Investigating Android Permissions and Intents for Malware Detection

Fauzia Idrees Abro

A dissertation submitted in partial fulfilment

of the requirements for the degree of

Doctor of Philosophy

of the

City, University of London

School of Mathematics, Computer Science and Engineering

2018

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed: Fauzia Idrees Abro

Date: 23 February 2018

Abstract

Today's smart phones are used for wider range of activities. This extended range of functionalities has also seen the infiltration of new security threats. Android has been the favorite target of cyber criminals. The malicious parties are using highly stealthy techniques to perform the targeted operations, which are hard to detect by the conventional signature and behaviour based approaches. Additionally, the limited resources of mobile device are inadequate to perform the extensive malware detection tasks. Impulsively emerging Android malware merit a robust and effective malware detection solution.

In this thesis, we present the ***PIndroid*** — a novel ***P***ermissions and ***In***tents based framework for identifying An***droid*** malware apps. To the best of author's knowledge, PIndroid is the first solution that uses a combination of permissions and intents supplemented with ensemble methods for malware detection. It overcomes the drawbacks of some of the existing malware detection methods. Our goal is to provide mobile users with an effective malware detection and prevention solution keeping in view the limited resources of mobile devices and versatility of malware

behavior. Our detection engine classifies the apps against certain distinguishing combinations of permissions and intents. We conducted a comparative study of different machine learning algorithms against several performance measures to demonstrate their relative advantages. The proposed approach, when applied to 1,745 real world applications, provides more than 99% accuracy (which is best reported to date). Empirical results suggest that the proposed framework is effective in detection of malware apps including the obfuscated ones.

In this thesis, we also present ***AndroPIn***—an **Andro**id based malware detection algorithm using ***P***ermissions and ***In***tents. It is designed with the methodology proposed in PInDroid. AndroPIn overcomes the limitation of stealthy techniques used by malware by exploiting the usage pattern of permissions and intents. These features, which play a major role in sharing user data and device resources cannot be obfuscated or altered. These vital features are well suited for resource constrained smartphones. Experimental evaluation on a corpus of real-world malware and benign apps demonstrate that the proposed algorithm can effectively detect malicious apps and is resilient to common obfuscations methods.

Besides PInDroid and AndroPIn, this thesis consists of three additional studies, which supplement the proposed methodology. First study investigates if there is any correlation between permissions and intents which can be exploited to detect malware apps. For this, the statistical significance test is applied to investigate the correlation between permissions and intents. We found statistical evidence of a strong correlation between permissions and intents which could be exploited to detect malware applications.

The second study is conducted to investigate if the performance of classifiers can further be improved with ensemble learning methods. We applied different ensemble methods such as bagging, boosting and stacking. The experiments with ensemble methods yielded much improved results.

The third study is related to investigating if the permissions and intents based system can be used to detect the ever challenging colluding apps. Application collusion is an emerging threat to Android based devices. We discuss the current state of research on app collusion and open challenges to the detection of colluding apps. We compare existing approaches and present an integrated approach that can be used to detect the malicious app collusion.

Acknowledgements

My last four and a half years have been challenging yet a life transforming experience. I never stopped working: reading, thinking and writing... and enjoying every minute of it. Throughout the course of my PhD study, everyone associated with my research has been incredibly brilliant and generous in their time and attention. I would like to thank all the people who contributed in some way in my PhD research for their unconditional and endless support, inspiration and encouragement.

Firstly, I would like to express my deepest gratitude to my supervisor Professor Muttukrishnan Rajarajan for his patience, motivation, immense knowledge and continuous support during my PhD study. His guidance helped me in all the time of research starting from selecting problems to work on, elaborating the problems, publishing the results to writing of this thesis. I could not have imagined having a better supervisor and mentor for my Ph.D study.

I would also like to thank my second supervisor Professor Tom Chen for his insightful comments, support and encouragement which helped me e to widen my research from various perspectives.

I am deeply grateful to Dr Yogachandran Rahulamathavan and Dr Suresh Veluru for being instrumental in refining my research endeavours. Without their support, the contributions and results from this research would not have been possible.

I also owe a great debt of gratitude to Erasmus Mundus Strong ties scholarship and City University London for sponsoring and supporting my PhD studies. Without their funding, it was not possible to pursue my PhD at the City University London.

I am sincerely grateful to my family and friends for supporting me throughout my PhD. Their patience, support and encouragements have been invaluable. I couldn’t ask for anything better than what I got from them as they were always a kindle of hope for me.

