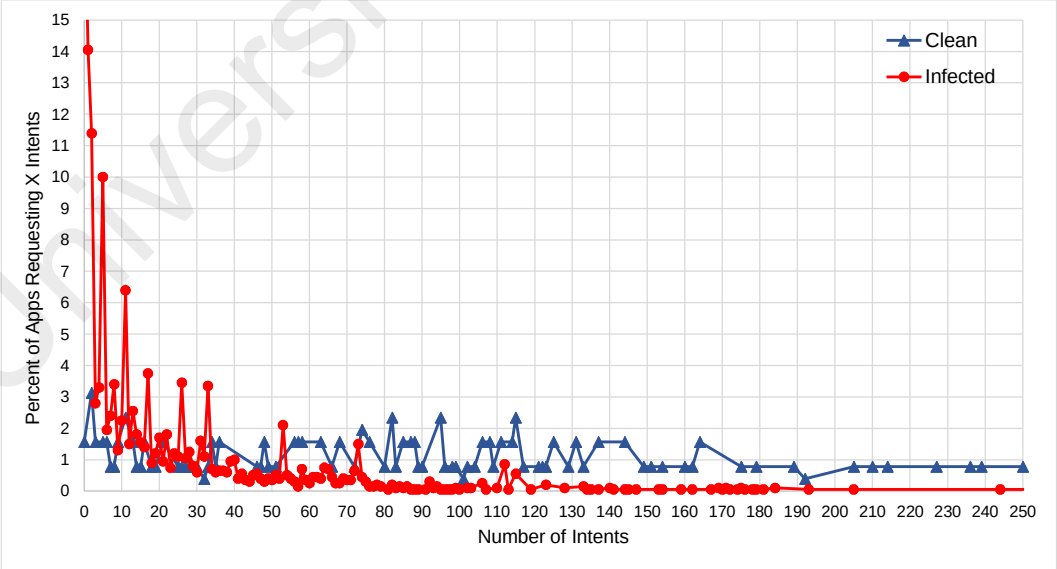


Figure 5.3. Percent of Applications That Request Specific Number of Intents

Figure 5.2 shows the percentage of applications that requested permissions (clean and infected) in two datasets. The graph shows that infected applications request more

permissions as there are spikes at multiple points in the figure. Furthermore, only 2% of clean applications requested between 35-55 permissions, compared to 7% of infected applications. This is indicative of the vicious intentions of infected applications.

Similarly, Figure 5.3 shows the percentage of applications that requested Intents (implicit and explicit) in two datasets. When comparing Figure 5.2 and Figure 5.3, the difference between their x-axis is obvious. While permissions have a maximum number of 55, the number of Intents ends at 250. The wide difference is due to the fact that developers use Intents much more frequently than permissions in the code to perform actions.

Intent and permission are potentially useful features for Android malware detection. However, according to Moonsamy et al. (Moonsamy et al., 2013b), there are requested permissions as well as required permissions. It is possible that actual permissions used by applications are different from the requested permissions that are sent to the user for approval. However, Intent reflects the actual intentions of applications resulting directly from activities. This indicates that Intent is more effective for malware detection.

5.2.1.3 The Architecture

Figure 5.4 shows the architecture for our experiment, AndroDialysis.[12] The top level of the architecture, the Android application framework, refers to applications installed on the device. The detector module performs the main task of detection. It consists of four sub-modules: decompiler, extractor, intelligent learner, and decision maker. The system sends the results to users through the graphical user interface. The following sections describe four sub-modules in more detail.

[12] **Andro**id **D**eep **I**ntent An**alysis**

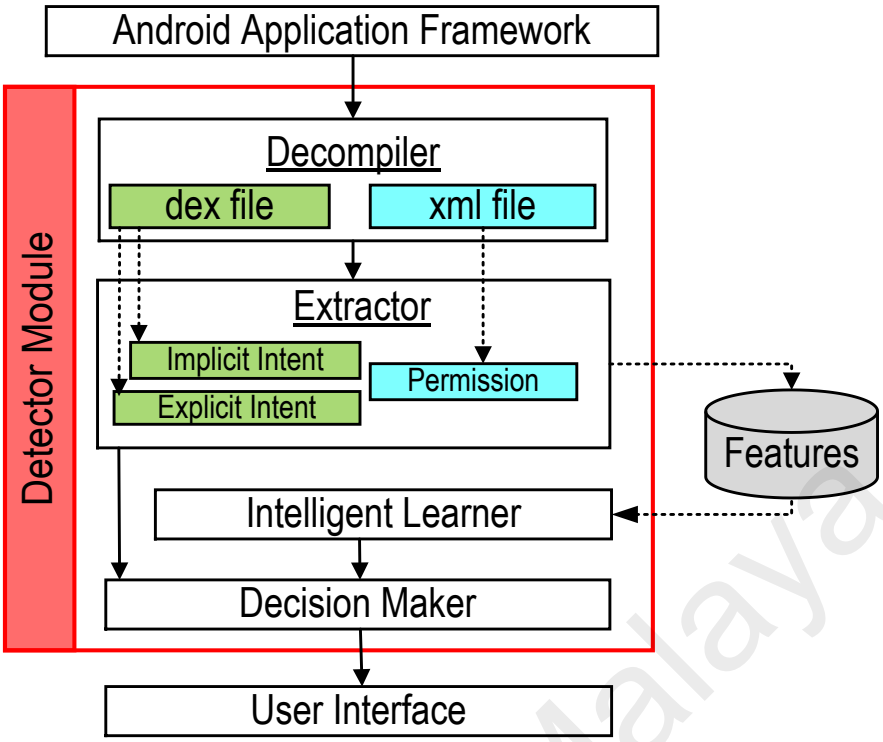


Figure 5.4. Overview of AndroDialysis

(a) ***Decompiler***

The decompiler sub-module is responsible for dissecting the APK files and decoding their components. Every APK file has various components. AndroidManifest.xml is a scrambled file and needs to be decoded in order to make it readable. Similarly, the DEX file is a Java source code compiled in Dalvik format and needs to be decompiled. After decompilation, the produced file is not a pure Java code, but it is easy to read. We used Apktool for decompiling Android files, as it utilizes the latest Android SDK, which is better in optimizing files (Wiśniewski, 2010). Decompiling files results in readable AndroidManifest.xml files and generates smali versions of Java code.

(b) ***Extractor***

The extractor sub-module extracts explicit Intent, implicit Intent, and permission from Java code and AndroidManifest.xml file for processing in subsequent sub-modules. The BeautifulSoup package of the Python language is used to extract permission section from the AndroidManifest.xml file (Richardson, 2007). In order to extract Intents from Java

code, we used Androguard to reverse DEX files and get Intents (implicit and explicit) from the code. The extracted data are stored in a feature database for use in the next process. Furthermore, a copy of the data is sent to the decision maker sub-module for determining maliciousness of the data, which will be discussed later.

(c) ***Intelligent Learner***

This sub-module takes data from the features database and uses the Bayesian Network algorithm to learn the pattern of the data. It then sends the output model to the decision maker sub-module. The Bayesian Network algorithm (Friedman et al., 1997) was chosen to evaluate AndroDialysis because it has been successfully used in real-world problems. Cohen et al. (Cohen et al., 2003) for example used the Bayesian Network in human facial expression recognition and achieved a good outcome. It is a dual-process algorithm; it first learns network structure, and then it learns probability tables. The Bayesian Network uses local score metrics to learn the network structure of data. It is considered an optimization problem in which the quality of the network is optimized. To calculate the local score, the Bayesian Network employs search algorithms. Once the network structure of the data has been learnt, the Bayesian Network utilizes estimators to learn the probability tables (Bielza & Larrañaga, 2014). Two widely used estimators are the simple and multinomial estimator. The aforementioned two steps are defined as follows:

Suppose that $V = \{x_1, \dots \dots, x_k\}, k \geq 1$ is a set of variables. Bayesian Network B over V is a network structure B_S that is a directed acyclic graph known as DAG over the set of variables V. It is also a set of probability tables $B_P = \{p(v|pa(v))|v \in V\}$ where $pa(v)$ is the set of parents of v in B_S. Finally, a Bayesian Network represents a probability distribution $P(V) = \prod_{v \in V} p(v|pa(v))$.

Compared to other algorithms, the Bayesian Network has the following advantages:

a) It is a fast algorithm with low computational overhead once trained.

b) It has the ability to model both expert and learning systems with relative ease. It integrates probabilities into the system. It is also considered a performance-tuning tool, but without incurring computational overhead.

c) Many outstanding real-world applications have used this algorithm and have performed comparably well against other state-of-the-art algorithms (Bielza & Larrañaga, 2014).

As mentioned above, Bayesian Networks are collections of directed acyclic graphs (DAGs), where the nodes are random variables, and where the arcs specify the independence assumptions between these variables. It is difficult to find the Bayesian Network that best reflects the dependence relationship in a database of cases because of the large number of possible DAG structures, given even a small number of nodes to connect. As a result, researchers have developed various search algorithms to overcome this problem. In this paper, we use four search algorithms for our experiments: K2, Geneticsearch, HillClimber, and LAGDHillClimber algorithms.

The K2 algorithm heuristically searches for the most probable belief network structure in a given database of cases, which includes different combinations of values for attributes (C. Ruiz, 2005). The **Geneticsearch** algorithm uses the genetic algorithm to find the optimum result in a Bayesian Network. The algorithm is based on the mechanics of natural selection and natural genetics. Although it is capable of solving complex problems, it is a time consuming algorithm for some data (see Table 5.9) (L. J. Yan & Cercone, 2010). It combines survival of the fittest among string structures with a structured, yet randomized, information exchange to form a search algorithm that under certain conditions evolves into the optimum with a probability that is arbitrarily close to one (Larrañaga et al., 1996).

The **HillClimber** search algorithm starts learning by initializing the structure of the Bayesian Network. Unlike previous algorithms that potentially get stuck in the search process, the Hill Climber solves that problem (Chickering et al., 1995). Each possible arc from any node is then evaluated using leave-one-out cross validation to estimate the accuracy of the network with that arc added. If no arc shows any improvement in accuracy, the current structure is determined. An arc that has the most improvement is retained, but the node the arc points to is removed. This process is repeated until there is just one node remaining, or no arc can be added to further improve upon the classification accuracy (Jo et al., 2011). The **LAGDHillClimber** search algorithm uses a Look Ahead Hill Climbing algorithm. Unlike Hill Climber, it does not calculate a best arc (by adding, deleting or reversing an arc), but considers a sequence of best arcs instead of considering the best arc at each step. As it is very time consuming to find the best sequence among all the possible arcs, it must first find a set of good arcs and then find the best sequence of arcs among them (Salehi & Gras, 2009). This improvement over the Hill Climber algorithm results in better performance (see Table 5.6).

We evaluate the performance of the Bayesian Network using k-fold cross validation. In this method, the dataset is divided into *k* subsets, and the holdout method is repeated *k* times. Each time, one of the *k* subsets serves as the test set and the other *k*-1 subsets are compiled to form a training set. Then, the average error across all *k* trials is computed. The advantage of this method is that it matters less how the data are divided. Every data point gets to be in a test set exactly once, and in a training set *k*-1 times. The variance of the resulting estimate is reduced as *k* increases (Feizollah et al., 2013). Specifically, a 10-fold option is used, which is described as applying the classifier to data 10 times and every time the dataset is divided into 90:10 groups - 90% of data used for training, and 10% used for testing, which is widely used among researchers (Damopoulos et al., 2012). At the end, this sub-module produces a model that is based on available data in the features

database that is used for detection purpose. It is worth noting that the intelligent learner is constantly learning from the data added to the features database.

(d) ***Decision Maker***

The decision maker sub-module is responsible for determining whether the data are clean or malicious. It receives two sets of data from the extractor and the intelligent learner sub-modules. A set of data from the intelligent learner sub-module contains a produced model based on the collection of data in the features database. The model is then used to vet the data received from the extractor sub-module. Another set of data that is received from the extractor sub-module contains extracted data of one application. The decision maker sub-module utilizes the model to determine the maliciousness of the application. The final decision is passed to the user interface module, which prepares an appropriate message for the user and presents it through the graphical user interface. This design of the decision maker sub-module ensures faster detection and higher performance, as it was adopted by Shabtai et al. (Shabtai et al., 2014).

5.2.1.4 Results

In this section, we discuss our results and findings. It is important to restate that the purpose of this experiment is to study the effectiveness of Android Intent (implicit and explicit) in malware detection, and not malware detection *per se*. We present the results from experiments conducted on permissions, Intents, and both in Android malware detection. Additionally, to get a better assessment of the current development of Android Intent, we analyse our datasets.

(a) ***Intent Analysis and Attacks***

We analyse Intents in our datasets from the security standpoint to assess the current status or importance of Intents. As mentioned in Section 5.2.1.1, implicit Intent does not specify its destination component. However, it is offered to entities that can receive a specific

type of Intent. Therefore, when an application sends an implicit Intent, there is no guarantee that the Intent will be received by the intended recipient. A malicious application can intercept an implicit Intent simply by declaring an intent-filter (in AndroidManifest.xml file) with all the actions, data, and categories listed in the Intent. This situation (unauthorized Intent receipt) causes the malicious application to gain access to all the data in any matching Intent, resulting in activity hijacking (Chin et al., 2011).

In the collected dataset, infected applications declare intent-filter 7.5 times more often than clean applications. On average, each clean application declares 1.18 intent-filters, whereas each infected application declares 1.61 intent-filters. Thus, it is evident that infected applications tend to intercept Intents using intent-filters until they succeed in hijacking the activities.

In view of this threat, it is suggested that developers use explicit Intent so that the recipient is clearly specified in order to hinder malicious applications from hijacking the activities. We have analysed our dataset with regard to this threat, and found that 28.78% of Intents used were implicit and 71.22% were explicit. In general, developers are doing what is appropriate; nevertheless, it is essential to remain vigilant, as attackers are known to frequently change their attack plans.

(b) ***Experimental Results***

This experiment was performed on a Sony Xperia Z3 Compact device, model D5803. It is running Android Marshmallow, version 6.0.1, with the latest updates. The device has 2GB of RAM and 16GB of storage.

We aim to answer the following questions. **A.** Is Intent a plausible feature for Android malware detection? **B.** What are best configurations in the Bayesian Network that produce the best results? **C.** How effective is Android Intent compared to Android permission?

i Effectiveness

We employed the Bayesian Network with different configurations for our experiment. As discussed earlier, the Bayesian Network uses a search algorithm for calculating the local score metrics, and an estimator algorithm for learning the probability table. In order to achieve the best results, we experimented with different configurations, and the results are presented in Table 5.6. It shows results of permission and Intent with simple estimator and multinomial estimator algorithms; and K2, Geneticsearch, HillClimber, and LAGDHillClimber as search algorithms.

Table 5.6. Results of Android Permission and Android Intent Experiments

	Android Permission				Android Intent			
	Simple Estimator		Multinomial		Simple Estimator		Multinomial	
	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR
K2	82%	18%	24%	76%	89%	11%	19%	81%
Geneticsearch	83%	17%	Null	Null	91%	9%	Null	Null
HillClimber	82%	18%	24%	76%	89%	11%	19%	81%
LAGDHillClimber	83%	17%	Null	Null	91%	9%	Null	Null

The results of the experiments reflect the performance of our method. Detection rate, also known as a true positive rate (TPR), is the probability of correctly detecting an instance as a malware. In contrast, false positive rate (FPR) is another measurement that is defined as wrongly detecting normal traffic as being infected. The higher the TPR, the better the result. Conversely, the lower the FPR, the better the result. The best results were obtained by combining a simple estimator and Geneticsearch, and a simple estimator and

LAGDHillClimber, both combinations achieving 83% TPR for Android permission, and 91% for Android Intent. We conducted our experiment in 30 iterations. As the number of iterations increased, the system learnt the pattern of the data more accurately. Figure 5.5 shows the TPR and the false positive rate for each iteration of the experiment.

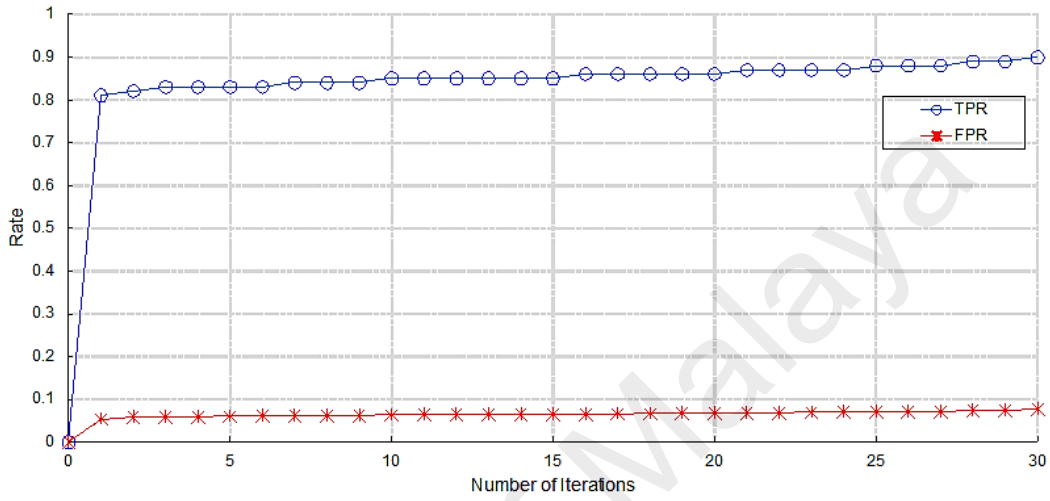


Figure 5.5. True Positive Rate versus False Positive Rate for 30 Iterations

Figure 5.5 shows that TPR increased from just above 80% to 90% as the number of iterations increased. However, the FPR did not follow the same rate of increase as the TPR. It started from 6% and increased to 9%, which is considered as a good result, considering that the TPR increased by 10%.

Additionally, we conducted experiments for each malware family to assess the effectiveness of Android Intent for an individual family. The results are tabulated in Table 5.7 and the best result for each family is highlighted. The experiments were conducted on the families with the highest number of malware samples in our dataset. As our previous results with a multinomial algorithm were not encouraging, we used a simple estimator for this experiment. The lowest detection rate, among all families, belongs to the DroidKungfu family for either K2 or HillClimber algorithm. This malware gains root access in the device and installs an application called legacy that pretends to be a legitimate Google Search application bearing the same icon. The DroidKungfu then

performs its malicious activities through the legacy application (Jiang, 2011). We believe that this strategy makes it trickier to detect, as malicious activities are performed by an agent application other than the main one. Other malware families showed relatively high detection results.

Table 5.7. The results of Android Intent Experiments for Each Malware Family

Malware Family	Measurements	K2	Geneticsearch	HillClimber	LAGD HillClimber	Number of malware
FakeInstaller	TPR	85.78%	84.02%	84.91%	84.02%	925
	FPR	14.21%	15.97%	15.08%	15.97%	
DroidKungFu	TPR	76.41%	76.14%	76.41%	76.14%	667
	FPR	23.58%	23.85%	23.58%	23.85%	
Plankton	TPR	79.59%	79.59%	79.34%	79.54%	625
	FPR	20.40%	20.40%	20.65%	20.45%	
Opfake	TPR	93.06%	93.06%	92.76%	93.06%	613
	FPR	6.93%	6.93%	7.23%	6.93%	
GinMaster	TPR	77.35%	77.35%	77.15%	77.58%	339
	FPR	22.64%	22.64%	22.84%	22.41%	
BaseBridge	TPR	81.96%	81%	83%	80.17%	330
	FPR	18.03%	19%	17%	19.82%	
Iconosys	TPR	76.74%	76.87%	76.74%	76.87%	152
	FPR	23.25%	23.12%	23.25%	23.12%	
FakeDoc	TPR	81.89%	81.65%	81.89%	81.65%	132
	FPR	18.10%	18.34%	18.10%	18.34%	
Geinimi	TPR	87.39%	87.39%	79.91%	80.55%	92
	FPR	12.60%	12.60%	20.08%	19.44%	
					Total	3,875

It is necessary to verify that Android Intent is in fact an effective feature, and that our results were not just a coincidence. Therefore, we conducted experiments using both features (Android permissions and Android Intents). This was essential to show that the features are not overlapping, and Android Intent can really increase the detection rate.

Table 5.8 represents the results of the experiments on the combination of Android Permissions and Android Intents. Not only do the results show that Android Intent (explicit and implicit) is an effective feature, it also boosts other features (i.e. Android permissions) in malware detection.

Table 5.8. Results of Experiments Using Both Permissions and Intents

	Simple Estimator	
	TPR	FPR
K2	95.5%	4.4%
Geneticsearch	95.4%	4.5%
HillClimber	95.5%	4.4%
LAGDHillClimber	95.4%	4.5%

It is worth noting that the choice of Android permissions in this study is based on the fact that this feature has been widely explored and its importance and effectiveness has been established. Feizollah et al. (Feizollah et al., 2015) conducted an extensive study on Android features. Among static features, Android permission is the most widely used feature. Various approaches have been taken to analyse Android permissions. Some uthors used permissions to evaluate applications and rank them based on possible risk (Au et al., 2012; Grace, Zhou, Zhang, et al., 2012; Pandita et al., 2013; Peng et al., 2012). Numerous studies simply extracted permissions and utilized machine learning to detect malicious application, (Aung & Zaw, 2013; Samra et al., 2013; Borja Sanz, Santos, Laorden, Ugarte-Pedrero, Bringas, et al., 2013; Suleiman Y Yerima et al., 2014). Some researchers argue that merely analysing requested permissions is not sufficient for detecting malicious applications (C. Y. Huang et al., 2013; Moonsamy et al., 2013b). They analysed the used permissions in addition to the requested permissions in order to detect malware. AppGuard (Backes et al., 2013) has gone one step further and has extended Android's permission system to alleviate current vulnerabilities. They claim that their system is a practical extension for the Android permission system as it is

possible to use it on devices without any modification or root access. As a result, Android permissions is a strong candidate for this paper in order to compare it with Android Intents.

ii Efficiency

Besides evaluating the effectiveness of our system, we calculated the time taken by each combination to produce the results of Table 5.6, as shown in Table 5.9.

Table 5.9. Time Taken to Produce Results (seconds)

	Android Permission		Android Intent	
	Simple Estimator	Multinomial	Simple Estimator	Multinomial
K2	0.06	0.89	0.01	0.07
Geneticsearch	2.86	Null	0.91	Null
HillClimber	0.02	0.87	0.01	0.07
LAGDHillClimber	0.05	Null	0.05	Null

Based on Table 5.9, results in Android permission are produced faster when the simple estimator and HillClimber are combined. With regard to Android Intent, combining the simple estimator with LAGDHillClimber achieved a TPR of 91% in less time than Geneticsearch.

In addition, we show the Receiver Operating Characteristic (ROC) curve for the best results of permission and Intent. The ROC curve is normally used to measure performance in detecting intrusions. It indicates how the detection rate changes, as the internal threshold is varied to generate more or fewer false alarms. It plots intrusion detection accuracy against false positive probability. ROC curves signify the tradeoff between false positive and true positive rates, which means that any increase in the true positive rate is accompanied by a decrease in the FPR. As the ROC curve line is closer to the left-hand

border and the top border, it indicates that it produces the best results among other curves. The ROC curves for Android permission and Android Intent are shown in Figure 5.6.