Lastly but certainly most importantly, I would like to thank Almighty for His countless blessings on me and my family.

Dedication

I dedicate this greatest academic milestone to my family — My parents (M. Yousuf and Mehr), who have always prayed for my success and supported me out rightly from my childhood to this day; my husband (Idrees), who has faith in me and always supported me unconditionally to fulfil my dreams; my children (Ali and Ezza) for their understanding and sacrificing the times I should have spent with them. Thank you for your unending support, unconditional love and thorough understanding.

Publications

1. Fauzia Idrees, Muttukrishnan Rajarajan, Mauro Conti, Thomas M. Chen and Rahulamathavan Yogachandran "***Pindroid: A novel Android malware detection system using ensemble learning methods.***" Computers & Security*,* 68, 36-46, 2017, Elsevier

2. Fauzia Idrees, Muttukrishnan Rajarajan and Thomas Chen, "***Mobile Malware detection with permissions and intents analysis.***" In 18th ACM Workshop on Mobile Computing Systems and Applications (HotMobile), USA, 2017, ACM

3. Fauzia Idrees, Muttukrishnan Rajarajan, Mauro Conti, Thomas M. Chen and Rahulamathavan Yogachandran ***"AndroPIn: Correlating Android Permissions and Intents Praxis for Malware Detection."*** In the 8th Int. conf. on IT, Electronics and Mobile communications (IEMCON), Canada, 2017, IEEE

4. Fauzia Idrees, Muttukrishnan Rajarajan, Thomas M. Chen and Rahulamathavan Yogachandran, **"Android *Application Collusion Demystified."*** In proceedings (Vol. 759, p. 176), Future Network Systems and Security: 3rd Int. conf. FNSS, USA, 2017, Springer

5. Fauzia Idrees, Muttukrishnan Rajarajan, Mauro Conti, Thomas M. Chen and Rahulamathavan Yogachandran, *"**AndroPIn: Correlating Android Permissions and Intents for Malware Detection."*** Poster presentation, the 6th N2Women Networking workshop, USA, 2017, ACM

6. Fauzia Idrees and Muttukrishnan Rajarajan, “***Investigating Android Permissions and Intents for Malware Detection.”*** In 10th Int. conf. on Wireless and Mobile Computing, Networking and Communications (WiMob), pp. 354-358, Cyprus, 2014, IEEE

7. Fauzia Idrees and Muttukrishnan Rajarajan, **"*War against mobile malware with cloud computing and machine learning forces.*"** In 3rd IEEE International conf. on Cloud Networking (CloudNet), pp. 278-280, Luxemburg, 2014, IEEE

8. Fauzia Idrees, Muttukrishnan Rajarajan, A. Y. Memon, **"*Framework for distributed and self-healing hybrid intrusion detection and prevention system*."** In 5th Int. conf. on ICT Convergence (ICTC), pp. 277-282, Korea, 2013, IEEE

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

This chapter presents malware threats to Android system, available malware detection solutions, their limitations and research gaps that this thesis aims to fill up with the novel contributions. Author's investigations of the problem, contributions, published work, methodology and implementation are elaborated in subsequent chapters. The thesis statement can be deduced as follows:

Permissions and intents used by Android applications can be used to efficiently and accurately distinguish malware whilst remaining resilient to code obfuscation.

1.2 Overview

In the past few years, smartphones have evolved from simple mobile phones into sophisticated computers. They are much more portable and consume less energy in comparison to personal computers. This fact extends their usage in business and home related activities such as surfing the Internet, Emails, SMS and MMS messages, online transactions and Internet banking, etc. All of these features make the smartphone a useful tool in our daily lives, but at the same time they render it more vulnerable to attacks by malicious applications [1]. Given that most users store sensitive information on their mobile phones, such as phone numbers, SMS messages, emails, pictures and videos, smart phones are a very appealing target for attackers and malware developers.

Android OS was introduced by Google in 2008 for smartphones and by the fourth quarter of 2010, Android became the market leader by taking over global market share of nearly 85%. In May 2012, the number of available apps in the Google Play Store amounted to 500,000 and exceeded 1.4 Million apps in the third quarter of 2014 and increased to 3 Million by March 2017 as shown in recently published statistics by Statista[1].

1https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

Android is a Linux kernel based operating system and its applications are written in Java language by using built-in APIs. Its security framework is comprised of application sandboxing, application signing, cryptographic APIs, secure inter-process communication using intents and permission model [3]. Permission model is a main security mechanism to control the misuse of vital hardware and software resources [4, 5]. To protect the system and users, Android requires apps to request permission before the apps can use certain system data and features. The system grants the permission automatically, or it may ask the user to approve the request if the permissions are required to access the sensitive areas. However, its effectiveness relies on the user's response and other built-in features, mainly the intents. Intent is a messaging object used to request an action from another app component. It facilitates the inter-process communication between components of the same or different applications.

Android being the market leader is the major target of Smartphone malware attacks [6]. The Android based mobile devices have been under constant attacks due to their ever increasing popularity and effortless development, improvising, re-packaging and publishing of apps [7, 8 and 9]. Malware targeting the Android platform has increased caustically over the last two years [10]. Situation is getting worse with the provision of installing third party applications and the increasing number of seemingly benign apps with malign activities. Android security framework has not proven effective in stopping the malware proliferation [11].

Existing end-point protections such as Anti-Virus software are unable to completely eliminate the malware threats [12, 13 and 14]. This is due to the fact that most of the solutions are signature based and need regular updates to protect against increasing number of malware variants and they lack obfuscation resilience [15, 16, 17 and 18]. There is a need for innovative and resource rich detection solutions to overcome the challenges of limited resources of mobile devices, outdated signatures of AV solutions and code obfuscation techniques used by the malware.

A lot of research had been done on permission based malware detection; however, intents were less explored till start of this thesis. Moreover, the major challenge to mobile malware detection is its limited resources that are characterized by short battery life, low memory and less processing power [19, 20 and 21].

We propose a novel methodology: PInDroid to fill up the research gap in the Android malware detection. Our goal is to provide mobile users with an effective malware detection method keeping in view the limited resources of mobile devices and versatility of malware behaviour.

We conducted a comparative study of six machine learning algorithms against different performance measures to select the best classifier for malware classification. Decision Table came up as a robust and most efficient classifier in our extensive validation experiments. Performance of the proposed approach is verified by applying the technique to the real world malicious and benign samples.

Different ensemble methods such as bagging, boosting and stacking are also investigated to ascertain if they can further improve the detection results. A significant improvement is attainable with application of ensemble methods.

The proposed methodology is also implemented as a malware detection algorithm: ***AndroPIn***, which has been tested on different real malware samples. The performance of AndroPIn is comparable to existing state-of-the-art solutions.

Additionally, permissions and intents are also investigated for detection of malicious colluding and obfuscated apps respectively. Colluding apps are those apps which cooperate using covert or overt communication means to perform a joint malicious action which they are not able to perform separately. Code Obfuscation is the process of modifying the code of app so that it is understandable. Malware writers deliberately obfuscate code to conceal its purpose or its logic in order to prevent someone reading the source code. While the process may modify actual method instructions or metadata, it does not alter the output of the program. The results of the studies on detection of colluding apps and obfuscated apps are presented in Chapters 6 and 7 respectively.

1.3 Research Questions

The thesis poses the following research questions:

Q1: Is a set of permissions able to distinguish malware from benign applications? (Chapter 3)

Q2: Can the occurrences of intents be used for discriminating a malware from a trusted application? (Chapter 3)

Q3: Is there any combinations of permissions and intents frequently used by malware which can classify malware and benign applications? (Chapter 3 and 4)

Q6: Can obfuscated malware apps be detected with the approach? (Chapter 3)

Q4: Can we ascertain which machine learning algorithm is best for mobile malware detection? (Chapter 4)

Q5: Can ensemble methods be applied to optimise the classifiers output? (Chapter 4)

Q7: Can we extend our approach to detect the ever increasing threat of colluding applications? (Chapter 5)

1.4 Contributions

The main contributions of this thesis are as follows:

- **Permissions and Intents amalgamation**. This is the first work which is combining two vital security mechanisms of Android OS - permissions and intents - for malware detection.

- **Investigation of inter-dependence between permissions and intents.** We accomplish an extensive evaluation of permissions and intents used by Android apps to understand their inter dependence and show how this interdependence could be used to stop the malware syndrome.

- **Machine learning algorithms comparison.** We conducted a comparative study of several machine learning algorithms to understand which classifier performs best to detect malware.

- **Ensemble methods for performance improvements.** We exploit different ensemble methods to ascertain if their application can further improve the detection accuracy.