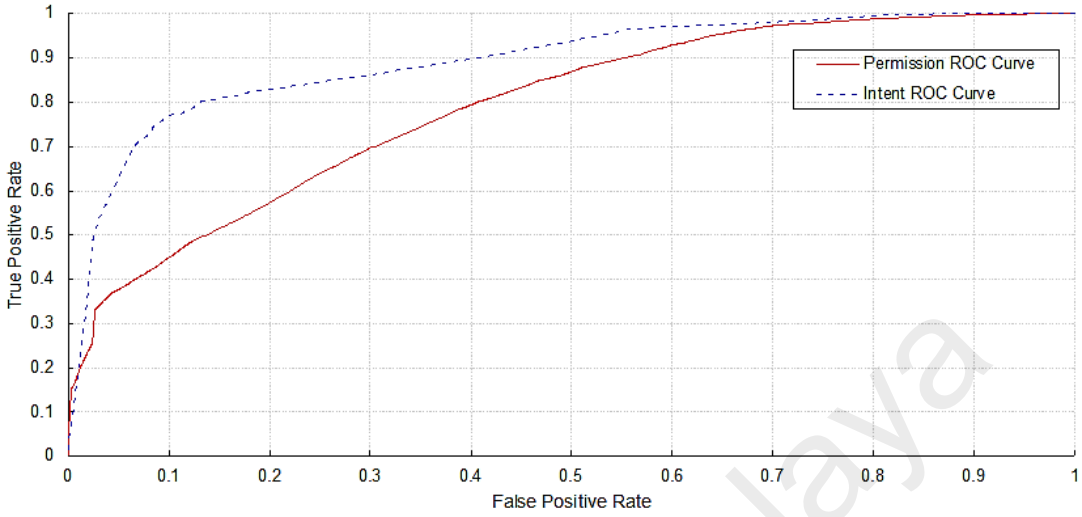


Figure 5.6. ROC Curve for Android Permission and Android Intent

The ROC curves are difficult to compare, as they seem to be almost similar under some situations, therefore, the area under the curve (AUC) is used to measure the accuracy of detection. An area of 1 means a perfect result, while an area of 0.5 is a worthless result. The AUC point system is as follows: 0.90 - 1.00 = excellent (A); 0.80 - 0.90 = good (B); 0.70 - 0.80 = fair (C); 0.60 - 0.70 = poor (D); and 0.50 - 0.60 = fail (F). The AUC of Android permissions is 0.7897, and Android Intent is 0.8929. This shows that Android Intent performed better.

5.2.1.5 Conclusion

This experiment showed that Android Intent is in fact an effective feature for mobile malware detection. Moreover, the combination of Intent and permission achieved higher results, which indicates that Android Intent is also considered complementary to other features. As this is the first experiment on implicit and explicit Intent, we could not compare our results to other works; however, comparing the results with Android permission showed that Android Intent (implicit and explicit) produces good results in mobile malware detection.

5.3 Dynamic-related Analysis

The dynamic analysis complements static analysis in order to have a comprehensive analysis and detection system. This section consists of two experiments. The first one examines the pool of available network traffic features to identify the best ones by using feature selection algorithms. Each algorithm is described in terms of functionality and its advantages. The second experiment investigates machine learning classifiers to find the best one with the highest results in terms of accuracy.

5.3.1 Android Malware Network Traffic

Two of the most important behaviours used in the dynamic analysis are system calls and network traffic. When an application is running, it should request some operations from an operating system, such as read, write, or open, in order to perform tasks. Therefore, if an application is calling too many functions, it would sound suspicious. Crowdroid (Burguera et al., 2011) focused on collecting system calls and processing them to detect an anomaly. As the Android operating system has the Linux kernel, collecting system calls is a complicated task as described in (Burguera et al., 2011). In most cases, the device must be rooted, which means disabling part of the operating system's security architecture, consequently leaving the device more vulnerable against threats. Network traffic is collected on the device by an application such as tPacketCapturePro.[13] Therefore, we present AndroPsychology, an experiment on analysing the network behaviour of the Android application.

5.3.2 Description of the Experiment

AndroPsychology is presented in Figure 5.7. For these experiments, 50,000 malware samples as well as 50,000 clean applications were specifically acquired from AndroZoo.

[13] https://play.google.com/store/apps/details?id=jp.co.taosoftware.android.packetcapturepro

The network traffic process was captured by running each application for 20 minutes and collecting the generated network traffic in the format of a PCAP file.

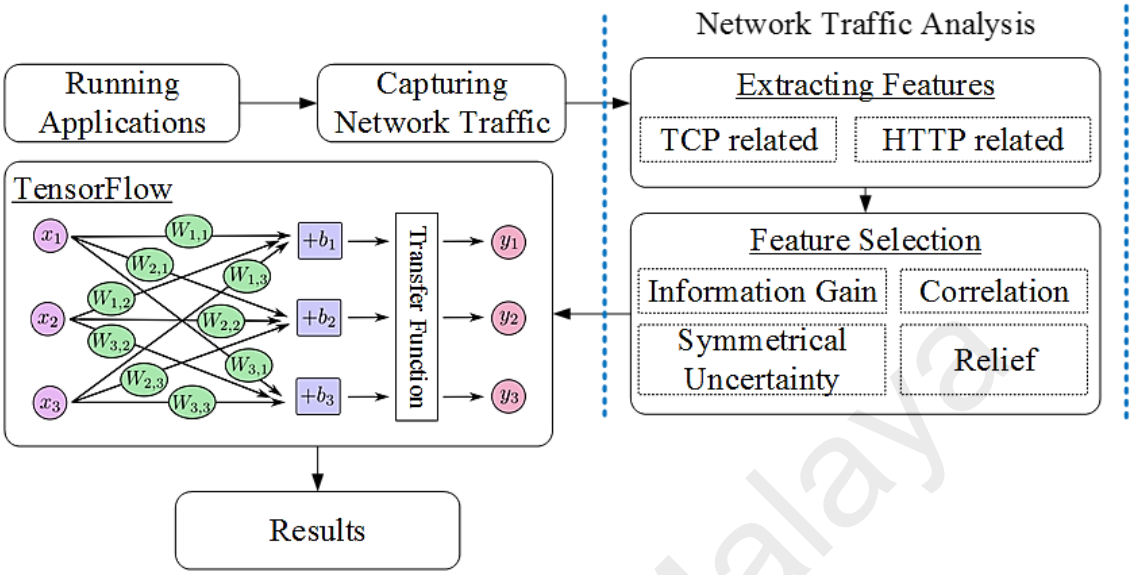


Figure 5.7. The AndroPsychology Architecture

The first process is extracting features that include TCP related and HTTP related features. The Tshark command line program was used to extract 30 features. The second process is selecting the best features by using feature selection algorithms. This process is explained as experiment 2 in Section 5.3.3.

5.3.3 Experiment 2: Selecting Best Network-related Features

This experiment deals with the selection of the best network-related features. Network traffic contains hundreds of features and protocols. It is essential to choose the appropriate features from those available.

The problem of identifying applications by analysing the network traffic they generate has received significant attention in the literature over the past decade and a half. This is for the following reason. With growing adoption of traffic compression, as well as stronger internet privacy legislation, HTTP payloads are increasingly becoming inaccessible to traffic monitors (Sicker et al., 2007; White et al., 2013). consequently, analysis that relies on TCP/IP headers is most useful (Alan & Kaur, 2016).

However, Android malware families have used TCP and HTTP to perform malicious activities. TrendMicro discovered that 400 Android applications inside Google Play were infected with DressCode malware. As many as 500,000 unique users downloaded infected applications. Successful installation enables the malware to connect to its command and control (C&C) servers via TCP protocol. Whenever the C&C responds back, an attacker can create a TCP connection between himself and the infected device. That link empowers the attacker to issue commands to the infected device (Duan, 2016). Another example is Fake Installer malware that steals information from the device, and sends it to a specific server. The communications with the external server are transmitted over HTTP. This includes some commands that are sent to the infected device in order to steal specific information (TrustGo, 2012).

The aforementioned malware families are small examples from hundreds of Android malware that use TCP and HTTP as communication channels between devices and attackers. Among related research works, some have used HTTP, while others have used TCP. Table 5.10 lists related works and shows whether they analysed TCP or HTTP.

Table 5.10. Comparison of Different Approaches in Related Works

Reference	TCP	HTTP	Reference	TCP	HTTP
(Shabtai et al., 2012)	√	-	(Arora et al., 2014)	√	-
(Dai et al., 2013)	-	√	(Conti et al., 2015)	√	-
(Tongaonkar et al., 2013)	-	√	(X. Wu et al., 2015)	-	√
(Shabtai et al., 2014)	√	-	(Aresu et al., 2015)	-	√
(Narudin et al., 2016)	√	√	This study	√	√

Table 5.10 shows that except one work, others selected TCP or HTTP, which is not comprehensive. Narudin et al. selected both TCP and HTTP; however, their experiment was done on a MalGenome data sample with 1,260 applications (Narudin et al., 2016). This work analyses TCP and HTTP of 50,000 applications. Thus, the focus of this study is on TCP and HTTP protocols for malware detection.

As mentioned previously, the collection of network traffic was performed using 50,000 clean applications and 50,000 malware samples gathered from AndroZoo. With the help of Wireshark documentation[14], 30 features are selected and presented in Table 5.11. Features begin with TCP or HTTP, which represents their respective category. In addition, the table assigns a number to each feature, which is used in the following section to show the results of feature selection algorithms.

[14] https://www.wireshark.org/docs/dfref/

Table 5.11. Extracted Network-related Features

1	tcp.analysis.bytes_in_flight	16	http.content_length
2	tcp.analysis.keep_alive	17	http.leading_crlf
3	tcp.analysis.keep_alive_ack	18	http.next_request_in
4	tcp.analysis.push_bytes_sent	19	http.next_response_in
5	tcp.analysis.retransmission	20	http.prev_request_in
6	tcp.checksum.status	21	http.prev_response_in
7	tcp.dstport	22	http.proxy_connect_port
8	tcp.hdr_len	23	http.request_in
9	tcp.len	24	http.request_number
10	tcp.options.rvbd.trpy.dst.port	25	http.response.code
11	tcp.port	26	http.response_in
12	tcp.window_size	27	http.response_number
13	http.chat	28	http.ssl_port
14	http.chunk_size	29	http.subdissector_failed
15	http.chunkd_and_length	30	http.te_and_length

In total, there are 18 HTTP and 12 TCP features. The next section describes feature selection algorithms.

5.3.3.1 Feature Selection Algorithms

Researchers have been using several feature selection algorithms for years. Section 2.4.1 mentioned the benefits of feature selection, such as reducing the dimensionality of databases, saving time and cost of experiments, and yielding more accurate results by removing noisy data. Such benefits also apply in choosing network-related features.

There are two main types of feature selection mechanisms, known as filter approach and wrapper approach. Filter approaches use an evaluation function that relies only on the properties of the data. Wrapper approaches use learning algorithms to estimate the value of a given subset. In the former, the measure of significance or relevance is defined independently of the learning algorithm, while in the latter, the measure of significance

is directly defined from the learning algorithm. In this study, we focus on the ranker-based filter technique, as there is more advantage in using the filter approach as compared to that of the wrapper approach (Kojadinovic & Wottka, 2000). The filter method is fast and simple, which makes it more suitable for high dimensional data (L. Yu & Liu, 2003) than wrapper methods, because when the dimensionality becomes very large, the filter method has lesser computational time complexity. Therefore, we chose four algorithms for this experiment, namely correlation-based feature selection, symmetrical uncertainty, information gain, and relief algorithms.

A correlation-based algorithm evaluates the worth of an attribute by measuring the correlation between it and the class. Nominal attributes are considered on a value by value basis by treating each value as an indicator. An overall correlation for a nominal attribute is arrived at via a weighted average. So, an indicator for the value of a nominal attribute is a numeric binary attribute that takes on the value of 1 when the value occurs in an instance and 0 otherwise (Hall, 1999).

Symmetrical uncertainty evaluates features individually by measuring their symmetrical uncertainty with respect to the class. The symmetrical uncertainty measure is based on the concept of entropy, which is a measure of the uncertainty of a random variable. The entropy of a variable X is defined as $H(X) = -\sum P(x_i)\log_2(P(x_i))$. The amount by which the entropy of X decreases reflects additional information about X provided by Y, given by $H(X|Y) = -\sum_i P(y_i)\sum_j P(x_i|y_j)\log_2(P(x_i|y_j))$. Where $P(x_i)$ is the prior probabilities for all values of X and $P(x_i|y_j)$ is the posterior probabilities of X given the values of Y.

The values of symmetrical uncertainty are within the range of $[0,1]$ with the value 1, indicating that knowledge of either one of the values completely predicts the value of the other, and the value 0, indicating that X and Y are independent. The symmetrical

uncertainty value has two main functions: (1) to remove the features with symmetrical uncertainty below the threshold and (2) to calculate every feature's weight that is to be used to guide the initialization of the population for genetic algorithms in a memetic framework. The feature with larger symmetrical uncertainty value gets a higher weight. The feature with the lower symmetrical uncertainty value is removed (Senthamarai Kannan & Ramaraj, 2010).

Information gain also uses entropy to select the best features. It measures the amount of information about class prediction in bits, if the only information available is the presence of a feature and the corresponding class distribution. Concretely, it measures the expected reduction in entropy, which is the uncertainty associated with a random feature.

A relief algorithm selects relevant features using a statistical method. Relief does not depend on heuristics; it is accurate even if features interact, and it is noise-tolerant. It requires only linear time in the number of given features and the number of training instances, regardless of the target concept's complexity. It randomly samples a given number of instances from the training set and updates the relevance estimation of each feature based on the difference between the selected instance and the two nearest instances of the same and opposite classes. Moreover, it evaluates the worth of an attribute by repeatedly sampling an instance and considering the value of the given attribute for the nearest instance of the same and different classes.

5.3.3.2 Results and Discussion

The results of applying the algorithms on sets of features are tabulated in Table 5.12, which shows the feature number and the weight for each algorithm. Weight is result of feature selection algorithms and ranges from 0 (lowest score) to 1 (highest score).

Table 5.12. Results of Network-related Feature Selection Algorithms

Info Gain		Correlation		Relief		Symmetrical	
Feature	Weight	Feature	Weight	Feature	Weight	Feature	Weight
11	0.232647	13	0.19405	9	0.012695531637	12	0.10994
12	0.211327	4	0.14945	11	0.01137271429	7	0.10994
7	0.211327	9	0.13003	7	0.0086361702	11	0.10809
13	0.042779	1	0.09362	12	0.0086361702	1	0.04743
1	0.033345	6	0.0815	1	0.00761390791	4	0.0445
4	0.028688	20	0.06425	13	0.00573305723	13	0.04052
9	0.020678	21	0.04955	14	0.005380952380	9	0.03103
16	0.0138	8	0.04225	25	0.005356671381	16	0.02603
25	0.006304	29	0.03155	4	0.004432393299	25	0.0156
6	0.005549	23	0.03132	5	0.004095238095	6	0.00945
8	0.005549	30	0.02635	23	0.003850767085	8	0.00945
20	0.002463	26	0.0258	26	0.0032215935879	20	0.00675
21	0.001686	11	0.02501	30	0.002761904761	21	0.00468
29	0.001344	14	0.02236	15	0.001732442181	29	0.00358
30	0.000896	16	0.022	29	0.001666666666	30	0.00242
17	0	27	0.01777	24	0.001166067342	17	0
28	0	25	0.01745	16	0.000805223125	28	0
5	0	24	0.01743	27	0.000652023809	5	0
3	0	15	0.01717	8	0.0002404761904	3	0
26	0	7	0.0158	21	0.000061723602	26	0
2	0	12	0.0158	28	0.0000367857142	2	0
27	0	5	0.01527	20	0.0000291005291	27	0
24	0	28	0.00837	6	0	24	0
14	0	22	0.00812	10	0	14	0
19	0	17	0.0076	2	0	19	0
18	0	2	0	19	0	18	0
22	0	3	0	18	0	22	0
23	0	19	0	3	0	23	0
10	0	18	0	17	-0.00000000	10	0
15	0	10	0	22	-0.000071428	15	0

Based on the results, we select the top 10 features of each algorithm for our analysis. A closer look at the results reveals that some features are repeated in the results of all four algorithms. For example, feature number 13, which is http.chat, is among the top 10 features in all algorithms. This repetition qualifies this feature for inclusion in the final selected dataset. The 10 chosen features based on the number of repetitions are shown in Table 5.13.

Table 5.13. Top 10 Features for Final Dataset

Rank	Feature number	Feature	Rank	Feature number	Feature
1	13	http.chat	6	11	tcp.port
2	1	tcp.analysis.bytes_in_flight	7	12	tcp.window_size
3	4	tcp.analysis.push_bytes_sent	8	7	tcp.dstport
4	9	tcp.len	9	6	tcp.checksum.status
5	25	http.response.code	10	16	http.content_length

Reducing the dimensionality of the dataset (from 30 to 10 features), enables us to analyse each feature more thoroughly. Such analysis ensures that the distribution of data is appropriate for classification algorithms. For instance, features with zero values throughout are not suitable for classification purpose, as they mislead algorithms. Figure 5.8 shows the distribution of data in each of the top 10 features. It is worth noting that the x-axis in all figures represents the data in the dataset. Moreover, the y-axis shows the range of data in that particular feature.

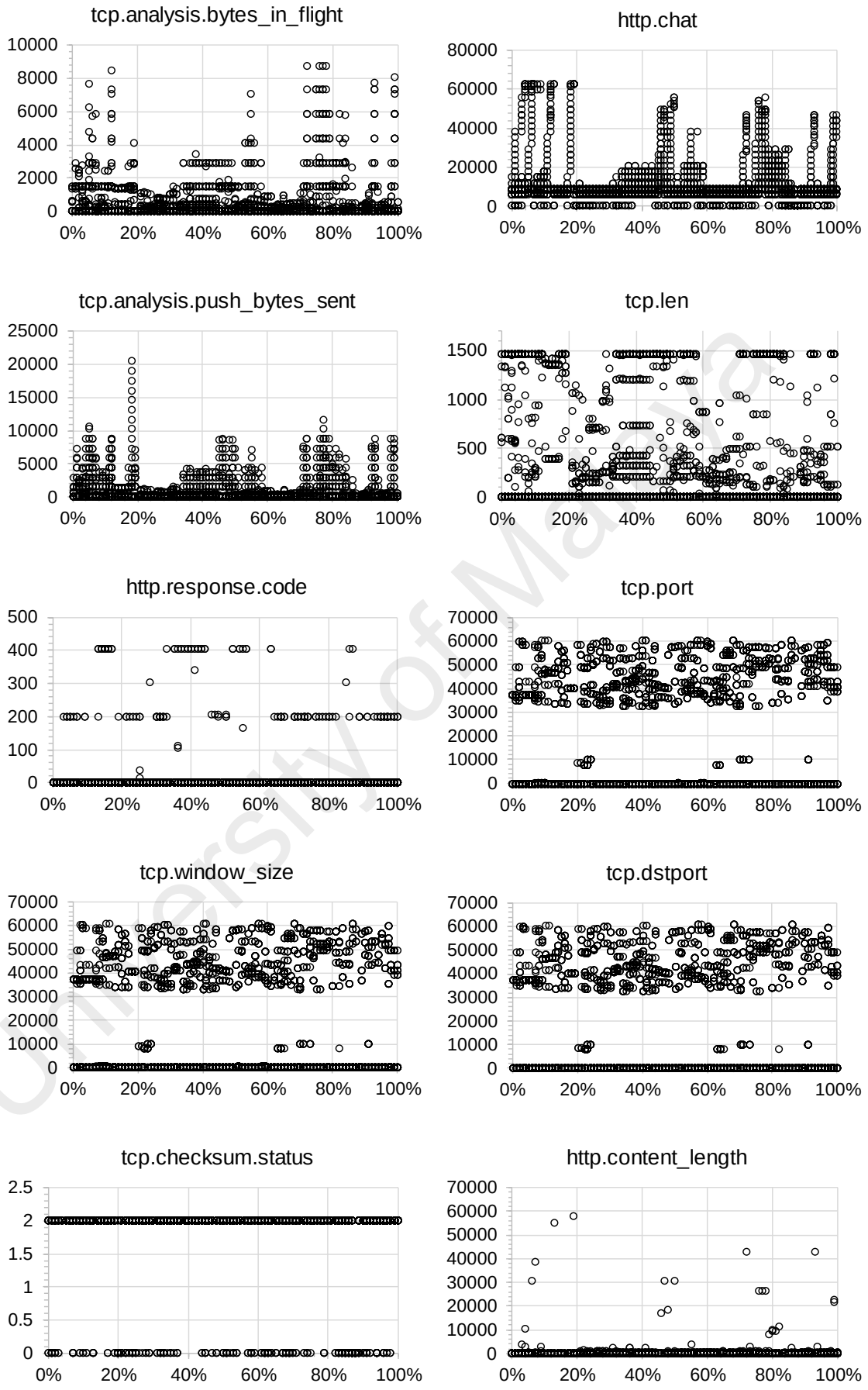


Figure 5.8. Data Distribution of Top 10 Network-related Features

As can be seen from Figure 5.8, the data of http.response.code feature are not distributed. In fact, the data focus on 0, 200, and 400. Although the algorithms select this feature as one of the best, it has the potential of confusing machine learning algorithms. The same situation is true for tcp.checksum.status, where data are concentrated around 0 and 2. This analysis is useful when analysing the effect of each feature in Section 5.3.4.2(c).

5.3.4 Experiment 3: Evaluating Deep Learning Classifiers

The previous experiment selected the top 10 features in network traffic on Android applications. The purpose of this experiment is to evaluate popular deep learning algorithms in mobile malware detection. The deep learning scheme has recently received particularly much attention. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection, drug discovery, genomics, and many other domains. Deep learning discovers intricate structures in large data sets by using the backpropagation algorithm to indicate how a machine should change the internal parameters used to compute an outcome. However, it is essential to investigate the performance of deep learning in mobile malware detection. To the best of our knowledge, this is a gap in the current literature.

5.3.4.1 Deep Learning Algorithms

In this study, we evaluate the performance of deep neural networks (DNN) and long short-term memory (LSTM). The LSTM is a type of recurrent neural network (RNN) that is discussed in the following sections.

The concept of neural networks and deep neural networks has been around since the 1980s. DNNs have recently become popular for two reasons, i.e. today's growing computing power and the dramatic increase in the amount of data and appearance of big data (Hashem et al., 2015). These reasons are compelling enough to revisit the concept of DNNs (LeCun et al., 2015).

A DNN consists of three basic layers, an input layer, a hidden layer, and an output layer. The difference between neural networks and DNNs is the number of hidden layers; two hidden layers and more are considered a deep network. A basic neural network consists of a fundamental unit called a neuron. Every neuron accepts input values and is given a weight. The neuron computes some functions on the weighted inputs to produce output.

The output of the neuron is transmitted as inputs to the next neuron. Connection of neurons form a network that is called neural network, as shown in Figure 5.9.