- **Detection of colluding applications.** We investigate the mechanisms used by the colluding apps and explore the possibility of extending our approach to detect the colluding apps.

- **Detection of Obfuscated applications.** We investigate the obfuscation used by malicious apps and applied our approach to detect the obfuscated malicious apps.

- **Developing the malware detection algorithm.** We develop an algorithm to detect the malware using the permissions and intents of target apps.

1.5 Thesis Outline

The rest of the thesis is structured as follows:

Chapter 2 presents the introduction to Android system, applications taxonomies and architecture. It also presents an overview of malware, its types, evolution, propagation methods and characteristics and the malware detection approaches. A detailed survey on existing state of the art techniques used for malware detection is also delineated. Some of the most cited works are compared with our proposed approach and the research gap is identified.

Chapter 3 presents the two vital features of Android: Permissions and Intents which have been used in our proposed methodology for the detection of malware apps. The statistical correlation approach is also described which is used to test if there is any correlation between permissions and intents. Our categorization of permissions and intent into dangerous and normal types is also explained. Mostly used permissions and intents by malicious and benign apps are discussed. It also describes the outcomes of study carried out to investigate the distinguishing usage pattern of permissions and intents by malicious and benign apps and if these patterns can be exploited for malware detection. A malware detection algorithm based on permissions and intents: AndroPIn is also presented. Experimental results demonstrate that the AndroPIn can be used for malware detection.

Chapter 4 presents the proposed methodology of PInDroid. Machine Learning (ML) algorithms used to classify the malware apps are discussed. Comparison of ML algorithms against different performance metrics is described systematically. To validate the proposed approach, different experiments are carried out which are discussed with details of experimental setup and configurations. Results are discussed in terms of various performance measures such as True Positive Rate (TPR), False Positive Rate (FPR), accuracy and F-measure.

Different ensemble methods such as boosting, bagging and stacking are also presented that are used to optimize the classification results. To validate the proposed approach, different experiments were carried out which are discussed with details of experimental setup and configurations. Results are discussed in terms of accuracy.

Chapter 5 presents a research study on colluding apps. It investigates the attacking behaviour of app collusion and main features that facilitate the collusion. It also explores the possibility of applying the PInDroid methodology for detection of malicious colluding apps.

Chapter 6 concludes this thesis with a summary of its key achievements, challenges and open ended research questions which may be relevant to future research studies.

Chapter 2

A Survey on Android and Malware Detection Systems

2.1 Overview

A malware is (short for “malicious software”) is considered an annoying or harmful type of software intended to secretly access a device without the user's knowledge [22]. Android has become the most widely used OS for smartphones, therefore is target of growing attacks from cyber criminals [23]. The vulnerabilities of the operating system and applications are being exploited by the hackers to penetrate into the systems, steal user data and gain financial benefits [24, 25, and 26].

Android malware is evolving in a rapid manner. According to McAfee Security Company, its database contains more than 100 million samples as Android malware has increased multifold over the years [27]. Detection of new malware apps has become quite challenging due to new stealth techniques and encapsulation methods being used by malware. Existing Android antivirus solutions are less effective in detecting and combating highly sophisticated malware [28].

The objective of this chapter is to survey the state-of-the-art of malware detection approaches in order to identify specific factors affecting the performance of the malware detection systems, identify the state of the art analysis methods used to reduce the false positive rate and further investigate how these approaches can be improved on. The rest of this chapter is organized as follows: In Section 2.2, Android architecture, application framework and Android security architecture are discussed. Section 2.3 details different types of malware, the classifications of malware and Section 2.4 describes the malware analysis techniques, detection systems and identifies their strengths and limitations as well as most recent potential solutions to these limitations. Finally, Section 2.5 to 2.7 focuses specifically on feature based detection systems and details the analytical techniques used in such systems.

2.2 Android Operating System

Android platform was developed by Android Inc. in 2003 for the devices with limited resources (processing power, memory and storage space). It is based on a modified version of the Linux kernel version 2.6.25 [33]. Android architecture and its main components are discussed in the subsequent paragraphs.

2.2.1 Architecture

The Android architecture shown in Figure 2.1 is composed of several software stacks which that can be divided into three main groups: Linux Kernel/ Operating System (OS), Middleware and Applications. Green components are written in native code (C/C++), while blue items are Java components interpreted and executed by the Dalvik Virtual Machine. The red components belong to the Linux [2].

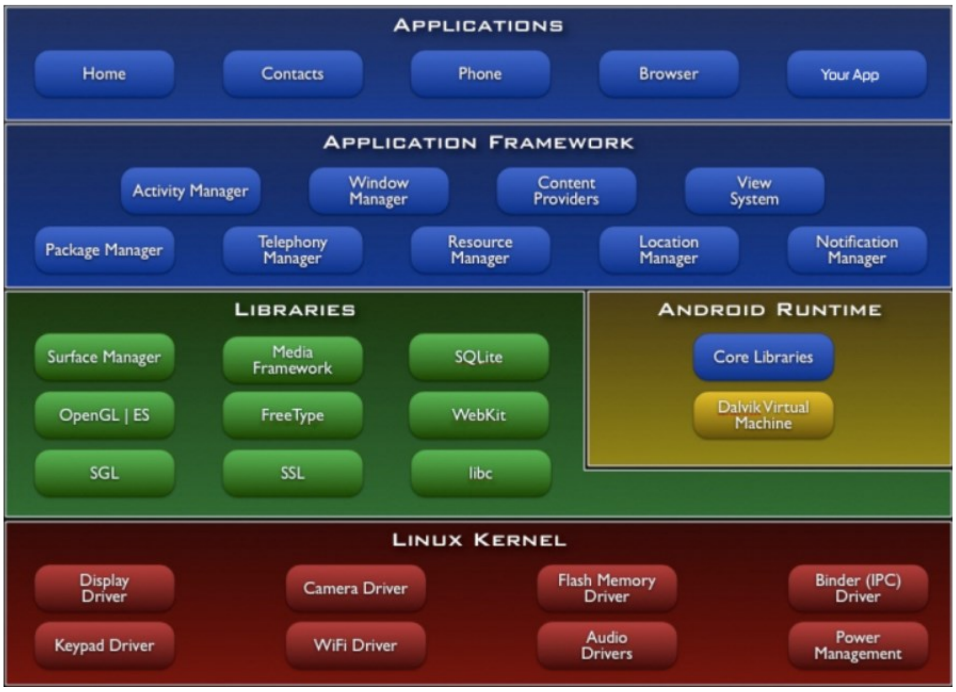


Figure 2.1: Android platform architecture

Source: http://elinux.org/Android_Architecture

2.2.1.1 Linux Kernel

Initial versions of Android OS were built on the Linux 2.6 kernel[1] with some architectural changes which include wake locks, a memory management system and the Binder IPC driver etc. Version 1.0 and above

1Computing Handbook, Third Edition: Computer Science and Software Engineering are based on the Linux 3.3 kernel. Linux kernel is the basic layer of Android which contains all the hardware drivers. It manages and processes requests for hardware resources.

2.2.1.2 Middleware

The middleware comprises of native libraries and Android Runtime.

- **Libraries**

  Android Libraries are written in C/C++ programming language and can be used through the Application framework. These are external libraries which are modified to make them compatible with ARM hardware and Android's implementation.

- **Android Runtime**

  Android Runtime includes Dalvik Virtual Machine (DVM) and core java libraries [34]. DVM is used to execute applications written in Java language.

  - **Dalvik Virtual Machine**

    DVM runs multiple VMs at the same time ensuring isolation, security and threading support without overloading the processor. DVM executes files in .dex file format which is an optimized java code for the low resource systems and are created from .class file during compilation [34].

- **Core Java Libraries**

These are implementation of general purpose APIs for use by the applications executed by the DVM.

2.2.1.3 Application framework

The android applications are developed by using some basic tools which manage the primary functions of device, for example, calls reception, text messaging and monitoring of battery usage etc. Some of the important blocks of Application framework are described below:-

- **Activity Manager**

  The activity Manager keeps the track of all active applications of the device and also inhibits the background processes in case of memory shortage. It also identifies those applications which do not respond to an input event for more than five seconds.

- **Content Providers**

  Content Providers are responsible for data sharing among different applications [2]. For example, photos and contact list can be accessed by multiple applications therefore these are stored in content provider.

- **Telephony Manager**

  Telephony manager manages the phone calls and also enables access to parameters like set's (IMEI).

- **Location Manager**

  The location Manager is responsible for providing the location services which are used by different applications to determine the geographical location by using embedded GPS or cell tower communication.

- **Resource Manager**

  Resource Manager manages the resources which are used by different applications.