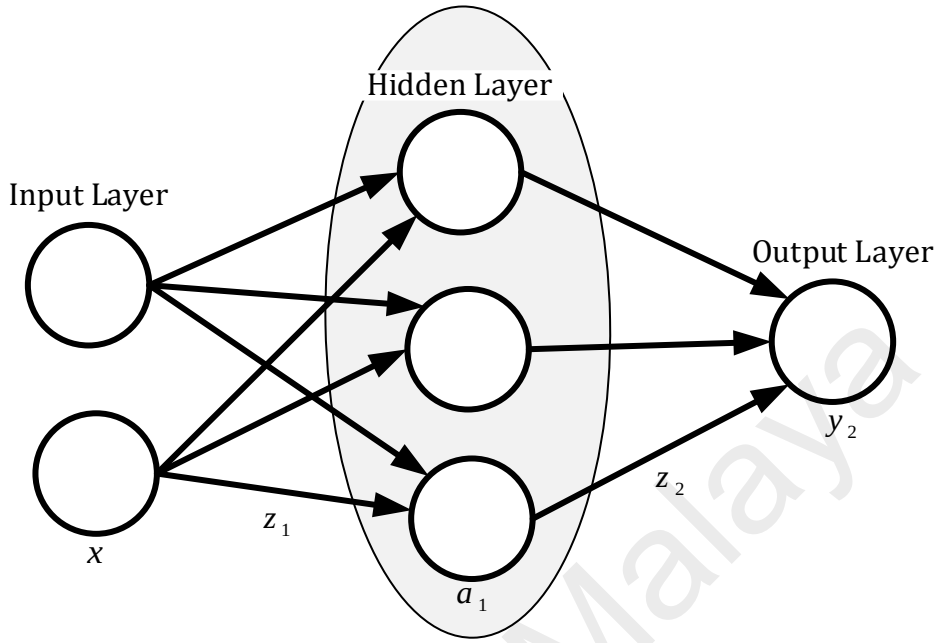


Figure 5.9. Representation of a Neural Network

The first step in a neural network is the forward propagation in which the inputs are propagated across the layers. In addition, the network predicts the output based on inputs. Based on the explanations, the following functions are defined in the forward propagation.

$$z_1 = xW_1 + b_1$$

$$a_1 = \tanh(z_1)$$

$$z_2 = a_1W_2 + b_2$$

$$y_2 = softmax(z_2)$$

The equations z_1 and z_2 are functions that take x as input and use W and b as weight and bias respectively. The $tanh$ is an activation function that takes z_1 as input and passes the result to the next layer. In the output layer, the $softmax$ function is used to calculate y_2 that is the prediction of the neural network.

The next phase is backpropagation, where the actual learning happens. It involves two steps: calculating the loss, and performing optimization. The loss is calculated by comparing the predicted output (y_2) with the actual value of data. Then, the purpose of the optimization function is to minimise the loss function by adjusting W and b (LeCun et al., 2015). As mentioned earlier, a DNN has the same structure as a neural network, but has two or more hidden layers.

A RNN is a new type of neural network that considers sequential information. In a traditional neural network, it is assumed that all inputs and outputs are independent of each other. Nevertheless, for many real-world problems that is not the case. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being dependent on the previous computations. Another way to think about RNNs is that they have a memory, which captures information about what has been calculated so far. They are capable of looking back only a few steps. Figure 5.10 shows the structure of a RNN graph.

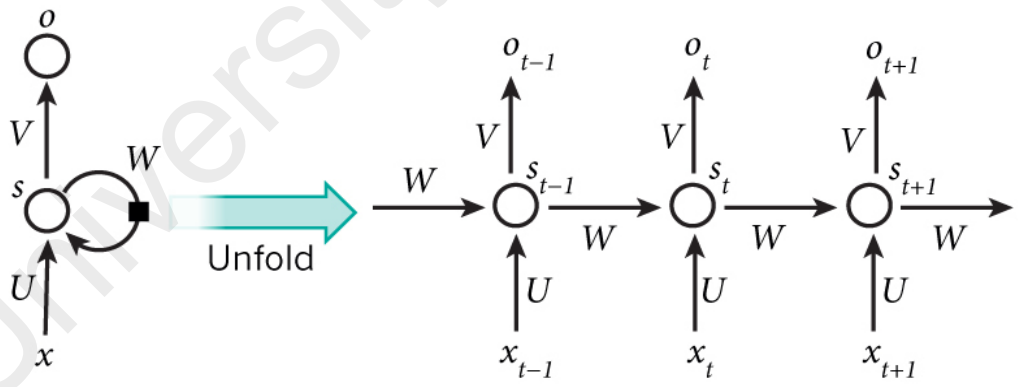


Figure 5.10. A Recurrent Neural Network

The x, s, and o represent input, hidden, and output layers respectively. The U, V, and W are parameters or the weights that need to be adjusted during training. The difference between the traditional neural network and the RNN is an additional input of s_{t-1} that is fed into the hidden layer s_t. Basically, the RNN considers previous steps in its hidden

layer computation, thus it considers sequential information rather than only one piece of data at a time.

A popular variant of RNN is called long short-term memory (LSTM). It was first introduced in 1997 by Sepp Hochrieiter and Jürgen Schmidhuber. LSTMs are capable of bridging time intervals in excess of 1000 time steps[15] even in case of noisy, incompressible input sequences, without loss of short time lag capabilities (Hochreiter et al., 2001). The architecture enforces a constant error flow through the internal states of a special unit known as the memory cell.

There are three gates to the cell: the forget gate, input gate, and output gate. These gates are sigmoid functions that determine how much information to pass or block from the cell. Sigmoid functions take in values and output them in the range of [0,1]. In terms of acting as a gate, a value of 0 means letting nothing through, and a value of 1 means letting everything through. These gates have their own weights that are adjusted via gradient descent in the training phase.

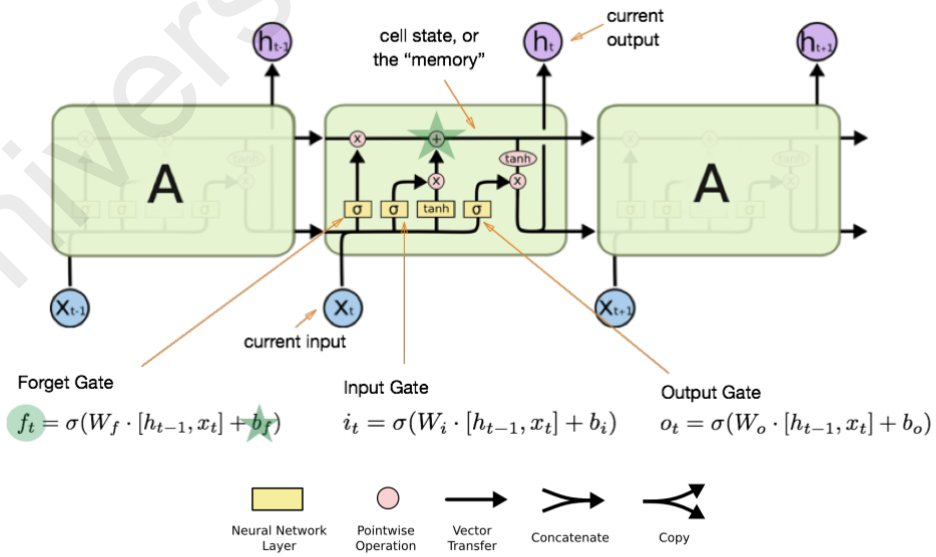


Figure 5.11. The Hidden Layer of LSTM (Mikami, 2016)

[15] Time step is the number of previous results that the algorithm can remember.

In the equations listed under the forget gate, input gate, and output gate in the diagram, h_{t-1} is the previous hidden state, x_t is the current input, W is the weight matrix, and b is the bias. The first step is the forget gate, in which the sigmoid function outputs a value ranging from 0 to 1 to determine how much information of the previous hidden state and current input it should retain. Forget gates are necessary for the performance of LSTM, because the network does not necessarily need to remember everything that has happened in the past.

The next step involves two parts. First, the input gate determines what new information to store in the memory cell. Next, a tanh layer creates a vector of new candidate values to be added to the state.

To determine what to output from the memory cell, we again apply the sigmoid function to the previous hidden state and current input, then multiply that with tanh applied to the new memory cell (this will establish the values between -1 and 1).

The extra complications with the gates may make it difficult to see exactly why the LSTM is better than RNN. The LSTM has an actual memory built into the architecture, which is lacking in RNN. We update the cell memory by adding new information, highlighted with a green star in diagram 5.11, which makes the LSTM maintain a constant error when it must be backpropagated at depth. Instead of determining the subsequent cell state by multiplying its current state with the new input, the addition prevents the gradient from exploding or vanishing. However, we still have to multiply the forget gate to the memory cell.

5.3.4.2 Results

This section presents the results of an experiment on deep learning classifiers. The experiment was performed on a desktop PC, with Intel Core i5-2400 at 3.10 GHz and 20

GB of RAM, running Microsoft Windows 10 with the latest updates. Python 2.7 and TensorFlow were also installed on the machine. Depending on the extent of our experiments, we either performed them on the PC or on the Google Cloud ML platform. This section is divided into three parts: preliminary results, the effect of hyperparameter optimization, and the effect of features.

Since this experiment is continuation of pervious one, the same dataset is used, which is a list of selected features. For this experiment, we divided the data into training and testing set with ratio of 70% to 30% respectively. The evaluation results are presented by using two common metrics. Accuracy is a ratio of all correct predictions made by an algorithm. Logarithmic loss (or loss) is a performance metric for evaluating the predictions of probabilities of membership to a given class. The scalar probability between 0 and 1 can be seen as a measure of confidence for a prediction by an algorithm. Predictions that are correct or incorrect are rewarded or punished proportionally to the confidence of the prediction.

(a) ***Preliminary Results***

This section reports the results of the experiments using DNN and LSTM algorithms. The experiments used the top 10 features and default settings of the algorithms. The purpose of these experiments was to observe the performance of DNN and LSTM as a baseline, without tweaking any algorithms. Table 5.14 shows the results of the experiments.

Table 5.14. Preliminary Results of DNN and LSTM

	Accuracy	Loss
Deep Neural Network (DNN)	80.93%	0.18
Long Short-term Memory (LSTM)	81.96%	0.13

The above results show that LSTM performed better than DNN, with an accuracy of 81.96%. It means that out of all predictions made by the algorithm, 81.96% was correct. The results also show that the LSTM performed better than DNN, as expected. It is due to its neurons' architecture that can memorize data. It makes predictions based on current and previous data.

(b) ***The Effect of Hyperparameter Optimization***

In the previous section we mentioned that internal parameters are adjusted during training. There are two types of parameters, those that are adjusted during training, and those that can be tuned by us. In this section the effects of the latter type of parameters (known as hyperparameters) are thoroughly investigated.

The process of finding a set of hyperparameter values that gives us the best model is called hyperparameter optimization. Epoch (or global step as referred to in Figure 5.11) is the number of iterations that the training dataset shows to an algorithm during training. Batch size is another parameter that is optimized. It is a number of data inputs to show to the algorithm during the training process, based on which weight is updated internally. Some algorithms such as LSTM are sensitive to the batch size (Bergstra & Bengio, 2012). An optimizer (discussed in Section 5.3.4.1) is another parameter that can be tuned for better output.

Table 5.15 shows the results of parameter optimization for epoch and batch size for DNN and LSTM algorithms. Epochs of 10, 50, 100, 500, and 1000 were evaluated. Additionally, batch sizes of 10, 20, 40, 60, 80, and 100 were tested. The accuracy for the combination of epoch and batch size is available in Table 5.15. Based on the results, an epoch of 50 and batch size of 20 achieved an accuracy of 81.2857%, which is higher than the other configurations for DNN algorithms. Similarly, an epoch of 50 and batch size of

Table 5.15. Results of Hyperparameter Optimization for Epoch and Batch Size

DNN			LSTM		
Accuracy (%)	Number of Epochs	Batch Size	Accuracy (%)	Number of Epochs	Batch Size
72.2857	10	10	80.4286	10	10
43.8333	50	10	82.5952	50	10
45.8571	100	10	80.9762	100	10
72.8571	500	10	81.0714	500	10
55.5	1000	10	80.9286	1000	10
80.5476	10	20	80.6667	10	20
81.2857	50	20	80.9286	50	20
27.8333	100	20	81.381	100	20
72.0238	500	20	81.3571	500	20
71.0952	1000	20	81.7143	1000	20
53.5714	10	40	80.5476	10	40
55.0476	50	40	80.9762	50	40
53.6905	100	40	80.9286	100	40
54.7381	500	40	81.4524	500	40
63.9048	1000	40	81.6429	1000	40
77.6667	10	60	72.0476	10	60
71.3333	50	60	80.8095	50	60
53.7381	100	60	82.0952	100	60
72.7857	500	60	81.5952	500	60
53.0952	1000	60	80.8333	1000	60
53.3333	10	80	57.619	10	80
62.4762	50	80	80.9048	50	80
54.5238	100	80	81.1429	100	80
62.8333	500	80	81.4762	500	80
35.9762	1000	80	81.5238	1000	80
64.2857	10	100	61.0238	10	100
45.5714	50	100	81.4524	50	100
54.2619	100	100	80.7857	100	100
54.6667	500	100	80.9762	500	100
44.9048	1000	100	81	1000	100

10 achieved 82.5952%; that is the best result compared to the other configurations for LSTM algorithms.

Moreover, we evaluate different optimizers for DNN and LSTM algorithms to identify the best one for each algorithm. Specifically, SGD, RMSprop, adagrad, adadelts, adam, adamax, and nadam optimizers are tested. The results are shown in Table 5.16. For DNN algorithms, the adam optimizer achieved the best result with an accuracy of 80.9286%. Similarly, the nadam optimizer achieved 82.6429% accuracy for the LSTM algorithm. It is deduced that running DNN and LSTM algorithms on our data using adam and nadam, respectively, results in the best performance.

Table 5.16. Results of Hyperparameter Optimization for Optimizers

DNN		LSTM	
Accuracy (%)	Optimizer	Accuracy (%)	Optimizer
27.8333	SGD	80.4048	SGD
61.9048	RMSprop	80.9286	RMSprop
51.3095	Adagrad	81.4762	Adagrad
80.9048	Adadelta	81.5238	Adadelta
80.9286	Adam	81.0238	Adam
65.5952	Adamax	81	Adamax
80.0238	Nadam	82.6429	Nadam

(c) ***Effect of Features***

In this study, 10 network-related features were selected. It is often asked how to know if 10 is the optimum number of features that yield the best results. It is also asked if each feature has positive effect of the results. In this section, we explore the effect of the selected features on the final results. As the previous results of the LSTM were higher than for DNN, the former algorithm is used in this section. Furthermore, this experiment

utilizes the optimized values of hyperparameters obtained from the previous section. Table 5.17 shows the results of this experiment.

Table 5.17. Results of Effects of Number of Features Experiment

Number of Features	Accuracy (%)	Loss
10	81.96	0.13
9	57.91	28.96
8	83.81	0.1308
7	83.93	0.1386
6	82.37	0.1405
5	82.55	0.1409
4	80.05	0.1645
3	80.98	0.1516
2	93.06	0.0694

It is worth noting that order of the features corresponds to Table 5.13. Based on Table 5.17, choosing ten features results in 81.96% of accuracy. This accuracy drops to 57.91% using nine features, and increases to 83.81% for eight features. It is believed that the ninth feature (tcp.checksum.status) introduces noise to data, removing of which results in increase in results (83.81%). It is also believed that having a tenth feature (http.content_length) in the dataset masks the noisiness of the ninth feature. When using seven to three features, the results range between 80.05% and 83.93%. Experiments using two features (http.chat, tcp.analysis.bytes_in_flight) show that they achieve 93.06% accuracy, which is the best result among the experiments.

Although two features result in the best achieved accuracy, they are most effective among 10 selected features. tcp.analysis.bytes_in_flight shows amount of data that has been sent, which indicate the data sent from a device to a server. A malware leaks user's data to

attacker, and this feature represents the leaked data. However, normal applications also send data to a server. Another feature (http.chat) is combined with tcp.analysis.bytes_in_flight to help differentiate malware from normal. This feature is a label that shows availability of back and forth transmission between the device and a server. The combination of these two features along with capability of machine learning to learn patterns in data, make the two features more effective than others.

As mentioned earlier, accuracy and loss are two measures that are calculated for the LSTM algorithm. Figure 5.12 shows the accuracy of the LSTM during the experiment. The experiment has 50 epochs that are shown as x-axis, and the y-axis represents the corresponding accuracy value for each epoch.

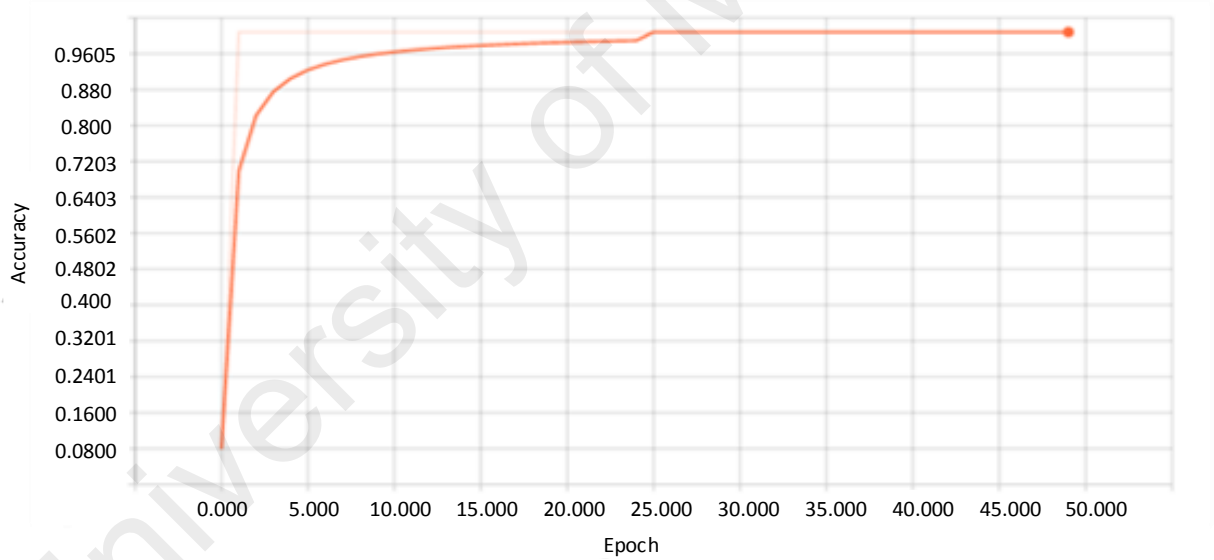


Figure 5.12. The Accuracy Result of LSTM

It is visible that the accuracy is increasing throughout the experiment, showing that the algorithm is learning to produce results that are more accurate. Figure 5.13 shows the loss during the experiment.

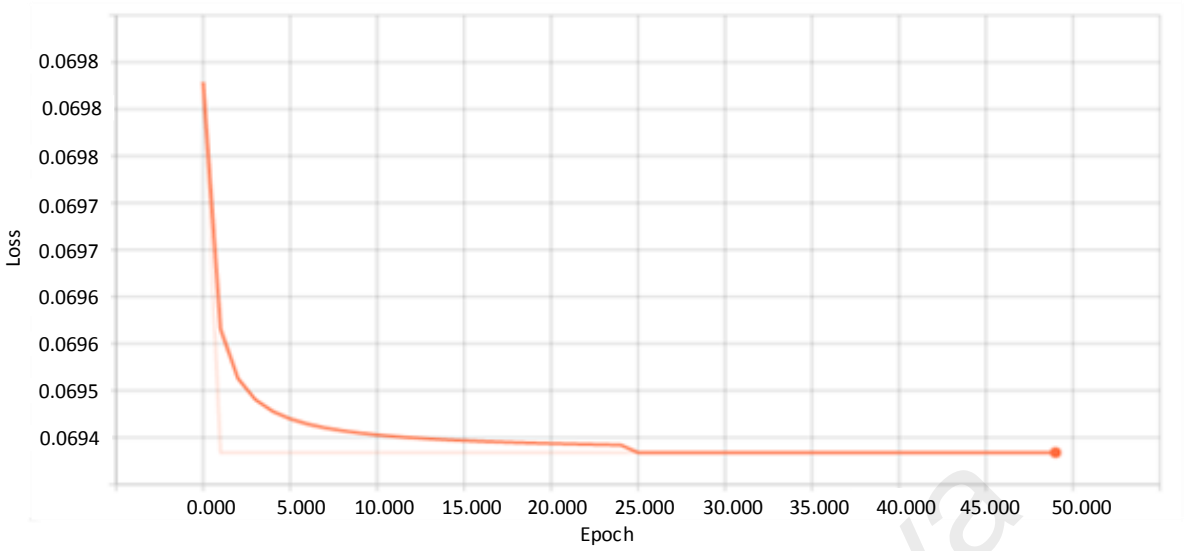


Figure 5.13. The "Loss" Result of LSTM

Figure 5.13 shows that as the results increase, the loss is decreasing. This means that the algorithm is adjusting its internal parameters to produce results that are less incorrect. The x-axis represents the number of epochs, which is 50 in this experiment. Similarly, the y-axis shows the loss in value for each epoch. Figure 5.14 depicts values of weight in each layer of the LSTM algorithm.

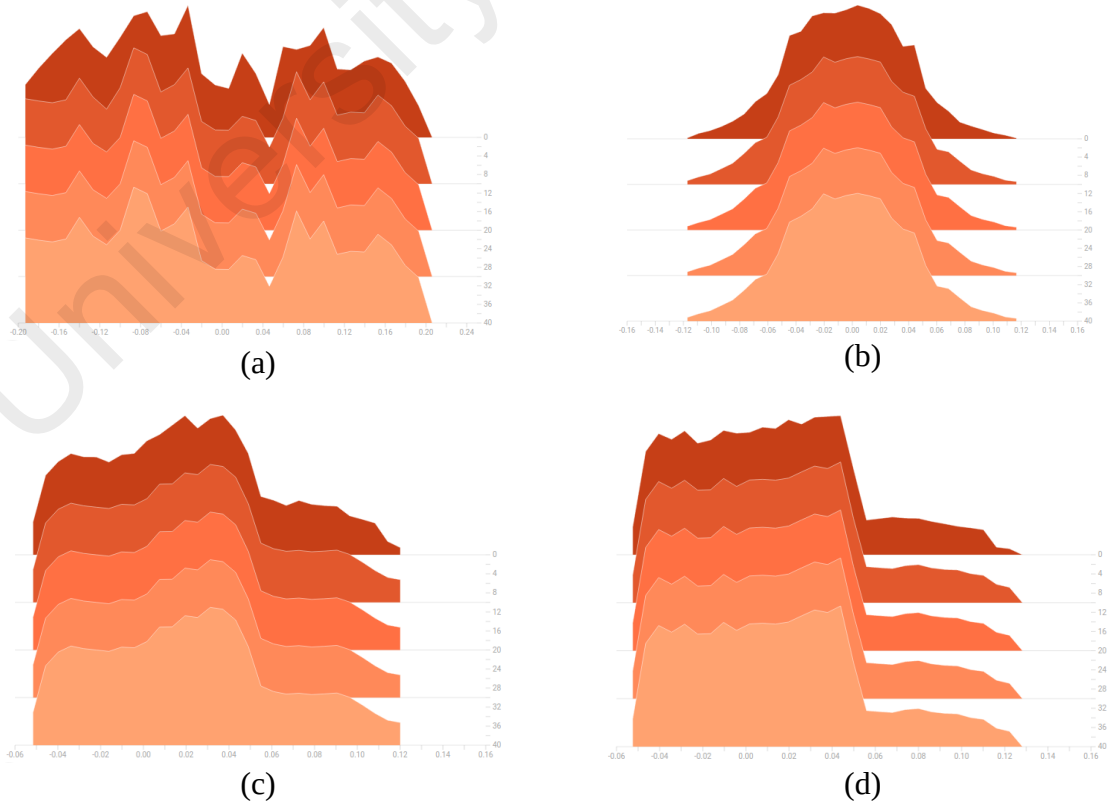


Figure 5.14. The Values of Weight in Four Layers During LSTM Experiment

The LSTM algorithm adjusts its weight parameters as it learns the data during the training phase. Figure 5.14 shows variations of the weight parameter in four layers of the LSTM. The x-axis is that value of the weight in each layer, and the y-axis is the epoch in the experiment. The represented graph is a histogram chart, showing the distributions of the values in each layer. Generally, the darker, more central, bands have more values in that range while the lighter, wider bands have fewer values. It is beneficial to think of each band as a bin in a regular histogram chart, the darker bands are the taller bins, etc. The difference is that the TensorFlow draw the histograms over time.

It is beneficial to compare the results to other related works. Zhu et al. performed taint analysis on Android APK files, which analyses leak data and finds the methods the attackers use to access them. This type of analysis can easily be bypassed by attackers using obfuscation method. The authors chose deep belief networks to conduct their experiment. Their dataset contains 3,000 benign applications and 8,000 malicious applications. The final results show 95.05% of detection score (Zhu et al., 2017).

Yu et al. used permissions and system calls to build a neural network. They constructed feed forward network and recurrent network as chosen types for neural network (W. Yu et al., 2014). They evaluated their system using 96 benign applications and 92 malware applications. Although the achieved results seem to have achieved high score of 95%, their evaluation was performed on very small dataset as opposed to 100,000 samples in this work. As size of dataset increases, the final system can be generalized to wider types of application and detect broader types of malware.

Martinelli et al. chose convolutional neural network to conduct their experiment. This type of neural network is best suited for image and video classification due to the layers' structure, however, the authors decided to use it in their experiment. They collected features pertaining to UI interactions and system events, such as touches, gestures,

reception of SMS, incoming call, etc. They used Monkey program to mimic user's interaction with the device, which is not as accurate as the real user. The experiment results show precision of 75% and 80% for UI interaction and system events respectively (Martinelli et al., 2017).

Xiao et al. used system calls for analysing Android applications. They claim that considering there is some semantic information in system call sequences as the natural language, they treat one system call sequence as a sentence in the language and construct a classifier based on the Long Short-Term Memory (LSTM) language model. However, not all Android system calls can be treated as a sentence in natural language, such as closehandle. The experiments show that this approach can achieve recall of 96.6% (Xiao et al., 2017).

Overall, the mentioned works merely used deep neural networks and output the results. However, the neural network algorithms consist of many parameters that can be tuned, which results in better outcome. In this work, we dedicated a sub-section for hyperparameters optimization to find out the optimum value for each parameter. Additionally, our data sample of 100,000 Android applications is much larger than other works that translates to having more generalized and accurate results.

5.3.5 Conclusion

Experiments 2 and 3 were related to dynamic analysis. In Experiment 2, we extracted 30 features from network traffic of Android applications. Then, using four algorithms, the top 10 features were selected. The selected features were used in experiment 3 to evaluate the performance of deep learning algorithms (DNN and LSTM). At the end, the results show an accuracy of 93.06% and a loss of 0.0694, which reveals that network traffic is an effective feature in malware detection.

At this stage, we have proposed and evaluated the framework (DroidProtect) that fulfils the objectives b and c (Section 1.4).

5.4 Experiment 4: Evaluation of Energy Consumption

As mentioned in Section 1.4, the main objective of this study (objective d) is to propose a malware analysis and detection system that minimises energy consumption. The purpose of this experiment is to serve that objective by measuring the energy usage of the developed prototype and compare it with the energy consumption of other available similar products.

5.4.1 Energy Consumption Fundamentals

Section 3.4 mentioned the definition of energy profilers, and discussed the evolution of such products. In this section, we will further explore the basics of measuring energy consumption.

The first topic is the difference between energy and power. The former denotes the capacity of a system to perform work. The latter is the rate of energy consumption that is how much work the system is doing. This concept is clearer with an analogy involving water. Power is gallons per minute, which goes to zero if the usage stops. Energy, however, is the total gallons used, and does not go to zero if the usage stops. Thus, we are looking to measure the total energy consumed by our application.

The unit for energy is joule (J) defined as the amount of energy required to continuously produce one watt for one second (W_s). The unit for power is watt (W) defined as one joule per second $\left(\frac{J}{S}\right)$. Therefore, the relationship between energy and power is defined as in equation $E_{(j)} = P_{(w)} \times t_{(s)}$, where the unit for power is watt and the unit for time is seconds.

The second topic relates to the methodology of calculating energy consumption. As mentioned in Section 3.4, this study employs AppScope and PowerTutor for calculating consumption. Simply put, these frameworks calculate energy consumption based on a

power model. That model is generated either off-device for various devices, by the developer, or is generated on-device by observing the device's consumption pattern. The research community has adopted both frameworks (Barbera et al., 2013; Chang et al., 2011; X. Chen et al., 2013; Saipullah et al., 2012).

PowerTutor is developed based on PowerBooter, which is an automated power model creation technique that uses on-device voltage sensors and battery discharge curves based on the Rint model to estimate power consumption (L. Zhang et al., 2010). The power consumption is then correlated with individual components using regression. The system does not require external measurement equipment; however, a smartphone-specific discharge curve is needed. The new idea of PowerBooter was to use battery-state-based power model generation. This involves keeping smartphone components in specific power states so that their power consumption can be determined through the change in the battery's state of discharge (SOD) using a voltage sensor. This change can be used to estimate the average power draw. When the component-specific average power draw is known, it is possible to derive the power model using regression.

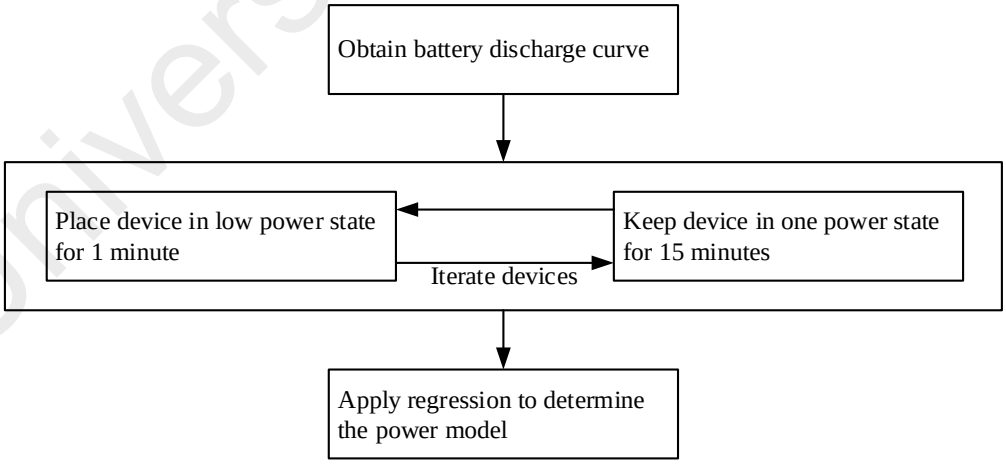


Figure 5.15. Overview of the PowerBooter Model

Figure 5.15 illustrates the key phases of PowerBooter. In the first step, the battery discharge curve for the phone is constructed. The discharge curve varies from phone to phone due to differences in battery type, age, temperature, and operating parameters. The

discharge curve can be obtained online and on-device by observing the constant discharge behaviour from a fully charged state. In the second step, the power consumption is determined for each component state. The state of a component is varied while keeping the rest of the system in a static configuration. The battery voltage is recorded at the beginning and end of a discharge interval. The voltage is measured for 1 minute and the battery is discharged for 15 minutes between the component voltage measurements. In the third step, regression is used to create the power model. The battery voltage differences for each discharge interval are used to determine the average power draw of the 15-minute intervals. Regression is then used to create the power model based on the component average power draw estimates.

AppScope uses the DevScope power model to estimate energy consumption. DevScope (Jung et al., 2012) is an example of an energy profiler that uses a smart battery interface to generate an on-device dynamic linear regression-based power model. The DevScope authors observed that the smart battery interface has a low update rate. They proposed a synchronisation technique between the update rate and component-specific control. The profiler works by probing the OS to obtain information about the components and the configuration, such as the CPU details. The profiler also examines the smart battery interface and determines the update rate of the battery interface.

Similar to PowerBooter and Sesame, the profiler then creates a component control scenario for power analysis for the specific smartphone. The control scenario is then run and DevScope first classifies the data to the terms of the power model and then analyses the classified data to update the power coefficients of the regression model. For example, each CPU frequency is tested with zero and maximum use, to derive the information needed for the CPU model. To alleviate the slow update rate of the smart battery interface, DevScope synchronises the smart battery update events with the component tests. This

contrasts with Sesame's solution of averaging battery readings for higher accuracy at a slower rate. DevScope also tries to recognize power-state transitions; however, this requires knowledge of the power-state durations and the battery update interval. Automatic detection of power-state transitions is difficult, because the state transitions are governed by the workload and the operating conditions. During component testing, DevScope repeatedly uses different workload sizes to determine the threshold size that results in a power-state change. This technique is applied for cellular and Wi-Fi connections to determine the wireless network parameters and power state details.

5.4.2 Results and Discussion

This work aims at proposing mobile malware analysis and detection methods that consume less energy compared to similar products. Thus, this experiment serves that purpose. The results are presented as follows. First, the energy consumption of a number of normal applications is calculated in order to establish a baseline of how much energy is consumed by applications that are used every day.

Second, we show how much energy our framework (DroidProtect) consumes when it is implemented using a local approach (refer to Section 2.4.4.1), which is running the analysis and detection on the device. Third, the amount of consumed energy is calculated when implementing the DroidProtect using the offloading method (refer to Section 2.4.4.2). The objective is to show that the offloading method consumes less energy than the local method. We have mentioned benefits of offloading in Section 4.3.1. It is beneficial to prove that practically. Fourth, the energy consumption of similar products is calculated and the results are compared to the previous experiments, to see whether this study achieved its objectives. It is worth noting that the screen brightness of the device was set to 50% during the following experiments.

Table 5.18 shows the energy consumption of normal applications during 10 minutes of usage. Four popular applications were measured that were selected from four categories of popular activities, i.e. multimedia, games, social networking and messaging. It is worth mentioning that the usage of these applications was medium, such as watching video with 480p resolution on YouTube, checking news feed on Facebook, etc.

Table 5.18. Energy Consumption (in Joules) of Three Popular Applications During 10 Minutes Usage

Application	CPU	Communications	Display	Total
YouTube	30.11	12.59	508.90	551.59
MX Moto	129.24	5.75	509.54	644.52
Facebook	137.76	27.42	471.42	637.27
WhatsApp	39.8	24.1	458.7	522.6

The calculations were performed in the form of several time series, each one associated with a component of the device, namely CPU, Wi-Fi or cellular communications, and display. Table 5.19 shows the results of the energy consumption test of the DroidProtect when analysing one application. The test was performed on the application with the size of 1.3 MB. The analysis took around two minutes for each type of analysis. The estimated consumption during 10 minutes is shown in parenthesis inside Table 5.19.

Table 5.19. The Results of Energy Consumption Test for DroidProtect (Joules)

	Local				Offloading			
	CPU	C	D	Total	CPU	C	D	Total
Static Analysis	18.7	-	49.4	68.1 (340.5)	14.6	1.0	28.1	43.7 (218.5)
Dynamic Analysis	5.3	-	27.1	32.4 (162)	1.6	1.1	21.6	24.3 (121.5)

C = communications, D = display

As can be seen from Table 5.19, the experiment was performed using local and offloading approaches. Thus, the difference between the two approaches is visible. In the static

analysis, the local approach consumed 68.1 Joules while the offloading approach consumed 43.7 Joules. As the detection process (the use of machine learning) is performed on the device in the local approach, it is expected to consume more energy than the offloading approach.

The dynamic analysis consumed 32.4 Joules using the local approach while it used 24.3 Joules using the offloading approach. The same rationale is also applicable here, namely that as the detection is performed on the device, the energy consumption is higher.

Another outcome of this experiment derives from comparing the static and dynamic analysis. Choosing either local or offloading methods reveals that the static analysis used more energy than the dynamic analysis. It is believed that since the static analysis involves decompiling the DEX file to Java, it consumes more energy than observing and collecting network traffic (dynamic analysis).

It is also inferred that the offloading method consumes energy to send the collected features to servers, which does not occur in the local approach. Despite that there is an additional component drawing energy in the offloading method, the overall process consumes less energy compared to the local approach.

Next, we mention energy consumption of similar security applications. As mentioned in Section 1.3, Polakis et al. analysed a number of security applications with regard to their energy consumption. Their results are presented in Figure 5.16 for AVG, Dr. Web, Sophos, Avast, Norton, and NQ. It shows that NQ uses the least and Dr. Web uses the most energy among the applications. The results are presented based on millijoules per second, while we calculated Joules per ten minutes in the previous experiments. It is estimated that NQ uses 3,600 Joules and Dr. Web uses 28,800 Joules in ten minutes (refer to Section 1.3).

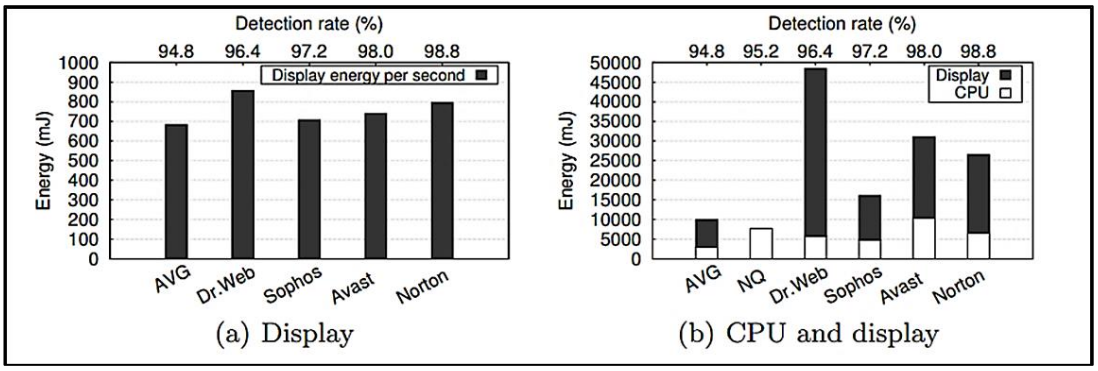


Figure 5.16. The Results of Energy Consumption Test for Security Applications (Polakis et al., 2015)

The comparison between the DroidProtect and the similar security applications shows that the DroidProtect consumes less energy. Considering 10 minutes of usage, while NQ uses 3,600 Joules energy, our proposed framework uses 121.5 Joules. The comparison was made between the least consumed energy in our work and similar products. Moreover, the comparison between normal applications and this study shows that the DroidProtect uses less energy.

5.5 Summary

The purpose of this chapter was to evaluate the proposed framework. It started by evaluating a static analysis of Android Intent using real-world applications. This was followed by a dynamic analysis of network traffic, including choosing the best network-related features and evaluating deep learning algorithms. The next experiment was related to energy consumption of the proposed framework, which fulfils the objectives of this study.

The next chapter implements the proposed framework to show how it works as a standalone application. It also discusses an activity diagram, and includes screenshots from various functions of the application.

CHAPTER 6: A PROTOTYPE IMPLEMENTATION OF MOBILE MALWARE ANALYSIS AND DETECTION SYSTEM

Following the system evaluation described in the previous chapter, this chapter presents the design and implementation process of a prototype of DroidProtect. This stage is crucial as the system is put into practice, and its various parts constitute a complete prototype. The implementation is divided into two sections: mobile and server. The mobile device section is further divided into static and dynamic sections. Each section is discussed in the following parts.

Java programming language was chosen for the implementation. Android Studio and Eclipse were selected for developing the mobile and server section respectively. They are well-known integrated development environments (IDE), offering various tools for programming and debugging. Google App Engine was selected for developing the server side of this work for the following reasons:

1) It performs better in terms of maintainability and scalability for mobile applications compared to Amazon EC2 and Microsoft Azure. Google App Engine automatically creates additional instances of the application when required due to the increase of usage (Jonge, 2011).

2) Google App Engine is characterized as a software developers' platform, whereas Amazon EC2 is characterized as a system administrators' platform. Whereas software developers upload their code to Google App Engine and test their application, users of Amazon EC2 need to configure several settings to run their applications, making the process more complex (Jonge, 2011).

Moreover, Google App Engine uses some security mechanisms. For instance, prior to uploading a file, the application first requests an upload URL. Upon receiving the upload

URL, the file is sent to the servers. Afterwards, the same link does not work to upload a new file, and a new request must be established. This mechanism prevents the upload of malicious files to the server. Another mechanism specifies the content type when uploading a file. Applications can set their own content type. Retrieving the file requires knowing the correct content type. Thus, the server is protected from cross-site scripting (XSS) attacks in which attackers can use text/html content type to gain access to files (Grossman, 2007).

6.1 Activity Diagram

Before describing each section in detail, it is beneficial to look at the process flow of this prototype through an activity diagram. An activity diagram describes the sequential or concurrent control flow between activities, and can be used to model the dynamic aspects of a group of objects, or the control flow of an operation. It emphasizes the activities of the object; hence, it is well suited to describe the realization of the operation in the design phase. Moreover, it describes the sequence of activities among the objects involved in the control flow during the implementation of an operation, the relationship between the activity and the object in the message flow, the state change of object in the object flow, and the execution of the activity (Linzhang et al., 2004). Figure 6.1 shows the activity diagram of DroidProtect.

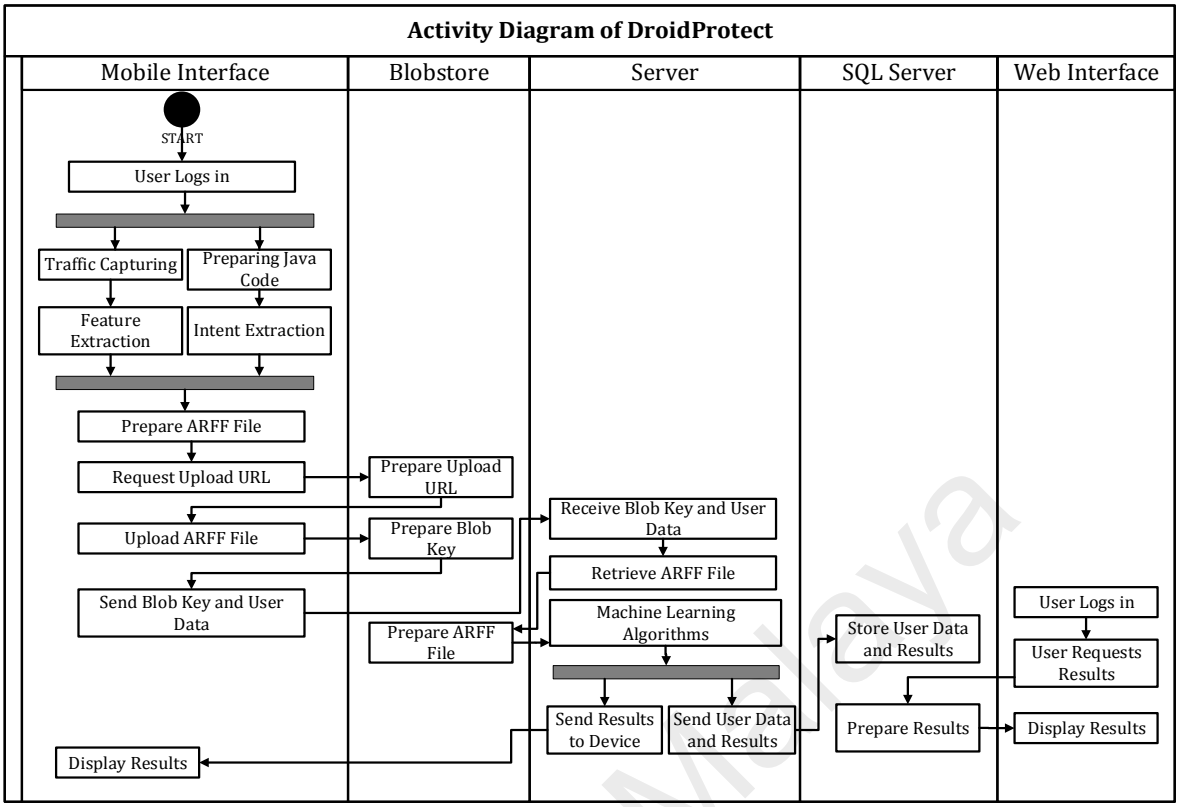


Figure 6.1. Activity Diagram of DroidProtect

The activity diagram includes five components, which are described as follows, based on the process flow of the system.

a) Mobile interface. The process starts when the user logs into the mobile application. The login process is designed to be very simple and requires no registration. The user can log in using Google account credentials. The mobile application collects only basic data such as name and email address. Upon logging in, the user starts the packet capturing and Intent extraction processes. These processes are designed to be lightweight and not to interfere with the user's activity. The user simply starts the processes and resumes working with other applications, as the processes are performed in the background. At the end, an ARFF file is produced that contains extracted features. Then this file is uploaded to the servers.

b) Blobstore. Blobstore is the name of a storage space inside the Google App Engine. It is designed to be fast and reliable. The mobile application requests an upload URL from the Blobstore. After receiving the URL, the ARFF file is uploaded. Then a unique key

(known as blob key) for the uploaded file is sent to the mobile application. Afterwards, the blob key and the user's data are sent to the server for further processing.

c) Server. The server retrieves the ARFF file and sends it to machine learning models. First, the machine learning algorithms are trained, and a model is generated. Models are used to predict the maliciousness of the incoming data. At the end, the results are produced and sent to the user's device and an SQL server.

d) SQL server. The machine learning results are stored in the SQL database along with the user's data. Hence, the results and their history can be retrieved and displayed on a web site.

e) Web interface. In addition to the mobile application, we developed a web interface (also hosted on the App Engine), so that the results are available online. This is more convenient for the user to manage the detection results, and to review the history of the analysis and detection processes.

6.2 Implementation of the Mobile Application

The mobile application consists of two sub-sections, static analysis and dynamic analysis. The objective of this application is to collect static and dynamic data, extract features, and send the features to the server for analysis. Each visible page of an Android application is called an activity. Figure 6.2 shows the first activity of the prototype, which appears after launching the application.

The first activity is designed to be very simple and intuitive. The only button is for signing in with the user's Google account. The user is then asked to grant permission so that the application can access basic data such as name and email address, as shown in Figure 6.3. This allows the application to get access to basic information.