2.2.1.4 Applications

Applications are the top most layer responsible for the interaction between user and the device. Mostly devices are pre-installed with some applications by the manufacturer to perform basic daily tasks like browser, e-mail, phone call, calculator, calendar etc) however users can install any app on their device from official or unofficial markets.

Android applications are written in the Java programming language. Android uses the Android Software Development Kit (SDK) and Java's programming environments, such as Eclipse or Netbeans to compile Java code and create an Android Package (APK) file. Applications are published with a unique Linux user ID and each application is granted its own VM to isolate it from the system resources and other applications.

2.2.2 Components of Application

Android applications come as .apk file which is signed ZIP files that contain the app's byte code along with all its data, resources, libraries and a manifest file [35]. The APK files are installed on the device using the Android Debug Bridge tool (adb) or by downloading them from Android's Market. An APK file consists of three main elements which are Manifest.xml, classes.dex and resources as shown in Figure 2.2.

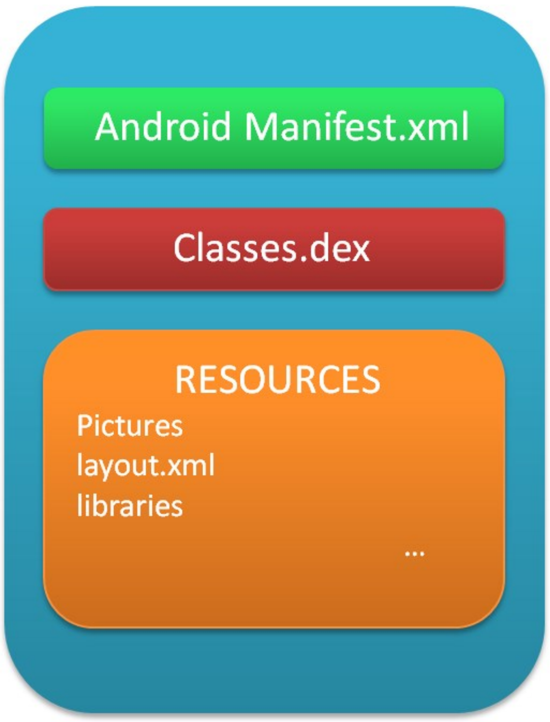
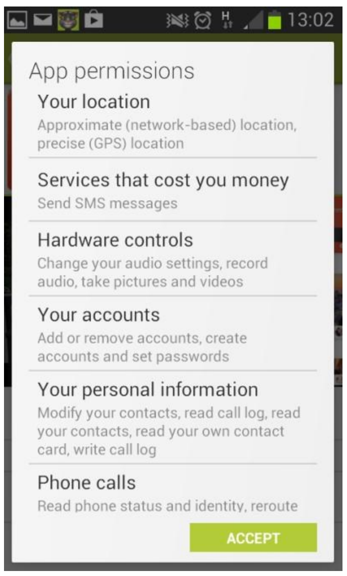


Figure 2.2: Android APK file


13:02

App permissions

Your location
Approximate (network-based) location, precise (GPS) location

Services that cost you money
Send SMS messages

Hardware controls
Change your audio settings, record audio, take pictures and videos

Your accounts
Add or remove accounts, create accounts and set passwords

Your personal information
Modify your contacts, read call log, read your contacts, read your own contact card, write call log

Phone calls
Read phone status and identity, reroute

Figure 2.3: Android Permissions screen

There are four protection levels assigned to the permissions depending on the capabilities and possible security risks. These groups are Signature or system, signature, dangerous and normal permissions. Android has an access system to check against these levels to ascertain that if the app should be granted the permission or not [36].

- **Classes .Dex**

This file is the compiled Java source code. It contains Dex byte code for the application and runs on the DVM [34].

- **Resources**

Resources include the libraries, files and pictures which are used by the application.

2.2 Android Security Model

Android is an open source operating system and securing an open source system requires robust and flexible security framework. Android's security is dependent upon the user's understanding of applications and system. Android security is mainly focused on the protection of user data and system resources as well as the application isolation. To achieve these goals, it relies on Linux kernel, application sandboxing, secure IPC, application signing and permissions [34]. Some of the key security features of Android security framework are discussed in subsequent sub paragraphs.

2.3.1 System and Kernel Level Security

Android provides conventional security guarantees of Linux kernel with an addition of secure IPC for maintaining the isolation between applications [34].