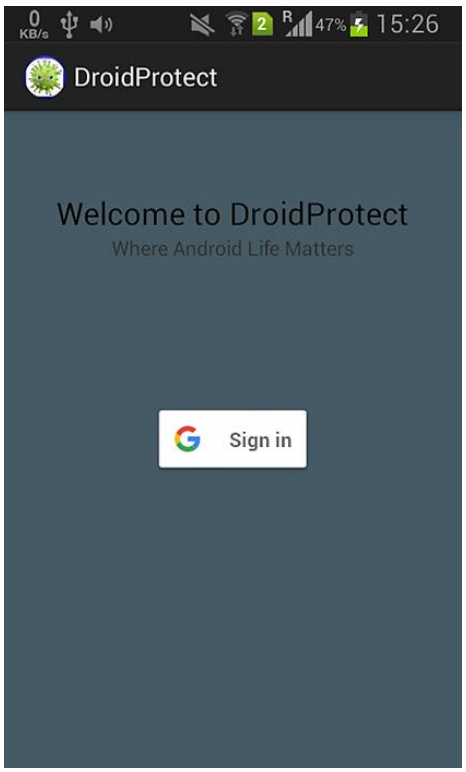


Figure 6.2. The First Activity of Mobile Application

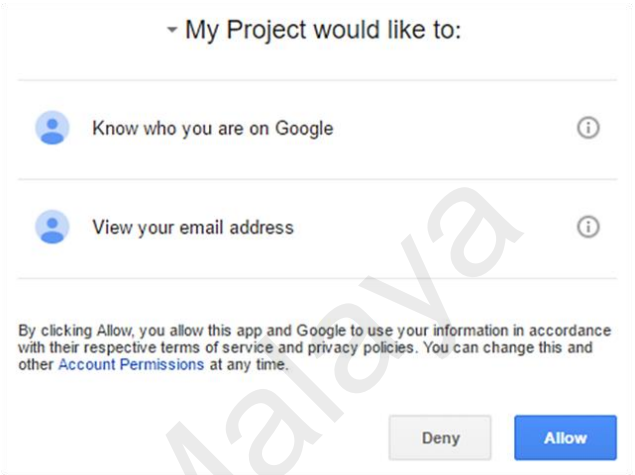
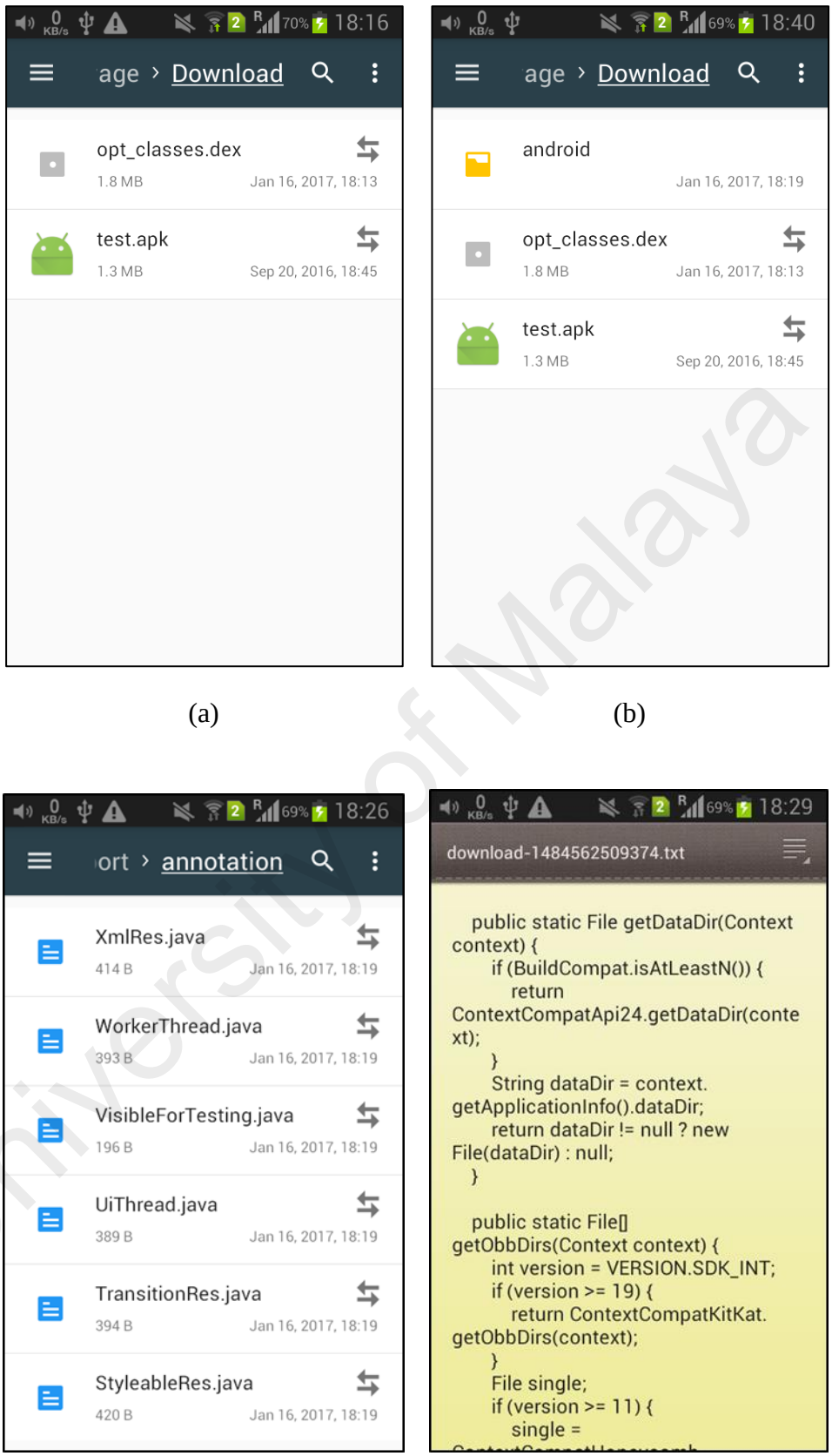


Figure 6.3. Google Asks Permission to Share User's Data

Subsequently we divide the application into two sections, static analysis and dynamic analysis. As mentioned in Section 2.4.1.1(c), Android Intent is our choice for static analysis. Similarly, network traffic was selected for the dynamic analysis.

Static analysis employs Java code to reverse the APK file to DEX file, and then to semi-readable Java files. The former process uses smali APIs and the latter uses Jadx APIs. Figure 6.4(a) shows the results of conversion from APK to DEX. Figure 6.4(b) shows that after decompiling the DEX file, an Android folder is created. The content of this folder is presented in Figure 6.4(c). Figure 6.4(d) shows the content of the produced files.



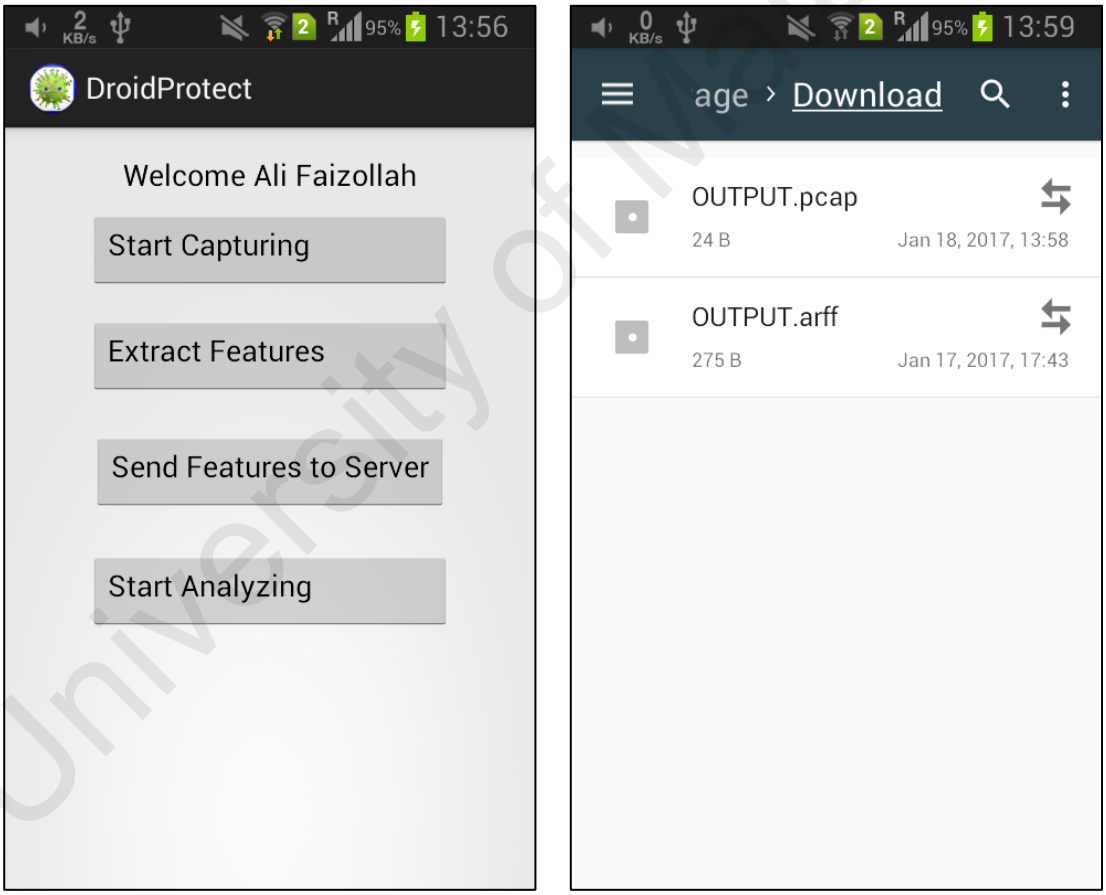
(a) (b)

(c) (d)

Figure 6.4. Screenshots of the Results of Static Analysis

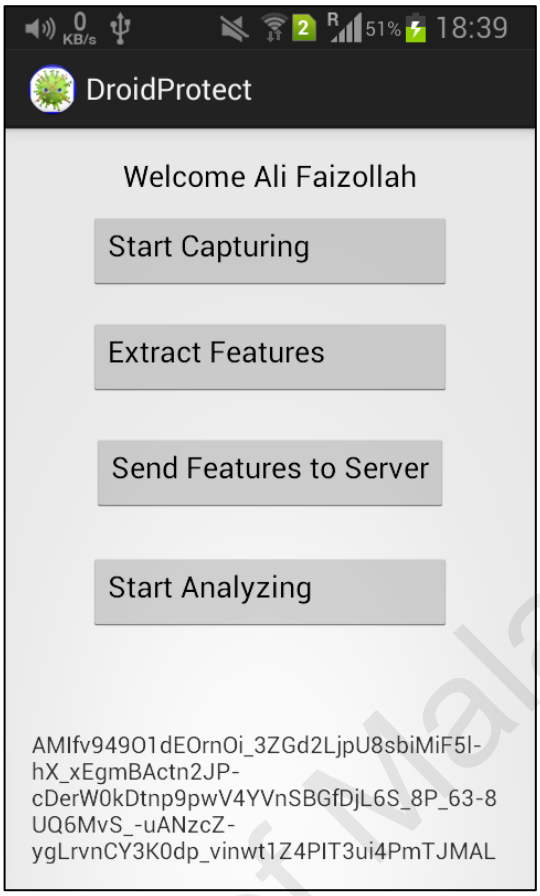
Although the final results are not pure Java codes, it is possible to search for Android Intent in the code. Thus, at the end the Intent feature is extracted by searching the Java code. The mentioned processes are performed as a background service so that they do not interfere with the user's activity.

The dynamic analysis is similar to the static analysis. The network traffic is captured by the TCPDUMP program. The collected traffic is saved as a PCAP file, and Java code is used to extract the features. The extracted features are saved as an ARFF file. The ARFF file is utilized by Weka for further processing. Figure 6.5 shows the above processes.

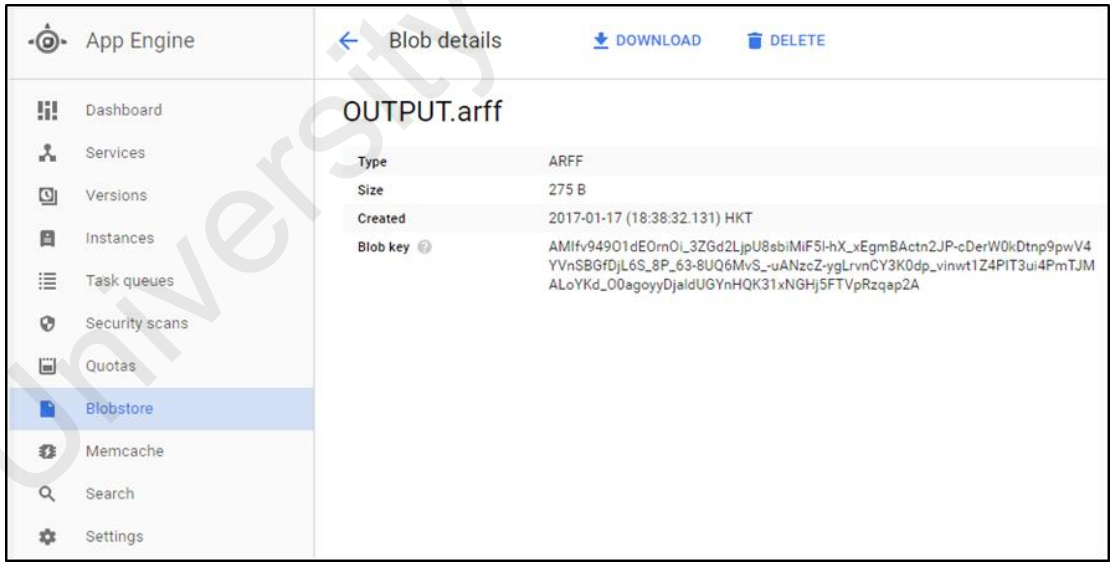


(a) (b)

Figure 6.5. Screenshots of Dynamic Analysis Process of the Mobile Application



(a)



(b)

Figure 6.6. Screenshots of the Upload Process from Mobile to Servers

Figure 6.5(a) shows the steps of the dynamic analysis section of the mobile application. Capturing network traffic and extracting features produces a PCAP and an ARFF file respectively, as shown in Figure 6.5(b). Figure 6.6(a) shows the application after

uploading the ARFF file in which the server returns a unique key representing the file. Afterwards, the key is used to refer to that specific file for analysis. Figure 6.6(b) shows the server side of this process, where the file is stored on the server, with the same key as presented on the mobile application. Figure 6.6(b) also displays the Google App Engine and the Blobstore section where the files are stored.

On the server side, the static and dynamic features are integrated. Then, a prepared machine learning model is used to determine the maliciousness of the data. At the end, the results are sent back to user's device. In addition, the results are stored on Google SQL server, which is part of the Google Cloud Services.

On the SQL server, each user is identified by the email address used when logging in with his Google account. Thus, the email address is the primary key in the SQL server.

Another component of this system is a website, which allows the users to check the results of the detection process. Users also can view their detection history.

6.3 Summary

This chapter demonstrated the implementation of the proposed framework in form of a prototype through screenshots. In addition, the activity diagram was illustrated, and each of its components was discussed.

The purpose of this chapter was to show the proposed framework as a final product. It helps to understand how the DroidProtect and its components work. Due to time constraints, it was not possible to show every detail of the DroidProtect. Achievements, limitations, and suggestions for future works are discussed in the next chapter.

CHAPTER 7: CONCLUSION

This chapter summarises the study by pointing out its achievements. It reviews the important findings as well as the limitations. The discussed limitations highlight potential areas for future improvement. A separate section is also dedicated to exploring future studies and how this work could be improved.

7.1 Research Contributions and Achievement of Objectives

This study began by providing an overview of the evolution of mobile malware since its inception. It then explored various components of the Android operating system and its security features, to establish the necessary fundamentals for discussing current research works. Subsequently current research studies were reviewed, which involved categorizing them into different groups based on their analysis and detection methods, as well as the used features. Afterwards, based on shortfalls of the reviewed works and available gaps, the study proposed a framework named DroidProtect. This framework was evaluated using real-world malware to examine its benefits over the related systems. The achievements of this study are detailed as follows.

1. *Comprehensive analysis of the most related and salient works*. We started by studying related works published in the span of five years. They were categorized into four different perspectives (Chapter 2). First, we studied them based on the Android features they used, as deciding what features to choose for analysis is an important step. Second, malware analysis methods were scrutinized, which is how malware families were analysed. Third, detection approaches of the current studies were reviewed. Lastly, we investigated the location in which the detection process is performed. The tools available for malware analysis and detection were also reviewed, as they helped to carry out our experiments (Chapter 3). This satisfies the first objective of this study (Section 1.4).

2. *Feature engineering.*[16] This work paid special attention to feature selection, as it determines the outcome of experiments (Section 2.4.1). Android Intent (explicit and implicit) was chosen as a static feature. To the best of our knowledge, analysis of this Android feature is unprecedented. Extensive analysis showed that this feature is indeed effective, especially when combined with other static features such as Android permission (Section 5.2.1.1). The achievements in this field are the selection of TCP and HTTP protocols for analysis and the analysis of 50,000 applications, which is a novel approach (Section 5.3.1).

3. *A framework for mobile malware analysis and detection*. Based on drawbacks and available gaps in the literature, we proposed a framework for mobile malware analysis and detection (Chapter 4). The framework uses hybrid analysis, which consists of static and dynamic analyses. Android Intent and network traffic were chosen as static and dynamic features respectively (Chapter 2). The novelty of the framework is based on the fact that feature collection and extraction are performed on the device and the results are sent to remote servers for further analysis. Current approaches perform the whole analysis on the device that in turn consumes more energy, or they send whole applications to the server, which could be intercepted and thus pose a security risk. This achievement fulfils our second objective (Section 1.4).

4. *Thorough evaluation of the proposed framework.* The proposed framework underwent extensive evaluation by using real-world malware. The objective of the evaluation was to examine whether the framework is effective enough in terms of malware detection. The static analysis component of the framework, named AndroDialysis, was evaluated using the Drebin data sample (Section 5.2.1.4). The dynamic analysis section, named AndroPsychology, was evaluated using an AndroZoo

[16] Feature engineering is the process of transforming raw data into features that better represent the underlying problem to the predictive models, resulting in improved model accuracy on unseen data.

data sample (Section 5.3). Results of the evaluations and the comparison to related works showed a high performance of the framework. This achievement fulfils our third objective (Section 1.4).

5. *Evaluating the energy consumption of the framework.* Based on the problem statement, the main objective of this study is to propose a framework that is effective and considers the limited battery resources of mobile devices. Thorough experiments were carried out to measure energy consumption of the framework under various situations. We calculated the energy consumed by static and dynamic components, as well as by using local detection and offloading detection methods, to show that the offloading method is more efficient. The comparison of the results with similar products shows that the proposed framework consumes less energy. This satisfies our last objective (Section 1.4).

6. *Implementation of the proposed framework.* In order to examine the feasibility and practicality of the framework, we implemented it as a client application on a mobile device and a remote server. We also designed a web module that displays detection results, as well as the detection history of the user. This design offers convenience to the user. This contribution ensures that the proposed system actually works in the real-world situation.

Overall, it is believed that the objectives of this study have been achieved (Section 1.4).

7.2 Limitations of This Study

As discussed in the previous section, this study has achieved its objectives. However, limitations were encountered during this process. The limitations that relate to this study are discussed in this section.

1. *Limitation of static analysis.* During the experiment, the decompilation process stopped responding due to RAM limitation. This issue occurred on applications with very large APK files. Low-size applications were running smoothly during the static analysis.

2. *The Android emulator.* This study required gathering network traffic for dynamic analysis. As running the whole data sample of 100,000 applications on a physical device was not possible, this process was carried out by running them on an emulator. Some malware families, e.g. Obad malware, are able to detect whether they run on an emulator and hide their malicious behaviour.

3. *The implementation of the framework.* The implementation of the proposed framework was presented in Chapter 6. The presentation shows the major components of the system. However, some aspects, such as different types of potential errors, or user interface experience, were not presented. Despite this limitation, the overall result of this work remains.

4. *Test on limited number of physical devices.* The implemented framework was tested on two physical devices, namely Samsung Galaxy Grand Quattro GT-I8552 and Sony Xperia Z3 Compact. Although these are medium and high-end devices respectively, it is beneficial to test the framework on various devices, to observe its performance and identify potential setbacks. Due to time and budget constraints, such evaluation was not possible.

7.3 Suggestions for Future Work

Although this study achieved its objectives, a number of suggestions for future studies have been identified. This section presents suggestions for future works based on the discussed limitations.

Attackers always try to evade detection methods by finding new ways to bypass such methods. Although this work experimented on a vast number of real-world malware families, it is advantageous to collect more samples of malware. This enables researchers to discover new behaviours and attack methods of malware families.

The development of an advanced version of the Android emulator would enable researchers to analyse more malware samples in less time, with more realistic results. The Android emulators lack some features compared to the real device, such as IMEI, routing table, timing attacks, sensory output, and serial number. Research on this issue would benefit the research community in the future.

REFERENCES

Aafer, Y., Du, W., & Yin, H. (2013, 2013/01/01). DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. *Proceedings of the 9th International Conference on Security and Privacy in Communication Networks*, Sydney, Australia, pp. 86-103.

Adeel, M., & Tokarchuk, L. N. (2011). Analysis of Mobile P2P Malware Detection Framework through Cabir & Commwarrior Families. *Proceedings of the Third International Conference on Privacy, Security, Risk and Trust*, Boston, USA, pp. 1335-1343.

Aftab, M. U. B., & Karim, W. (2014). *Learning Android Intents*: Packt Publishing.

Alan, H. F., & Kaur, J. (2016). Can Android Applications Be Identified Using Only TCP/IP Headers of Their Launch Time Traffic? *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, Darmstadt, Germany, pp. 61-66.

Allen, F. E. (1970). Control flow analysis. *SIGPLAN Not.*, 5(7), pp. 1-19.

Allix, K., Bissyand, T. F., Klein, J., & Traon, Y. L. (2016). AndroZoo: collecting millions of Android apps for the research community. *Proceedings of the 13th International Conference on Mining Software Repositories*, Austin, USA, pp. 468-471.

Almohri, H. M. J., Yao, D., & Kafura, D. (2014). DroidBarrier: know what is executing on your android. *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, San Antonio, Texas, USA, pp. 257-264.

Amos, B., Turner, H., & White, J. (2013). Applying machine learning classifiers to dynamic Android malware detection at scale. *Proceedings of the 2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, Sardinia, Italy, pp. 1666-1671.

AndroidPolice. (2011). The Mother Of All Android Malware Has Arrived. Retrieved 1st July, 2016, from http://www.androidpolice.com/2011/03/01/the-mother-of-all-android-malware-has-arrived-stolen-apps-released-to-the-market-that-root-your-phone-steal-your-data-and-open-backdoor/

Apvrille, L., & Apvrille, A. (2013). Pre-filtering mobile malware with Heuristic techniques. *Proceedings of the 2nd International Symposium on Research in Grey-Hat Hacking*, Grenoble, France,

Aresu, M., Ariu, D., Ahmadi, M., Maiorca, D., & Giacinto, G. (2015). Clustering Android Malware Families by Http Traffic. *Proceedings of the 10th International Conference on Malicious and Unwanted Software*, Puerto Rico, pp. 128-135.

Armando, A., Merlo, A., Migliardi, M., & Verderame, L. (2012). Would You Mind Forking This Process? A Denial of Service Attack on Android (and Some

Countermeasures). *Proceedings of the 27th Information Security and Privacy Conference (SEC 2012)*, Crete, Greece, pp. 13-24.

Arora, A., Garg, S., & Peddoju, S. K. (2014). Malware detection using network traffic analysis in android based mobile devices. *Proceedings of the Eighth International Conference on Next Generation Mobile Apps, Services and Technologies (NGMAST)*, Oxford, United Kingdom, pp. 66-71.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., & Rieck, K. (2014). DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. *Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium*, San Diego, USA, pp. 1-15.

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., McDaniel, P. (2014). FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Not.*, 49(6), pp. 259-269.

Au, K. W. Y., Zhou, Y. F., Huang, Z., & Lie, D. (2012). Pscout: analyzing the android permission specification. *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, Raleigh, NC, USA, pp. 217-228.

Aung, Z., & Zaw, W. (2013). Permission-Based Android Malware Detection. *International Journal of Scientific & Technology Research*, 2(3), pp. 228-234.

AVG.ThreatLabs. (2013). Android/Dowgin. Retrieved 1st July, 2016, from http://www.avgthreatlabs.com/virus-and-malware-information/info/android-dowgin/

Backes, M., Gerling, S., Hammer, C., Maffei, M., & Styp-Rekowsky, P. v. (2013). AppGuard: enforcing user requirements on android apps. *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Rome, Italy, pp. 543-548.

Baeza-Yates, R.A., & Ribeiro-Neto. (1999). *Modern Information Retrieval*. Boston: Addison-Wesley Longman Publishing Co.

Baliga, A., Bickford, J., & Daswani, N. (2013). Titan: A Carrier-based Approach for Detecting and Mitigating Mobile Malware: AT&T.

Barbera, M. V., Kosta, S., Mei, A., & Stefa, J. (2013, 14-19 April 2013). To offload or not to offload? The bandwidth and energy costs of mobile cloud computing. *Proceedings of the IEEE INFOCOM*, Turin, Italy, pp. 1285-1293.

Barbera, M. V., Kosta, S., Stefa, J., Hui, P., & Mei, A. (2012, 3-5 Sept. 2012). CloudShield: Efficient anti-malware smartphone patching with a P2P network on the cloud. *Proceedings of the 12th International Conference on Peer-to-Peer Computing (P2P)*, Tarragona, Spain, pp. 50-56.

Bente, I. (2013). *Towards a network-based approach for smartphone security*. Universität der Bundeswehr München.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *The Journal of Machine Learning Research,* 13(1), pp. 281-305.

Bielza, C., & Larrañaga, P. (2014). Discrete Bayesian network classifiers: a survey. *ACM Computing Surveys (CSUR),* 47(1), pp. 5.

Blasing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., & Albayrak, S. (2010, 19-20 Oct. 2010). An Android Application Sandbox system for suspicious software detection. *Proceedings of the 2010 5th International Conference on Malicious and Unwanted Software (MALWARE)*, Nancy, France, pp. 55-62.

Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for Android. *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, Chicago, Illinois, USA, pp. 15-26.

Chakradeo, S., Reaves, B., Traynor, P., & Enck, W. (2013). MAST: triage for market-scale mobile malware analysis. *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, Budapest, Hungary, pp. 13-24.

Chang, H. C., Agrawal, A. R., & Cameron, K. W. (2011, Nov. 30 2011-Dec. 2 2011). Energy-aware computing for android platforms. *Proceedings of the 2011 International Conference on Energy Aware Computing*, Istanbul, Turkey, pp. 1-4.

Chekina, L., Mimran, D., Rokach, L., Elovici, Y., & Shapira, B. (2012). Detection of deviations in mobile applications network behavior. *Proceedings of the Annual Computer Security Applications Conference*, Orlando, USA, pp. 1-10.

Chen, B. X., & Bilton, N. (2014). Building a Better Battery. Retrieved 1st February, 2014, from http://www.nytimes.com/2014/02/03/technology/building-a-better-battery.html?_r=0

Chen, J., Alalfi, M. H., Dean, T. R., & Zou, Y. (2015). Detecting Android Malware Using Clone Detection. *Journal of Computer Science and Technology,* 30(5), pp. 942-956.

Chen, X., Chen, Y., Ma, Z., & Fernandes, F. C. A. (2013). How is energy consumed in smartphone display applications? *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, Jekyll Island, Georgia, USA, pp. 1-6.

Chickering, D., Geiger, D., & Heckerman, D. (1995). Learning Bayesian networks: Search methods and experimental results. *Proceedings of the Fifth Conference on Artificial Intelligence and Statistics*, Florida, USA, pp. 112-128.

Chin, E., Felt, A. P., Greenwood, K., & Wagner, D. (2011). Analyzing inter-application communication in Android. *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, Bethesda, Maryland, USA, pp. 239-252.

Choi, S., Sun, K., & Eom, H. (2013). Android malware detection using library api call tracing and semantic-preserving signal processing techniques. *Proceedings of the International Conference on Security and Management (SAM)*, pp. 1.

CNET. (2013). Android dominates 81 percent of world smartphone market. Retrieved 1st February, 2014, from http://news.cnet.com/8301-1035_3-57612057-94/android-dominates-81-percent-of-world-smartphone-market/

Cohen, I., Sebe, N., Gozman, F. G., Cirelo, M. C., & Huang, T. S. (2003, 18-20 June 2003). Learning Bayesian network classifiers for facial expression recognition both labeled and unlabeled data. *Proceedings of the 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Wisconsin, USA, pp. I-595-I-601 vol.591.

Conti, M., Mancini, L. V., Spolaor, R., & Verde, N. V. (2015). Can't you hear me knocking: Identification of user actions on android apps via traffic analysis. *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, San Antonio, USA, pp. 297-304.

Cordy, J. R., & Roy, C. K. (2011). The NiCad clone detector. *Proceedings of the IEEE 19th International Conference on Program Comprehension (ICPC)*, Kingston, Ontario, Canada, pp. 219-220.

Creus, G. B., & Kuulusa, M. (2007). Optimizing Mobile Software with Built-in Power Profiling *Mobile Phone Programming: Application to Wireless Networking* (pp. 449-462). Dordrecht, Netherlands: Springer

Crussell, J., Gibler, C., & Chen, H. (2012). Attack of the Clones: Detecting Cloned Applications on Android Markets. *Proceedings of the 17th European Symposium on Research in Computer Security*, Pisa, Italy, pp. 37-54.

Dai, S., Tongaonkar, A., Wang, X., Nucci, A., & Song, D. (2013). Networkprofiler: Towards automatic fingerprinting of android apps. *Proceedings of the INFOCOM*, Turin, Italy, pp. 809-817.

Damopoulos, D., Menesidou, S. A., Kambourakis, G., Papadaki, M., Clarke, N., & Gritzalis, S. (2012). Evaluation of anomaly-based IDS for mobile devices using machine learning classifiers. *Security and Communication Networks,* 5(1), pp. 3-14.

Dash, S. K., Suarez-Tangil, G., Khan, S., Tam, K., Ahmadi, M., Kinder, J., & Cavallaro, L. (2016). DroidScribe: Classifying Android Malware Based on Runtime Behavior. *Proceedings of the Mobile Security Technologies (MoST 2016)*, San Jose, USA, pp. 1-12.

Deshotels, L., Notani, V., & Lakhotia, A. (2014a). DroidLegacy: Automated Familial Classification of Android Malware. *Proceedings of the ACM on Program Protection and Reverse Engineering Workshop*, San Diego, CA, USA, pp. 1-12.

Deshotels, L., Notani, V., & Lakhotia, A. (2014b). DroidLegacy: Automated Familial Classification of Android Malware. *Proceedings of the ACM SIGPLAN on*

Program Protection and Reverse Engineering Workshop, San Diego, CA, USA, pp. 1-12.

Desnos, A. (2010). Reverse engineering, Malware and goodware analysis of Android applications. Retrieved 1st September, 2016, from https://github.com/androguard/androguard/

Desnos, A. (2012, 4-7 Jan. 2012). Android: Static Analysis Using Similarity Distance. *Proceedings of the 2012 45th Hawaii International Conference on System Science (HICSS)*, Maui, USA, pp. 5394-5403.

Dini, G., Martinelli, F., Saracino, A., & Sgandurra, D. (2012). MADAM: A Multi-level Anomaly Detector for Android Malware. *Proceedings of the 6th International Conference on Mathematical Methods, Models and Architectures for Computer Network Security*, Saint Petersburg, Russia, pp. 240-253.

Duan, E. (2016). DressCode and its Potential Impact for Enterprises. Retrieved 1st January, 2017, from http://blog.trendmicro.com/trendlabs-security-intelligence/dresscode-potential-impact-enterprises/

Eder, T., Rodler, M., Vymazal, D., & Zeilinger, M. (2013, 2-6 Sept. 2013). ANANAS - A Framework for Analyzing Android Applications. *Proceedings of the 2013 Eighth International Conference on Availability, Reliability and Security (ARES)*, Regensburg, Germany, pp. 711-719.

Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., & Sheth, A. (2010). TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pp. 393-407.

Eset. (2013). Eset Virusradar. Retrieved 1st July, 2016, from http://www.virusradar.com/en/Android_Adware.Dowgin/chart/history

ESET. (2016). The Rise of Android Ransomware. Retrieved 1st July, 2016, from http://www.welivesecurity.com/wp-content/uploads/2016/02/Rise_of_Android_Ransomware.pdf

F-Secure. (2013). Android Accounted For 79% Of All Mobile Malware In 2012, 96% In Q4 Alone, Says F-Secure. Retrieved 1st October, 2016, from http://www.f-secure.com/static/doc/labs_global/Research/Mobile%20Threat%20Report%20Q4%202012.pdf

Faruki, P., Ganmoor, V., Laxmi, V., Gaur, M. S., & Bharmal, A. (2013). AndroSimilar: robust statistical feature signature for Android malware detection. *Proceedings of the 6th International Conference on Security of Information and Networks*, Aksaray, Turkey, pp. 152-159.

Feizollah, A., Anuar, N. B., Salleh, R., & Amalina, F. (2014). Comparative Evaluation of Ensemble Learning and Supervised Learning in Android Malwares Using Network-Based Analysis. *Proceedings of the 1st International Conference on Communication and Computer Engineering (ICOCOE)*, Melaka, Malaysia, pp. 1-11.

Feizollah, A., Anuar, N. B., Salleh, R., Amalina, F., Ma'arof, R. u. R., & Shamshirband, S. (2013). A Study Of Machine Learning Classifiers for Anomaly-Based Mobile Botnet Detection. *Malaysian Journal of Computer Science,* 26(4), pp. 251-265.

Feizollah, A., Anuar, N. B., Salleh, R., & Wahab, A. W. A. (2015). A review on feature selection in mobile malware detection. *Digital Investigation,* 13(C), pp. 22-37.

Felt, A. P., Chin, E., Hanna, S., Song, D., & Wagner, D. (2011). Android permissions demystified. *Proceedings of the 18th ACM Conference on Computer and Communications Security*, Chicago, Illinois, USA, pp. 627-638.

Feng, Y., Anand, S., Dillig, I., & Aiken, A. (2014). Apposcopy: semantics-based detection of Android malware through static analysis. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Hong Kong, China, pp. 576-587.

Flinn, J., & Satyanarayanan, M. (1999). PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, Washington DC, USA, pp. 2.

Fortinet. (2014). Fortinet's FortiGuard Labs Reports 96.5% Of All Mobile Malware Tracked Is Android-Based. Retrieved 1st July, 2016, from http://www.fortinet.com/resource_center/whitepapers/threat-landscape-report-2014.html

Friedman, N., Geiger, D., & Goldszmidt, M. (1997). Bayesian network classifiers. *Machine Learning,* 29(2-3), pp. 131-163.

Gartner. (2011). Gartner Says Worldwide Mobile Device Sales to End Users Reached 1.6 Billion Units in 2010. Retrieved 1st July, 2016, from http://www.gartner.com/newsroom/id/1543014

Gartner. (2013). Gartner Says Worldwide PC, Tablet and Mobile Phone Shipments to Grow 5.9 Percent in 2013 as Anytime-Anywhere-Computing Drives Buyer Behavior. Retrieved 1st February, 2014, from http://www.gartner.com/newsroom/id/2525515

Gartner. (2016). Gartner Says Five of Top 10 Worldwide Mobile Phone Vendors Increased Sales in Second Quarter of 2016. Retrieved 1st December, 2016, from http://www.gartner.com/newsroom/id/3415117

Gartner. (2017). Gartner Says Worldwide PC Shipments Declined 4.3 Percent in Second Quarter of 2017. Retrieved 1st September, 2017, from http://www.gartner.com/newsroom/id/3759964

Gascon, H., Yamaguchi, F., Arp, D., & Rieck, K. (2013). Structural detection of android malware using embedded call graphs. *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, Berlin, Germany, pp. 45-54.

Gianazza, A., Maggi, F., Fattori, A., Cavallaro, L., & Zanero, S. (2014). PuppetDroid: A User-Centric UI Exerciser for Automatic Dynamic Analysis of Similar Android Applications. http://arxiv.org/abs/1402.4826

Google. (2014). permission. Retrieved 1st April, 2016, from http://developer.android.com/guide/topics/manifest/permission-element.html

Google. (2016). Android Security 2015 Annual Report. Retrieved 1st July, 2016, from https://security.googleblog.com/2016/04/android-security-2015-annual-report.html

Grace, M., Zhou, Y., Wang, Z., & Jiang, X. (2012). Systematic Detection of Capability Leaks in Stock Android Smartphones. *Proceedings of the 19th Network and Distributed System Security Symposium*, San Diego, USA,

Grace, M., Zhou, Y., Zhang, Q., Zou, S., & Jiang, X. (2012). RiskRanker: scalable and accurate zero-day android malware detection. *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, Low Wood Bay, Lake District, UK, pp. 281-294.

Grossman, J. (2007). *XSS Attacks: Cross-site scripting exploits and defense*: Syngress.

Guido, M., Ondricek, J., Grover, J., Wilburn, D., Nguyen, T., & Hunt, A. (2013). Automated identification of installed malicious Android applications. *Digital Investigation,* 10, pp. S96-S104.

Gunasekera, S. A. (2012). *Android Apps Security*: Apress.

Guyon, I., & Elisseeff, A. (2003). An introduction to variable and feature selection. *The Journal of Machine Learning Research,* 3, pp. 1157-1182.

Hall, M. (1999). *Correlation-based Feature Selection for Machine Learning.* The University of Waikato, Hamilton, NewZealand. Retrieved from http://www.cs.waikato.ac.nz/~mhall/thesis.pdf

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1), pp. 10-18.

Ham, Y. J., & Lee, H.-W. (2014). Detection of Malicious Android Mobile Applications Based on Aggregated System Call Events. *International Journal of Computer and Communication Engineering*, 3(2), pp. 149-154.

Ham, Y. J., Moon, D., Lee, H.-W., Lim, J. D., & Kim, J. N. (2014). Android Mobile Application System Call Event Pattern Analysis for Determination of Malicious Attack. *International Journal of Security and Its Applications*, 8(1), pp. 241-236.

Hashem, I. A. T., Yaqoob, I., Anuar, N. B., Mokhtar, S., Gani, A., & Ullah Khan, S. (2015). The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47, pp. 98-115.

Hellman, E. (2013). *Android programming: Pushing the limits*: John Wiley & Sons.

Ho, T.-H., Dean, D., Gu, X., & Enck, W. (2014). PREC: practical root exploit containment for android devices. *Proceedings of the 4th ACM Conference on*

Data and Application Security and Privacy, San Antonio, Texas, USA, pp. 187-198.

Hochreiter, S., Bengio, Y., Frasconi, P., & Schmidhuber, J. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. Retrieved 1st January, 2017, from http://www.bioinf.jku.at/publications/older/ch7.pdf

Hoffmann, J., Neumann, S., & Holz, T. (2013). Mobile Malware Detection Based on Energy Fingerprints — A Dead End? *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Rodney Bay, Saint Lucia, pp. 348-368.

Houmansadr, A., Zonouz, S. A., & Berthier, R. (2011). A cloud-based intrusion detection and response system for mobile phones. *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 31-32.

Hu, G., Yuan, X., Tang, Y., & Yang, J. (2014). Efficiently, effectively detecting mobile app bugs with AppDoctor. *Proceedings of the Ninth European Conference on Computer Systems*, Amsterdam, Netherlands, pp. 1-15.

Huang, C. Y., Tsai, Y. T., & Hsu, C. H. (2013). Performance Evaluation on Permission-Based Detection for Android Malware. *Proceedings of the International Computer Symposium ICS 2012*, Hualien, Taiwan, pp. 111-120.

Huang, J., Zhang, X., Tan, L., Wang, P., & Liang, B. (2014). AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. *Proceedings of the 36th International Conference on Software Engineering*, Hyderabad, India, pp. 1036-1046.

Hyo-Sik, H., & Mi-Jung, C. (2013). Analysis of Android malware detection performance using machine learning classifiers. *Proceedings of the 2013 International Conference on ICT Convergence (ICTC)*, Jeju, South Korea, pp. 490-495.

Iland, D., Pucher, A., & Schauble, T. (2011). Detecting Android Malware on Network Level: UC Santa Barbara.

Ishii, Y., Watanabe, T., Akiyama, M., & Mori, T. (2016). Clone or Relative?: Understanding the Origins of Similar Android Apps. *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, New Orleans, Louisiana, USA, pp. 25-32.

Isohara, T., Takemori, K., & Kubota, A. (2011, 3-4 Dec. 2011). Kernel-based Behavior Analysis for Android Malware Detection. *Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security*, pp. 1011-1015.

Jang, J.-w., Kang, H., Woo, J., Mohaisen, A., & Kim, H. K. (2016). Andro-Dumpsys: Anti-malware system based on the similarity of malware creator and malware centric information. *Computers & Security*, 58(May 2016), pp. 125-138.

Jensen, R., & Shen, Q. (2008). *Computational Intelligence and Feature Selection: Rough and Fuzzy Approaches*. New Jersey, USA: John Wiley & Sons.

Jiang, X. (2011). New Sophisticated Android Malware DroidKungFu Found in Alternative Chinese App Markets. Retrieved 1st November, 2016, from https://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html

Jo, N. Y., Lee, K. C., & Park, B.-W. (2011). Exploring the optimal path to online game loyalty: Bayesian networks versus theory-based approaches *Ubiquitous Computing and Multimedia Applications* (pp. 428-437): Springer.

Jonge, A. d. (2011). *Essential App Engine: Building High-Performance Java Apps with Google App Engine*: Addison-Wesley.

Jung, W., Kang, C., Yoon, C., Kim, D., & Cha, H. (2012). DevScope: a nonintrusive and online power analysis tool for smartphone hardware components. *Proceedings of the eighth International Conference on Hardware/Software Codesign and System Synthesis (IEEE/ACM/IFIP)*, Scottsdale, AZ, USA, pp. 353-362.

Karami, M., Elsabagh, M., Najafiborazjani, P., & Stavrou, A. (2013, 18-20 June 2013). Behavioral Analysis of Android Applications Using Automated Instrumentation. *Proceedings of the 2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, pp. 182-187.

Kaspersky. (2012). Android OS Mobile Malware Up by 3x this 2012. Retrieved 1st July, 2016, from http://gb-sb.blogspot.my/2012/08/android-os-mobile-malware-up-by-3x-this.html

Kaspersky. (2013). Kaspersky Lab detects 315,000 new malicious files every day. Retrieved 1st July, 2016, from http://www.kaspersky.co.uk/about/news/virus/2013/Kaspersky_Lab_detects_315000_new_malicious_files_every_day_

Kaspersky. (2016a). Mobile malware evolution 2015. Retrieved 1st July, 2016, from https://securelist.com/analysis/kaspersky-security-bulletin/73839/mobile-malware-evolution-2015/

Kaspersky. (2016b). Rise of the Triada: mobile malware becomes very sophisticated. Retrieved 1st December, 2016, from https://business.kaspersky.com/rise-of-the-triada/5280/

Khune, R. S., & Thangakumar, J. (2012, 21-22 Dec. 2012). A cloud-based intrusion detection system for Android smartphones. *Proceedings of the 2012 International Conference on Radar, Communication and Computing (ICRCC)*, Tiruvannamalai, India, pp. 180-184.

Kim, B. (2016). *Learning Structured Representations for Geometry*. Swiss Federal Institute of Technology Zurich. Retrieved from http://e-collection.library.ethz.ch/eserv/eth:50260/eth-50260-01.pdf

Kim, D.-u., Kim, J., & Kim, S. (2013). A Malicious Application Detection Framework using Automatic Feature Extraction Tool on Android Market. *Proceedings of the*

3rd International Conference on Computer Science and Information Technology (ICCSIT'2013), Bali, Indonesia, pp. 4-5.

Knoernschild, K. (2010). Rich Mobile Application Platforms for the Smartphone 2010: Burton Group.

Kojadinovic, I., & Wottka, T. (2000). Comparision between a filter and a wrapper approach to variable subset selection in regression problems. *Proceedings of the European Symposium on Intelligent Techniques (ESIT)*, Aachen Germany, pp. 311-321.

Kou, X., & Wen, Q. (2011, 28-30 Oct. 2011). Intrusion detection model based on Android. *Proceedings of the 4th IEEE International Conference on Broadband Network and Multimedia Technology*, pp. 624-628.

Kumar, K., Liu, J., Lu, Y.-H., & Bhargava, B. (2013). A Survey of Computation Offloading for Mobile Systems. *Mobile Networks and Applications,* 18(1), pp. 129-140.

Larrañaga, P., Poza, M., Yurramendi, Y., Murga, R. H., & Kuijpers, C. M. (1996). Structure learning of Bayesian networks by genetic algorithms: A performance analysis of control parameters. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 18(9), pp. 912-926.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. Retrieved 1st January, 2017, from http://www.cs.toronto.edu/~hinton/absps/NatureDeepReview.pdf

Lee, S.-H., & Jin, S.-H. (2013). Warning system for detecting malicious applications on android system. *International Journal of Computer and Communication Engineering,* 2(3), pp. 324.

Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., McDaniel, P. (2015). IccTA: Detecting inter-component privacy leaks in Android apps. *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 280-291.

Liang, S., Keep, A. W., Might, M., Lyde, S., Gilray, T., Aldous, P., & Horn, D. V. (2013). Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, Berlin, Germany, pp. 21-32.

Lin, Y.-D., Lai, Y.-C., Chen, C.-H., & Tsai, H.-C. (2013). Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers & Security,* 39, Part B(November 2013), pp. 340-350.

Linzhang, W., Jiesong, Y., Xiaofeng, Y., Jun, H., Xuandong, L., & Guoliang, Z. (2004). Generating test cases from UML activity diagram based on gray-box method. *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, Washington DC, USA, pp. 284-291.

Liu, H., & Motoda, H. (2007). *Computational Methods of Feature Selection*. USA: CRC Press.