2.3.2 The Application Sandbox

Android uses Linux user-based protection for identification and isolation of applications. Android creates a kernel level sandbox for each application where each application has its own user id and it runs in its own process. Applications interaction with system resources and other applications is controlled with permissions. Application sandbox is equally effective in exercising same controls on the system applications and native code as it lies in the kernel level. The operating system libraries, application framework, their runtime, and applications run within the application sandbox [8, 34, 36].

2.3.3 File system Permissions

These permissions ensure privacy of user's information (files). In case of Android, files of one application are not shared with other applications unless the developer ensures such a provision [37].

2.3.4 Security-Enhanced Linux

Android uses Security-Enhanced Linux (SELinux) for access control. SELinux is a Linux kernel security module that provides a mechanism for supporting access control security policies including Mandatory Access Control (MAC) system [34]. It provides a mechanism to enforce the separation of information based on confidentiality and integrity requirements, which allows threats of tampering, and bypassing of application security mechanisms, to be addressed and enables the confinement of damage that can be caused by malicious or flawed applications. It includes a set of sample security policy configuration files designed to meet common, general-purpose security goals[1].

2.3.5 Android Permission Model

The Android applications have limited access to system resources. The permission model manages the access to system resources and restricts them by linking the access with permissions [38]. During the application installation phase, the permissions are requested to access the resources as a whole, thereby, linking the application installation with the grant of permissions [39]. Hence, denial is not an option for the intended user. Once granted, until recently, the permissions were for the entire duration of the installed application. However, in the latest versions of Android, the user can scroll and select/de-select the permissions. In such cases, some features of the app will not work due to non-availability of required permissions. Applications can also set their own permissions for other applications [40, 41]. The permissions are defined (how and who) in a protection level attribute which communicates with the system for the purpose [42, 43].

1https://en.wikipedia.org/wiki/Security-Enhanced_Linux

2.3.6 Protected APIs

The resources which are only accessible by the operating system only are called protected APIs [44]. The examples are Camera, GPS, telephony, Bluetooth, SMS/MMS and network/data. In order to use these resources, it is essential that the application defines them in its manifest.

2.3.7 Cost Sensitive APIs

APIs which may involve cost in their usage are categorized as cost sensitive APIs which include telephony, SMS/MMS, Data/Network and NFC [45]. These APIs are included in the OS controlled list of protected APIs for which an exclusive approval from the device's user is required [5].

2.3.8 Inter Components Communication (ICC)

Inter-Component Communication Android application consists of components. There are four kinds of components, activities, services, broadcasts and providers [46]. Android platform provides a secure ICC that is similar to IPC to the Unix system. ICC is provided by the binder mechanism which is in the middleware layer of Android. The binder is a remote procedure call that is from a custom Linux driver (Android Developers). ICC is achieved by intents. Intent is a message that shows the target with some data optionally [47]. It can be used in explicit communication if it identifies the name of the receiver, or used in the implicit communication that let the receiver see if it can access this intent or not.

Inter process communication takes place via traditional UNIX-type mechanism within the ambit of Linux permissions. Components of Android IPC are described below:

- **Binder**

  It is a Remote Procedure Call (RPC) mechanism to handle in-process and cross-process calls.

- **Services**

  Services can provide interfaces directly accessible using binder.

- **Intent**s

  Intent is a communication mechanism which tells the system about the intention to do some action [48, 49]. For example, if a website is to be opened, the 'intent' is sent to the system open the corresponding URL. The system would hand over the intent to the browser to carry out the action required by the intent.

- **Content Providers**

  A Content Provider facilitates to use the device's data [50] such as the contact list or music preferences. An application can access the data that is provided by the other applications through Content Provider, and it can define its own Content Providers to share its own data as well [51, 52].

2.3.9 Application Signing

Android requires that all apps be signed by the developers with a digital certificate before installing on app store. If the app is not digitally signed then its installation is blocked by the Google Play store and installer package. Application signing is used to identify the developer of app and to update the application without complicated procedures and further permissions [53]. It also facilitates the inter-app communication through well-defined IPC [54]. APK files contain the developer signature which is verified by the Package Manager [55]. Android does not carry out CA verification of application certificates. The app signing key creates a digital certificate which contains the public key of a public/private key pair, as well as some other metadata identifying the owner of the key. The owner of the certificate holds the corresponding private key. When a developer sign an APK, the signing tool attaches the public-key certificate to it and associates the APK to the developer and its corresponding private key. This helps Android ensure that future app are legitimate and from the creator of app. Every app must use the same certificate throughout its lifespan in order for users to be able to install new versions as updates to the app. Applications can share user ID if they are signed with the same certificate [56].