Lookout. (2010). Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild. Retrieved 21st April, 2014, from https://blog.lookout.com/blog/2010/12/29/geinimi_trojan/

Lu, H., Zhao, B., Su, J., & Xie, P. (2014). Generating lightweight behavioral signature for malware detection in people-centric sensing. *Wireless Personal Communications*, 75(3), pp. 1591-1609.

Lu, L., Li, Z., Wu, Z., Lee, W., & Jiang, G. (2012). CHEX: statically vetting Android apps for component hijacking vulnerabilities. *Proceedings of the 2012 ACM conference on Computer and communications security*, Raleigh, North Carolina, USA, pp. 229-240.

Luoshi, Z., Yan, N., Xiao, W., Zhaoguo, W., & Yibo, X. (2013). A3: Automatic Analysis of Android Malware. *Proceedings of the 1st International Workshop on Cloud Computing and Information Security*, Shanghai, China, pp. 89-93.

MacAfee. (2016). MacAfee Labs Threats Report. Retrieved 1st July, 2016, from http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2016.pdf

Machiry, A., Tahiliani, R., & Naik, M. (2013). Dynodroid: an input generation system for Android apps. *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, Saint Petersburg, Russia, pp. 224-234.

Maggi, F., Valdi, A., & Zanero, S. (2013). AndroTotal: a flexible, scalable toolbox and service for testing mobile malware detectors. *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, Berlin, Germany, pp. 49-54.

Martinelli, F., Marulli, F., & Mercaldo, F. (2017). Evaluating Convolutional Neural Network for Effective Mobile Malware Detection. *Procedia Computer Science*, 112, pp. 2372-2381.

MashableAsia. (2011). Android Captures Nearly 50% of Global Smartphone Market. Retrieved 1st July, 2016, from http://mashable.com/2011/08/02/android-market-share

Mikami, A. (2016). *Long Short-Term Memory Recurrent Neural Network Architectures for Generating Music and Japanese Lyrics*. (Ph.D.), Boston College. Retrieved from http://cslab1.bc.edu/~csacademics/pdf/16Mikami.pdf

Milin-Ashmore, J. (2016). 5 of the Most Dangerous Android Viruses and How to Get Rid of Them. Retrieved 1st September, 2017, from https://www.maketecheasier.com/dangerous-android-viruses/

Moonsamy, V., Rong, J., & Liu, S. (2013a). Mining permission patterns for contrasting clean and malicious android applications. *Future Generation Computer Systems*. http://www.sciencedirect.com/science/article/pii/S0167739X13001933

Moonsamy, V., Rong, J., & Liu, S. (2013b). Mining permission patterns for contrasting clean and malicious android applications. *Future Generation Computer Systems*, 36(July 2014), pp. 122–132.

Narudin, F. A., Feizollah, A., Anuar, N. B., & Gani, A. (2016). Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 20(1), pp. 343-357.

Nokia. (2016). Smartphone Infections Rise 96% In H1-2016: Malware Study. Retrieved 1st December, 2016, from http://www.darkreading.com/vulnerabilities---threats/smartphone-infections-rise-96--in-h1-2016-malware-study/d/d-id/1326949

PandaSecurity. (2011). Rootkits: almost invisible malware. Retrieved 1st July, 2016, from http://www.pandasecurity.com/homeusers/security-info/types-malware/rootkit/

Pandita, R., Xiao, X., Yang, W., Enck, W., & Xie, T. (2013). WHYPER: towards automating risk assessment of mobile applications. *Proceedings of the 22nd USENIX Security Symposium*, Washington, D.C, USA, pp. 527-542.

Pathak, A., Hu, Y. C., & Zhang, M. (2012). Where is the energy spent inside my app? fine grained energy accounting on smartphones with Eprof. *Proceedings of the 7th ACM European Conference on Computer Systems*, Bern, Switzerland, pp. 29-42.

Paturi, A., Cherukuri, M., Donahue, J., & Mukkamala, S. (2013, 20-24 May 2013). Mobile malware visual analytics and similarities of Attack Toolkits (Malware gene analysis). *Proceedings of the International Conference on Collaboration Technologies and Systems (CTS)*, pp. 149-154.

Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Molloy, I. (2012). Using probabilistic generative models for ranking risks of Android apps. *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, Raleigh, North Carolina, USA, pp. 241-252.

Polakis, I., Diamantaris, M., Petsas, T., Maggi, F., & Ioannidis, S. (2015). Powerslave: Analyzing the Energy Consumption of Mobile Antivirus Software. *Proceedings of the 12th Conference of Detection of Intrusions and Malware, and Vulnerability Assessment*, Milan, Italy, pp. 165-184.

Portokalidis, G., Homburg, P., Anagnostakis, K., & Bos, H. (2010). Paranoid Android: versatile protection for smartphones. *Proceedings of the Proceedings of the 26th Annual Computer Security Applications Conference*, Austin, Texas, USA, pp. 347-356.

Powell, M. L., & Miller, B. P. (1983). Process migration in DEMOS/MP. *ACM SIGOPS Operating Systems Review*, 17(5), pp. 110-119.

Ram, A. (2016). More bad news for the PC industry. Retrieved 1st November, 2016, from https://www.digitalnewsasia.com/personal-tech/more-bad-news-pc-industry

Rasthofer, S., Arzt, S., & Bodden, E. (2014, 14, 23-26 February 2014). A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks.

Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA,

Rasthofer, S., Arzt, S., Miltenberger, M., & Bodden, E. (2015). Harvesting runtime data in android applications for identifying malware and enhancing code analysis: Technische Universität Darmstadt.

Rastogi, V., Chen, Y., & Enck, W. (2013). AppsPlayground: automatic security analysis of smartphone applications. *Proceedings of the third ACM conference on Data and application security and privacy*, San Antonio, Texas, USA, pp. 209-220.

Rastogi, V., Yan, C., & Xuxian, J. (2014). Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks. *IEEE Transactions on Information Forensics and Security*, 9(1), pp. 99-108.

Reina, A., Fattori, A., & Cavallaro, L. (2013). A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *Proceedings of the 6th European Workshop on Systems Security*, Prague, Czech Republic, pp. 1-6.

Richardson, L. (2007). Beautiful soup documentation. Retrieved 1st April, 2016, from https://www.crummy.com/software/BeautifulSoup/bs4/doc/

Rosen, S., Qian, Z., & Mao, Z. M. (2013). Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. *Proceedings of the third ACM Conference on Data and Application Security and Privacy*, pp. 221-232.

Ruiz, C. (2005). Illustration of the K2 algorithm for learning Bayes net structures. Retrieved 1st January, 2017, from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.190.7306

Ruiz, F. (2012). Fakeinstaller leads the attack on android phones. Retrieved 1st July, 2016, from https://blogs.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones/

Sahs, J., & Khan, L. (2012). A Machine Learning Approach to Android Malware Detection. *Proceedings of the 2012 European Intelligence and Security Informatics Conference (EISIC)*, Odense, Denmark, pp. 141-147.

Saipullah, K. M., Anuar, A., Ismail, N. A., & Soo, Y. (2012). Measuring power consumption for image processing on android smartphone. *American Journal of Applied Sciences*, 9(12), pp. 2052.

Salehi, E., & Gras, R. (2009). An empirical comparison of the efficiency of several local search heuristics algorithms for Bayesian network structure learning. *Proceedings of the Learning and Intelligent Optimization Workshop (LION 3)*,

Samra, A. A. A., Yim, K., & Ghanem, O. A. (2013). Analysis of Clustering Technique in Android Malware Detection. *Proceedings of the 2013 Seventh International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, Taichung, Taiwan, pp. 729 - 733.

Sangkatsanee, P., Wattanapongsakorn, N., & Charnsripinyo, C. (2011). Practical real-time intrusion detection using machine learning approaches. *Computer Communications,* 34(18), pp. 2227-2235.

Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P., & Álvarez, G. (2013). PUMA: Permission Usage to Detect Malware in Android *International Joint Conference CISIS'12-ICEUTE´12-SOCO´12 Special Sessions* (Vol. 189, pp. 289-298): Springer Berlin Heidelberg.

Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., & Bringas, P. G. (2012, 14-17 Jan. 2012). On the automatic categorisation of android applications. *Proceedings of the 2012 IEEE Consumer Communications and Networking Conference (CCNC)*, Las Vegas, USA, pp. 149-153.

Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Nieves, J., Bringas, P. G., & Álvarez Marañón, G. (2013). MAMA: Manifest Analysis for Malware Detection in Android. *Cybernetics and Systems,* 44(6-7), pp. 469-488.

Sanz, B., Santos, I., Ugarte-Pedrero, X., Laorden, C., Nieves, J., & Bringas, P. (2014). Anomaly Detection Using String Analysis for Android Malware Detection. *Proceedings of the International Joint Conference SOCO'13-CISIS'13-ICEUTE'13*, Salamanca, Spain, pp. 469-478.

Sanz, B., Santos, I., Ugarte-Pedrero, X., Laorden, C., Nieves, J., & Bringas, P. G. (2013). Instance-based Anomaly Method for Android Malware Detection. *Proceedings of the 10th International Conference on Security and Cryptography*, Reykjavík, Iceland, pp. 387-394.

Sarma, B. P., Li, N., Gates, C., Potharaju, R., Nita-Rotaru, C., & Molloy, I. (2012). Android permissions: a perspective combining risks and benefits. *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, Newark, New Jersey, USA, pp. 13-22.

Senthamarai Kannan, S., & Ramaraj, N. (2010). A novel hybrid feature selection via Symmetrical Uncertainty ranking based local memetic search algorithm. *Knowledge-Based Systems,* 23(6), pp. 580-585.

Seo, S.-H., Gupta, A., Mohamed Sallam, A., Bertino, E., & Yim, K. (2014). Detecting mobile malware threats to homeland security through static analysis. *Journal of Network and Computer Applications*, 38, pp. 43-53.

Shabtai, A. (2010, 23-26 May 2010). Malware Detection on Mobile Devices. *Proceedings of the 2010 Eleventh International Conference on Mobile Data Management (MDM),* Kansas City, USA, pp. 289-290.

Shabtai, A., & Elovici, Y. (2010, 2010/01/01). Applying Behavioral Detection on Android-Based Devices. *Proceedings of the Mobile Wireless Middleware, Operating Systems, and Applications*, Chicago, IL, USA, pp. 235-249.

Shabtai, A., Fledel, Y., & Elovici, Y. (2010, 11-14 Dec. 2010). Automated Static Code Analysis for Classifying Android Applications Using Machine Learning.

Proceedings of the 2010 International Conference on Computational Intelligence and Security (CIS), Nanning, China, pp. 329-333.

Shabtai, A., Kanonov, U., & Elovici, Y. (2010). Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method. *Journal of Systems and Software*, 83(8), pp. 1524-1537.

Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., & Weiss, Y. (2012). Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1), pp. 161-190.

Shabtai, A., Tenenboim-Chekina, L., Mimran, D., Rokach, L., Shapira, B., & Elovici, Y. (2014). Mobile malware detection through analysis of deviations in application network behavior. *Computers & Security*, 43(June 2014), pp. 1-18.

Shalaginov, A., & Franke, K. (2013). Automatic rule-mining for malware detection employing neuro-fuzzy approach. *Norsk informasjonssikkerhetskonferanse (NISK)*.

Shneiderman, B., & Wattenberg, M. (2001). Ordered treemap layouts. *Proceedings of the IEEE Symposium on Information Visualization*, San Diego, California, USA,

Sicker, D. C., Ohm, P., & Grunwald, D. (2007). Legal issues surrounding monitoring during network research. *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, San Diego, California, USA, pp. 141-148.

Sohr, K., Mustafa, T., & Nowak, A. (2011). Software security aspects of Java-based mobile phones. *Proceedings of the 2011 ACM Symposium on Applied Computing*, Taichung, Taiwan, pp. 1494-1501.

Sophos. (2012). Angry Birds malware - Firm fined £50,000 for profiting from fake Android apps. Retrieved 1st July, 2016, from http://nakedsecurity.sophos.com/2012/05/24/angry-birds-malware-fine/

Sophos. (2013). Security Threat Report 2013. Retrieved 1st July, 2016, from http://www.sophos.com/en-us/medialibrary/PDFs/other/sophossecuritythreatreport2013.pdf

Spreitzenbarth, M., Freiling, F., Echtler, F., Schreck, T., & Hoffmann, J. (2013). Mobile-sandbox: having a deeper look into android applications. *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, Coimbra, Portugal, pp. 1808-1815.

StarCounter. (2016). Mobile internet usage surpasses desktop usage for the first time in history. Retrieved 1st September, 2017, from http://bgr.com/2016/11/02/internet-usage-desktop-vs-mobile/

Su, X., Chuah, M., & Tan, G. (2012). Smartphone Dual Defense Protection Framework: Detecting Malicious Applications in Android Markets. *Proceedings of the 2012 Eighth International Conference on Mobile Ad-hoc and Sensor Networks (MSN)*, Chengdu, China, pp. 153-160.

Suarez-Tangil, G. (2014). *Mining structural and behavioral patterns in smart malware.* (PhD), Universidad Carlos III de Madrid.

Suarez-Tangil, G., Tapiador, J. E., Lombardi, F., & Di Pietro, R. (2016). ALTERDROID: Differential Fault Analysis of Obfuscated Smartphone Malware. *IEEE Transactions on Mobile Computing,* 15(4), pp. 789-802.

Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., & Blasco, J. (2014). Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications,* 41(4, Part 1), pp. 1104-1117.

Symantec. (2013). Android Madware and Malware Trends. Retrieved 1st August, 2017, from http://www.symantec.com/connect/blogs/android-madware-and-malware-trends

Symantec. (2014a). The Future of Mobile Malware. Retrieved 1st July, 2016, from http://www.symantec.com/connect/blogs/future-mobile-malware

Symantec. (2014b). Simplocker: First Confirmed File-Encrypting Ransomware for Android. Retrieved 1st July, 2016, from http://www.symantec.com/connect/blogs/simplocker-first-confirmed-file-encrypting-ransomware-android

Symantec. (2015). The dawn of ransomwear: How ransomware could move to wearable devices. Retrieved 1st July, 2016, from http://www.symantec.com/connect/blogs/dawn-ransomwear-how-ransomware-could-move-wearable-devices

Tarkoma, S., Siekkinen, M., Lagerspetz, E., & Xiao, Y. (2014). *Smartphone Energy Consumption: Modeling and Optimization*: Cambridge University Press.

Tchakounté, F., & Dayang, P. (2013). System calls analysis of malwares on android. *International Journal of Science and Technology,* 2(9), pp. 669-674.

Techopedia. (2016). What is Mobile Malware? Retrieved 1st July, 2016, from https://www.techopedia.com/definition/29477/mobile-malware

Teufl, P., Ferk, M., Fitzek, A., Hein, D., Kraxberger, S., & Orthacker, C. (2013). Malware detection by applying knowledge discovery processes to application metadata on the Android Market (Google Play). *Security and Communication Networks,* 9(5), pp. 389–419.

The.Register. (2013). Earn £8,000 a month with bogus apps from Russian malware factories. Retrieved 1st July, 2016, from http://www.theregister.co.uk/2013/08/05/mobile_malware_lookout/

Tongaonkar, A., Dai, S., Nucci, A., & Song, D. (2013). Understanding mobile app usage patterns using in-app advertisements. *Proceedings of the International Conference on Passive and Active Network Measurement*, Hong Kong, pp. 63-72.

TrendMicro. (2010a). ANDROIDOS_DROIDSMS.A. Retrieved 1st July, 2016, from http://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/ANDROIDOS_DROIDSMS.A

TrendMicro. (2010b). Malicious Android App Spies on User's Location. Retrieved 1st July, 2016, from http://blog.trendmicro.com/trendlabs-security-intelligence/malicious-android-app-spies-on-users-location/

TrendMicro. (2012). A Brief History of Mobile Malware. Retrieved 1st July, 2016, from https://countermeasures.trendmicro.eu/wp-content/uploads/2012/02/History-of-Mobile-Malware.pdf

TrustGo. (2012). New Virus FakeLookout.A Discovered by TrustGo Security Labs. Retrieved 1st January, 2017, from http://blog.trustgo.com/fakelookout/

Vahala, J. *Prediction of Financial Markets Using Deep Learning.* Masaryk University. Retrieved from http://is.muni.cz/th/422802/fi_b/bakalarka_final.pdf

van Niedek, T. (2016). *Phonetic Classification in TensorFlow*. Radboud University. Retrieved from http://www.cs.ru.nl/bachelorscripties/2016/Timo_van_Niedek___4326164___Phonetic_Classification_in_TensorFlow.pdf

Varsha, M., Vinod, P., & Dhanya, K. (2016). Identification of malicious android app using manifest and opcode features. *Journal of Computer Virology and Hacking Techniques*, pp. 1-14.

Veen, V. v. d. (2013). *Dynamic Analysis of Android Malware.* (Master), VU University Amsterdam, Netherland. Retrieved from http://tracedroid.few.vu.nl/thesis.pdf

Virustotal. (2013). Antivirus. Retrieved 1st July, 2016, from https://www.virustotal.com/en/file/9add7b00a23efb96d487247d586a3be9878b1f3922a91ffa20e45398c873d5c3/analysis/

Walenstein, A., Deshotels, L., & Lakhotia, A. (2012). Program structure-Based feature selection for android malware analysis. *Proceedings of the International Conference on Security and Privacy in Mobile Information and Communication Systems*, pp. 51-52.

Webopedia. (2016). What is Mobile Malware? Retrieved 1st July, 2016, from http://www.webopedia.com/TERM/M/mobile_malware.html

Wei, X., Gomez, L., Neamtiu, I., & Faloutsos, M. (2012). ProfileDroid: multi-layer profiling of android applications. *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Istanbul, Turkey, pp. 137-148.

White, A. M., Krishnan, S., Bailey, M., Monrose, F., & Porras, P. (2013). Clear and Present Data: Opaque Traffic and its Security Implications for the Future. *Proceedings of the Network & Distributed System Security Symposium (NDSS)*, San Diego, USA, pp. 1-16.

Wiśniewski, R. (2010). ApkTool -A tool for reverse engineering Android apk files. Retrieved 1st September, 2016, from https://ibotpeaches.github.io/Apktool/

Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., & Wu, K.-P. (2012a, 9-10 Aug. 2012). DroidMat: Android Malware Detection through Manifest and API Calls Tracing. *Proceedings of the 2012 Seventh Asia Joint Conference on Information Security (Asia JCIS)*, Tokyo, Japan, pp. 62-69.

Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., & Wu, K.-P. (2012b). DroidMat: Android Malware Detection through Manifest and API Calls Tracing. *Proceedings of the Seventh Asia Joint Conference on Information Security (Asia JCIS)*, Tokyo, Japan, pp. 62-69.

Wu, X., Zhang, D., Su, X., & Li, W. (2015). Detect repackaged Android application based on HTTP traffic similarity. *Security and Communication Networks*, 8(13), pp. 2257-2266.

Xiao, X., Zhang, S., Mercaldo, F., Hu, G., & Sangaiah, A. K. (2017). Android malware detection based on system call sequences and LSTM. *Multimedia Tools and Applications*, pp. 1-21.

Xu, J., Yu, Y., Chen, Z., Cao, B., Dong, W., Guo, Y., & Cao, J. (2013). MobSafe: cloud computing based forensic analysis for massive mobile applications using data mining. *Tsinghua Science and Technology*, 18(4), pp. 418-427.

Yajin, Z., & Xuxian, J. (2012). Dissecting Android Malware: Characterization and Evolution. *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, San Fransico, USA, pp. 95-109.

Yan, L. J., & Cercone, N. (2010). Bayesian network modeling for evolutionary genetic structures. *Computers & Mathematics with Applications*, 59(8), pp. 2541-2551.

Yan, L. K., & Yin, H. (2012). DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. *Proceedings of the 21st USENIX Conference on Security symposium*, Bellevue, WA, USA, pp. 29-29.

Yang, Z. (2012). Powertutor-a power monitor for android-based mobile platforms (Vol. 2, pp. 19). University of Michigan.

Yang, Z., & Yang, M. (2012). Leakminer: Detect information leakage on android with static taint analysis. *Proceedings of the Third World Congress on Software Engineering (WCSE)*, pp. 101-104.

Yerima, S. Y., Sezer, S., & McWilliams, G. (2014). Analysis of Bayesian classification-based approaches for Android malware detection. *IET Information Security*, 8(1), pp. 25-36.

Yerima, S. Y., Sezer, S., McWilliams, G., & Muttik, I. (2013). A New Android Malware Detection Approach Using Bayesian Classification. *Proceedings of the 2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, Barcelona, Spain, pp. 121-128.

Yoon, C., Kim, D., Jung, W., Kang, C., & Cha, H. (2012). Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, USA, pp. 387-400.

Yu, J., Huang, Q., & Yian, C. (2016). DroidScreening: a practical framework for real-world Android malware analysis. *Security and Communication Networks*, 9(11), pp. 1435–1449.

Yu, L., & Liu, H. (2003). Feature Selection for High-Dimensional Data: A Fast Correlation-Based Filter Solution. *Proceedings of the 20th International Conference on Machine Learning*, Washington, DC, USA, pp. 856-863.

Yu, W., Ge, L., Xu, G., & Fu, X. (2014). Towards neural network based malware detection on android mobile devices. *Proceedings of the Cybersecurity Systems for Human Cognition Augmentation*, pp. 99-117.

Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R. P., Mao, Z. M., & Yang, L. (2010). Accurate online power estimation and automatic battery behavior based power model generation for smartphones. *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, Scottsdale, Arizona, USA, pp. 105-114.

Zhang, M., & Yin, H. (2014). AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA,

Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Zang, B. (2013). Vetting undesirable behaviors in android apps with permission use analysis. *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, Berlin, Germany,

Zhao, M., Ge, F., Zhang, T., & Yuan, Z. (2011a, 2011/01/01). AntiMalDroid: An Efficient SVM-Based Malware Detection Framework for Android. *Proceedings of the Second International Conference ICICA*, Qinhuangdao, China, pp. 158-166.

Zhao, M., Ge, F., Zhang, T., & Yuan, Z. (2011b). AntiMalDroid: An Efficient SVM-Based Malware Detection Framework for Android. *Proceedings of the 2nd International Conference on Intelligent Computing and Applications*, Qinhuangdao, China, pp. 158-166.

Zhao, M., Zhang, T., Ge, F., & Yuan, Z. (2012). RobotDroid: A Lightweight Malware Detection Framework On Smartphones. *Journal of Networks*, 7(4), pp. 715-722.

Zheng, C., Xiao, C., & Xu, Z. (2016). New Android Trojan "Xbot" Phishes Credit Cards and Bank Accounts, Encrypts Devices for Ransom. Retrieved 1st July, 2016, from http://researchcenter.paloaltonetworks.com/2016/02/new-android-trojan-xbot-phishes-credit-cards-and-bank-accounts-encrypts-devices-for-ransom/

Zheng, M., Sun, M., & Lui, J. (2013a). DroidAnalytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware. *arXiv*

preprint arXiv:1302.7212. Retrieved 1st October, 2013, from http://arxiv.org/abs/1302.7212

Zheng, M., Sun, M., & Lui, J. (2013b). DroidAnalytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware. *Proceedings of the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, Melbourne, Australia, pp. 163 - 171.

Zhou, W., Zhou, Y., Grace, M., Jiang, X., & Zou, S. (2013). Fast, scalable detection of "Piggybacked" mobile applications. *Proceedings of the Third ACM conference on Data and application security and privacy*, San Antonio, Texas, USA, pp. 185-196.

Zhou, W., Zhou, Y., Jiang, X., & Ning, P. (2012). Detecting repackaged smartphone applications in third-party android marketplaces. *Proceedings of the second ACM conference on Data and Application Security and Privacy*, San Antonio, Texas, USA, pp. 317-326.

Zhou, Y., Wang, Z., Zhou, W., & Jiang, X. (2012). Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, USA, pp. 5-8.

Zhu, D., Jin, H., Yang, Y., Wu, D., & Chen, W. (2017). DeepFlow: Deep learning-based malware detection by mining Android application for abnormal usage of sensitive data. *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*, Crete, Greece, pp. 438-443.

Zonouz, S., Houmansadr, A., Berthier, R., Borisov, N., & Sanders, W. (2013). Secloud: A cloud-based comprehensive and lightweight security solution for smartphones. *Computers & Security*, 37(September 2013), pp. 215-227.

LIST OF PUBLICATIONS AND PAPERS PRESENTED

Journal Papers:

Paper 1:

Feizollah, A., Anuar, N. B., Salleh, R., & Wahab, A. W. A. (2015). A review on feature selection in mobile malware detection. Digital Investigation, 13(C), pp. 22-37.

Paper 2:

Narudin, F. A., Feizollah, A., Anuar, N. B., & Gani, A. (2016). Evaluation of machine learning classifiers for mobile malware detection. Soft Computing, 20(1), pp. 343-357.

Paper 3:

Tam, K., Feizollah, A., Anuar, N. B., Salleh, R., & Cavallaro, L. (2017). The Evolution of Android Malware and Android Analysis Techniques. ACM Computing Surveys, 49(4), pp. 1-41.

Paper 4:

Feizollah, A., Anuar, N. B., Salleh, R., Suarez-Tangil, G., & Furnell, S. (2017). AndroDialysis: Analysis of Android Intent Effectiveness in Malware Detection. Computers & Security, 65(C), pp. 121-134.

Awards:

Ali Feizollah, Nor Badrul Anuar, Rosli Salleh., Malware Analysis and Detection System for Android Devices, IEEE Young Professionals Poster Competition, (2016), First Runner-up.

APPENDIX A: A LIST OF THE REVIEWED RESEARCH WORKS

This appendix includes the research works reviewed in this study.

Reference	Title	Objective
Peng *et al.* (Peng et al., 2012)	Using Probabilistic Generative Models for Ranking Risks of Android Apps	Using of the Android permissions in risk ranking the apps
Grace *et al.* (Grace, Zhou, Wang, et al., 2012)	Systematic Detection of Capability Leaks in Stock Android Smartphones	They analyzed permissions and java code of Android application to detect malicious applications that leak data from the device to the attacker
Wu *et al*. (D.-J. Wu et al., 2012a)	DroidMat: Android Malware Detection through Manifest and API Calls Tracing	Using permissions, API calls in code, intent and applying machine learning algorithms to detect malwares
Sanz *et al*.(Borja Sanz, Santos, Ugarte-Pedrero, et al., 2013)	Instance-based Anomaly Method for Android Malware Detection	They used applications' permissions and calculated the Manhattan distance, Euclidean distance and Cosine similarity to determine the deviation of an application from normal application
Sanz *et al.* (Borja Sanz, Santos, Laorden, Ugarte-Pedrero, Bringas, et al., 2013)	PUMA: Permission Usage to detect Malware in Android	They extracted applications' permissions and used machine learning algorithms to identify the malicious applications
Samra *et al.* (Samra et al., 2013)	Analysis of Clustering Technique in Android Malware Detection	Permissions were extracted from apk files and clustering technique was used to categorize the applications
Aung and Zaw (Aung & Zaw, 2013)	Permission-Based Android Malware Detection	Extracted permissions were fed to the k-means clustering algorithm for Android malware detection

Sanz *et al.* (Borja Sanz, Santos, Laorden, Ugarte-Pedrero, Nieves, et al., 2013)	MAMA: Manifest Analysis For Malware Detection In Android	Analysis of the Androidmanifest.xml file, which includes permissions, and machine learning algorithms to detect malwares
Aafer *et al.* (Aafer et al., 2013)	DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android	Extracting API calls and requested permissions and using classifiers to detect malwares
Zhou *et al.* (Zhou et al., 2013)	Fast, Scalable Detection of "Piggybacked" Mobile Applications	detecting piggybacked android applications through analyzing permissions and API calls in java code
Yerima *et al.* (Suleiman Y Yerima et al., 2014)	Analysis of Bayesian classification-based approaches for Android malware detection	Analyzing Android malwares via static analysis with Bayesian approach. They used permissions and java code as static features
Seo *et al.* (Seo et al., 2014)	Detecting mobile malware threats to homeland security through static analysis	Using permissions and API calls in java code to identify malicious applications pertaining to mobile banking, flight tracking and booking, etc.
Shabtai *et al.* (A. Shabtai et al., 2010)	Automated Static Code Analysis for Classifying Android Applications Using Machine Learning	They extracted some components from the java code such as strings, types, prototypes, methods, fields, static value, inheritance and opcodes. Machine learning methods applied to the extracted components for analysis.
Desnos (A. Desnos, 2012)	Android : Static Analysis Using Similarity Distance	It generates the signature from API calls, strings, exceptions and control flow. It then calculates the similarity between the generated signature and the malwares signature to detect malwares

Lu *et al.* (L. Lu et al., 2012)	CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities	CHEX analysis applications from the data-flow perspective.
Zhou *et al.* (W. Zhou et al., 2012)	Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces	They (DroidMOSS) extracted operands and opcodes from the java code. To confront obfuscation, they removed the operands and retained the opcodes, which are much harder to change.
Zheng *et al.* (M. Zheng et al., 2013a)	DroidAnalytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware	DroidAnalytics uses API calls to generate signature for a method. It then generates another signature based on methods in a class. The generated signatures are used to detect malwares
Yerima *et al.* (S. Y. Yerima et al., 2013)	A New Android Malware Detection Approach Using Bayesian Classification	The API calls in the java code are monitored for suspicious usage such as accessing messages or phone service
Deshotels *et al.* (Deshotels et al., 2014b)	DroidLegacy: Automated Familial Classification of Android Malware	DroidLegacy uses java code to create a signature for the application. It uses the API calls in the code
Suarez-Tangil *et al.* (Suarez-Tangil et al., 2014)	Dendroid: A Text Mining Approach to Analyzing and Classifying Code Structures in Android Malware Families	Dendroid analysis the structure of java code. It constructs the control flow graph which represents how the code executes in runtime

Rastogi *et al.* (V. Rastogi et al., 2014)	Catch Me if You Can: Evaluating Android Anti-malware against Transformation Attacks	It checks for the changes in the java code such as renaming of the class, method or field identifier; changing package name; code reordering by detecting *goto* instruction
Huang *et al.* (J. Huang et al., 2014)	AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction	AsDroid monitors the API calls related to the user interaction. Additionally, it monitors the user interaction with the device. The semantic mismatch of the two monitored events indicate a stealthy behavior of the application which is one of the main characteristics of the Android malwares
Rasthofer *et al.* (Rasthofer et al., 2014)	A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks	The authors extracted details of java code such as method name, return value type, parameter type, method modifier, class name, etc. They then used machine learning classifiers to analyze the collected details.
Burguera *et al*. (Burguera et al., 2011)	Crowdroid: Behavior-Based Malware Detection System for Android	Crowdroid is a cloud-based mobile malware detection system that processes system calls of the mobile devices. An agent application is installed on the device and logs system calls and sends it to the remote server for further analysis using machine learning technique.
Zhao *et al.* (Zhao et al., 2011a)	AntiMalDroid: An Efficient SVM-Based Malware Detection Framework for Android	It generates dynamic behavioral signature using system calls along other features. It then updates database of behavioral signatures.
Yan and Yin (L. K. Yan & Yin, 2012)	DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic	DroidScope uses system call as one of its features to analyze the malwares. In addition, it uses the virtualization method.

	Android Malware Analysis	
Su *et al*. (Su et al., 2012)	Smartphone Dual Defense Protection Framework: Detecting Malicious Applications in Android Markets	The authors developed dual protection system for mobile devices. The first layer is analyzing the system calls. They used numerous machine learning algorithms for their system
Khune and Thangakumar (Khune & Thangakumar, 2012)	A Cloud-Based Intrusion Detection System for Android Smartphones	They developed a cloud-based intrusion detection and recovery system using replicated and synchronized mobile devices on the cloud. System call is among several chosen features.
Reina *et al* (Reina et al., 2013)	A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors	CopperDroid collects and analyzes the system calls and the inter-process communication for Android malware detection.
Lin *et al* (Lin et al., 2013)	Identifying android malicious repackaged applications by thread-grained system call sequences	SCSdroid believes that the malicious behavior of malwares reflects in the system calls. It collects the sequence of the system calls and analyzes them to detect Android malwares.
Victor van der Veen (Veen, 2013)	Dynamic Analysis of Android Malware	TraceDroid collects system calls of the Android applications and analyzes them to identify the malware. It shows an improvement over similar systems
Ham and Lee (Ham & Lee, 2014)	Detection of Malicious Android Mobile Applications Based on Aggregated System Call Events	The author collected system calls from normal and malicious application and determined the Android malware

Ham *et al* (Ham et al., 2014)	Android Mobile Application System Call Event Pattern Analysis for Determination of Malicious Attack	They collected system calls of normal and malicious applications and analyzed their pattern. Through the similarity of the system calls, the malicious application is determined
Iland *et al* (Iland et al., 2011)	Detecting Android Malware on Network Level	It analyzes network traffic and looks for HTTP links to discover leaked data by malware. It parses the HTTP data to discover the transferred data such as IMEI, IMSI and credit card numbers
Wei *et al* (Wei et al., 2012)	ProfileDroid: Multi-layer Profiling of Android Applications	ProfileDroid is a multi-layer monitoring and profiling system. It has four layers, namely, static, user interaction, operating system and network traffic
Baliga *et al* (Baliga et al., 2013)	Titan: A Carrier-based Approach for Detecting and Mitigating Mobile Malware	Titan analyzes network traffic of the mobile devices. It uses several filter such as packet filter to inspect the network traffic
Zonouz *et al* (Zonouz et al., 2013)	Secloud: A Cloud-based Comprehensive and Lightweight Security Solution for Smartphones	Secloud uses cloud-based detection system. It emulates exact copy of the mobile device on the cloud. It analyzed the device using log data from a lightweight agent application on the device. It examines the network traffic and several other features
Feizollah *et al* (Feizollah et al., 2013)	a Study Of Machine Learning Classifiers For Anomaly-Based Mobile Botnet Detection	It collects network traffic and employs machine learning approach to train algorithms for Android malware detection
Maggi *et al* (Maggi et al., 2013)	AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors	AndroTotal collects system changes such as user interface, log files and network traffic. It then compare the collected data with the malware database

Shabtai *et al* (Shabtai et al., 2014)	Mobile Malware Detection through Analysis of Deviations in Application Network Behavior	They analyzed Android applications to discover pattern in network traffic. They used machine learning approach to train algorithms for anomaly detection
Blasing *et al.* (Blasing et al., 2010)	An Android Application Sandbox System for Suspicious Software Detection	AASandbox analyzes static and dynamic features. It extracts permissions and java code from the APK file and uses them as static features. It then installs the application; logs system calls, and uses it as dynamic feature.
Zhou *et al.* (Y. Zhou et al., 2012)	Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets	It extracts permissions and API calls from the APK file and collects system calls in runtime
Wei *et al.* (Wei et al., 2012)	ProfileDroid: Multi-layer Profiling of Android Applications	Examining Androidmanifest.xml and java code are static features chosen for ProfileDroid. User interaction, system calls and network traffic are dynamic features
Spreitzenbarth *et al.* (Spreitzenbarth et al., 2013)	Mobile-Sandbox: Having a Deeper Look into Android Applications	This system chose permissions, intents, java code and API calls as static feature; system calls, network traffic and user interaction as dynamic features.
Eder *et al.* (Eder et al., 2013)	ANANAS – A Framework For Analyzing Android Applications	ANANAS extracts static features from Androidmanifest.xml file and collects system calls, network traffic and file systems as dynamic features

Xu *et al.* (Xu et al., 2013)	MobSafe: Cloud Computing Based Forensic Analysis for Massive Mobile Applications Using Data Mining	MobSafe examines java code for static features and collects network traffic for dynamic analysis
Moonsamy *et al.* (Moonsamy et al., 2013a)	Mining Permission Patterns for Contrasting Clean and Malicious Android Applications	The authors collected the requested permissions from APK file, static feature, and required permissions from running the application, dynamic feature. The difference signifies the maliciousness of the application

The table below categorizes the reviewed works based on analysis type. It also mentions the number of malware samples they used for evaluation phase. It is worth mentioning that this study was conducted using 100,000 android applications, which is more than majority of the reviewed works. The number of malware samples ensures that the proposed system is evaluated with as many real-world malware samples as possible. Thus, the validity of the framework is ensured.

Reference	Approach	Number of Malware
(Zhemin Yang & Yang, 2012)	Static	1,750
(Arzt et al., 2014)	Static	-
(Suleiman Y Yerima et al., 2014)	Static	2,000
(A. Desnos, 2012)	Static	-
(Apvrille & Apvrille, 2013)	Static	-
(Aung & Zaw, 2013)	Static	500
(Grace, Zhou, Wang, et al., 2012)	Static	-
(Feng et al., 2014)	Static	-
(V. Rastogi et al., 2014)	Static	-
(Faruki et al., 2013)	Static	6,779
(Suarez-Tangil et al., 2014)	Static	1,231
(Rosen et al., 2013)	Static	2,782
(Peng et al., 2012)	Static	325,036
(Grace, Zhou, Zhang, et al., 2012)	Static	118,318
(L. Lu et al., 2012)	Static	5,486
(Crussell et al., 2012)	Static	9,400
(Sarma et al., 2012)	Static	158,062
(Samra et al., 2013)	Static	18,174
(Arp et al., 2014)	Static	129,013
(Deshotels et al., 2014a)	Static	1,100
(Luoshi et al., 2013)	Static	-
(Gascon et al., 2013)	Static	12,158
(Borja Sanz, Santos, Laorden, Ugarte-Pedrero, Bringas, et al., 2013)	Static	3,013

(Walenstein et al., 2012)	Static	-
(Borja Sanz, Santos, Laorden, Ugarte-Pedrero, Nieves, et al., 2013)	Static	666
(D.-J. Wu et al., 2012b)	Static	1,738
(J. Huang et al., 2014)	Static	125,249
(W. Zhou et al., 2012)	Static	91,093
(Aafer et al., 2013)	Static	20,000
(Lee & Jin, 2013)	Static	-
(S. Y. Yerima et al., 2013)	Static	2,000
(A. Shabtai et al., 2010)	Static	2,285
(Sahs & Khan, 2012)	Static	2,172
(M. Zheng et al., 2013b)	Static	150,368
(Borja Sanz et al., 2014)	Static	2,060
(Zhou et al., 2013)	Static	84,767
(C. Y. Huang et al., 2013)	Static	182
(Almohri et al., 2014)	Static	405
(M. Zheng et al., 2013b)	Static	24,009
(B. Sanz et al., 2012)	Static	2,144
(Paturi et al., 2013)	Static	-
(Seo et al., 2014)	Static	1,257
(Rasthofer et al., 2014)	Static	11,000
(Liang et al., 2013)	Static	52
(X. Wu et al., 2015)	Static	-
(Tchakounté & Dayang, 2013)	Dynamic	-
(Hyo-Sik & Mi-Jung, 2013)	Dynamic	14,794
(L. Yu & Liu, 2003)	Dynamic	-
(Shabtai & Elovici, 2010)	Dynamic	43
(Chekina et al., 2012)	Dynamic	10
(Backes et al., 2013)	Dynamic	-
(Baliga et al., 2013)	Dynamic	9
(Vaibhav Rastogi et al., 2013)	Dynamic	3,968
(Burguera et al., 2011)	Dynamic	-
(L. K. Yan & Yin, 2012)	Dynamic	-

(Dini et al., 2012)	Dynamic	56
(Enck et al., 2010)	Dynamic	30
(Portokalidis et al., 2010)	Dynamic	-
(Choi et al., 2013)	Dynamic	-
(Gianazza et al., 2014)	Dynamic	15
(Ham & Lee, 2014)	Dynamic	1,257
(Ham et al., 2014)	Dynamic	1,257
(Y. Zhang et al., 2013)	Dynamic	1,249
(Su et al., 2012)	Dynamic	120
(Maggi et al., 2013)	Dynamic	18,758
(Zhao et al., 2011b)	Dynamic	200
(Shabtai et al., 2014)	Dynamic	500,000
(Kou & Wen, 2011)	Dynamic	-
(Houmansadr et al., 2011)	Dynamic	-
(Iland et al., 2011)	Dynamic	18
(Amos et al., 2013)	Dynamic	1,738
(Karami et al., 2013)	Dynamic	20
(Damopoulos et al., 2012)	Dynamic	-
(Reina et al., 2013)	Dynamic	1,200
(Khune & Thangakumar, 2012)	Dynamic	-
(Zonouz et al., 2013)	Dynamic	-
(Isohara et al., 2011)	Dynamic	230
(Feizollah et al., 2013)	Dynamic	1,257
(Feizollah et al., 2014)	Dynamic	1,000
(Hoffmann et al., 2013)	Dynamic	-
(H. Lu et al., 2014)	Dynamic	331
(Lin et al., 2013)	Dynamic	100
(Asaf Shabtai et al., 2010)	Dynamic	5
(Veen, 2013)	Dynamic	-
(Bente, 2013)	Dynamic	-
(Machiry et al., 2013)	Dynamic	50
(Jang et al., 2016)	Dynamic	709
(Spreitzenbarth et al., 2013)	Hybrid	36,000

(Y. Zhou et al., 2012)	Hybrid	204,040
(Moonsamy et al., 2013b)	Hybrid	1,227
(Wei et al., 2012)	Hybrid	27
(Eder et al., 2013)	Hybrid	1,260
(Blasing et al., 2010)	Hybrid	-
(D.-u. Kim et al., 2013)	Hybrid	1,003
(Xu et al., 2013)	Hybrid	100,000
(M. Zheng et al., 2013b)	Hybrid	19
(Shalaginov & Franke, 2013)	Hybrid	604
(Guido et al., 2013)	Hybrid	-
(Teufl et al., 2013)	Metadata	-
(Pandita et al., 2013)	Metadata	-
(Kou & Wen, 2011)	Metadata	-

APPENDIX B: A COMPLETE LIST OF MALGENOME MALWARE FAMILIES

This appendix includes list of malware families available in the MalGenome data sample along with the number of samples, discovery date, and their characteristics.

No	Malware Family Name	No. of sample	Discovery Month	Characteristics
1	ADRD	22	2011-02	Sending out device info
2	AnserverBot	187	2011-09	Silently downloads an update for an application on run time containing a malicious code from a hacker
3	Asroot	8	2011-09	Root exploits without user permission
4	Basebridge	122	2011-06	Silently updates an application and downloads a malicious code from a hacker
5	BeanBot	8	2011-10	Sends out IMEI, IMSI and phone number, sends SMS to a premium number
6	BgServ	9	2011-03	Sends out IMEI, device info
7	CoinPirate	1	2011-07	Sends out device model, SDK version, IMEI, IMSI
8	CruseWin	2	2011-07	Deletes itself, deletes SMS, sends SMS to a premium number
9	DogWars	1	2011-08	Sends SMS to all the contacts in the phone without the user's awareness
10	DroidCoupon	1	2011-09	Root exploits without user permission
11	DroidDeluxe	1	2011-09	Root exploits without user permission
12	DroidDream	16	2011-03	Hijacks an application and controls the UI and performs commands received from a hacker
13	DroidDreamLight	46	2011-05	Sends out IMEI, IMSI, model, etc.

14	DroidKungFu1	34	2011-06	Malicious code is encrypted and it steals a user's phone number and sends it to a hacker
15	DroidKungFu2	30	2011-07	Malicious code is encrypted and it steals a user's phone number and sends it to a hacker
16	DroidKungFu3	309	2011-08	Malicious code is encrypted and it steals a user's phone number and sends it to a hacker
17	DroidKungFu4	96	2011-10	C&C server address is in the native program but in cipher text. It receives commands from a hacker
18	DroidKungFuSapp	3	2011-10	Sends out IMEI, phone info, data on SD card
19	DroidKungFuUpdate	1	2011-10	Remotely updates an application and downloads a malicious code from a hacker
20	Endofday	1	2011-05	Leaks user’s data via SMS
21	FakeNetflix	1	2011-10	Steals user's credentials and sends back to ground SMS messages.
22	FakePlayer	6	2010-08	Sends premium SMS without user's knowledge and steals user's phone number. Sends stolen data to a hacker
23	GamblerSMS	1	2011-07	Sends out incoming/outgoing SMS, outgoing phone call
24	Geinimi	69	2010-13	Makes phone calls in background and sends premium SMS. Commands are received from a hacker
25	GGTracker	1	2011-06	Apps advertisement redirects link to malicious web and malware subscribes premium-rate service.
26	GingerMaster	4	2011-08	Obfuscates the file names of associated root exploits
27	GoldDream	47	2011-07	Makes phone calls in background and sends premium SMS. Commands are received from a hacker

28	Gone60	9	2011-09	Sends out contacts, SMS, call list, visited URLs
29	GPSSMSSpy	6	2010-08	Listens to SMS-based commands to record and upload the victim's current location.
30	HippoSMS	4	2011-07	Sends out SMS to a premium number, deletes incoming SMS from a certain number.
31	Jifake	1	2011-10	Sends premium SMS without user's knowledge and steals user's phone number. Sends stolen data to a hacker.
32	jSMSHider	16	2011-06	Uses a publicly available private key by Android open source project and includes infected apps. Opens a backdoor.
33	Kmin	52	2011-10	Sends premium SMS without user's knowledge. Commands are received from a hacker.
34	Lovetrap	1	2011-07	Sends out IMSI and geo location
35	NickyBot	1	2011-08	Executes commands via SMS
36	Nickyspy	2	2011-07	Sends out call list, GPS, SMS
37	Pjapps	58	2011-02	Sends premium SMS without user's knowledge and steals user's phone number. Sends stolen data to a hacker
38	Plankton	11	2011-06	Downloads malicious code as an update from a hacker
39	RogueLemon	2	2011-10	Sends SMS and subscribes to service
40	RogueSPPush	9	2011-08	Sends premium SMS without user's knowledge by hiding confirmation SMS. Sends stolen data to a hacker
41	SMSReplicator	1	2010-11	Transmits incoming SMS to another device
42	SndApps	10	2011-07	Collects user's email addresses and sends them to remote sever.
43	Spitmo	1	2011-09	Steals user's sensitive banking information
44	TapSnake	2	2010-08	Sends out GPS info

45	Walkinwat	1	2011-03	Sends out name, phone number, IMEI
46	YZHC	22	2011-06	Sends SMS to a premium number
47	zHash	11	2011-03	Root exploits without user permission
48	Zitmo	1	2011-07	Steal user's sensitive banking information
49	zSone	12	2011-05	Hijacks an application, controls the UI and performs commands received from a hacker