2.3.10 Sensitive User Data

Android has some APIs that may provide access to user data of protected APIs [34]. Sensitive user data is classified into three groups namely personal information, sensitive input devices and device metadata as shown in Figure 2.4.

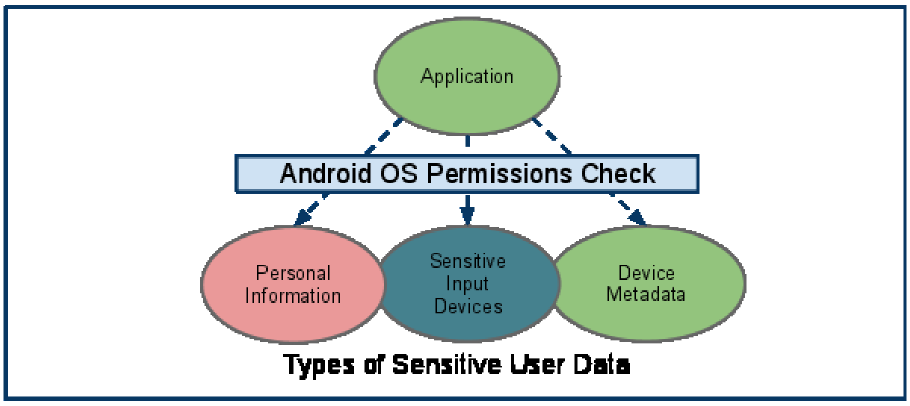


Figure 2. 4: Types of sensitive user data

Source: https://source.android.com/devices/tech/security/

▪ **Personal Information**

The content providers that contain personal information like contacts and calendar etc are controlled with clearly defined permissions users can get idea of the type of data which can accessed by the application. [34, 6]. Any application can access these resources if user grants the controlled permissions to the requesting app. By default, any application which collects personal information will restrict the data to the specific application; however, it can share the data with other applications using IPC and permissions mechanisms [57].

▪ **Sensitive Data Input Devices**

Android devices have sensitive data input sensors that allow many applications to interact with the external medium, such as GPS, microphone and camera. In case a third-party application requires accesses to these resources, it will need to request user to grant the permissions [5, 58].

- **Device Metadata**

  Android restricts access to sensitive data but it may share certain important information like user preferences or the manner in which user uses his device. The applications can only access the key resources with appropriate permissions. In case, permission is not granted, the installation will not proceed further [5, 6].

2.3.11 Publishing and Distribution of Apps

Publishing makes the Android apps ready for distribution to the users. Publishing involves two main tasks:

- **Preparation of the application for release**

  A release version of app is buildup which can be downloaded and installed on the Android devices.

- **Release of application to users**

  Application release involves the publicity, sell, and distribution of the release version of application to users [3]. Apps are released through app marketplaces, such as Google Play. However, apps can also be downloaded from some websites or through email. Android application is released on Google Play by configuring its options, uploading the assets and finally publishing the application [5].

2.4 Mobile Malware

Mobile Malware is malicious and an unwanted piece of software targeting mobile phones by damaging the device and loss or leakage of confidential data. First mobile malware surfaced in **2004** against Symbian operating system. First malware targeting Android was reported in **2010** and by **2011,** Android became the most favorite OS of malware encountering attacks every few weeks by new malware families [7]. Four types of the most common malware affecting mobile devices are expander, worm, Trojan and spyware. Expanders target mobiles for additional phone billing and profit. Worm endlessly reproduce itself and spread to other devices. Mobile worms may be transmitted via text messages SMS or MMS and typically do not require user interaction for execution [59]. Trojan horse always requires user interaction to be activated. This kind of virus is usually inserted into seemingly attractive and non-malicious executable files or applications that are downloaded to the device and executed by the user [60]. Once activated, the malware can cause serious damage by infecting and deactivating other applications or the phone itself, rendering it paralyzed after a certain period of time or a certain number of operations. Spyware poses a threat to mobile devices by collecting, using, and spreading a user's personal or sensitive information without the user's consent or knowledge.

2.4.1 Types of Android Malware

Mobile malware targeting Android smartphone is significant and growing at an alarming rate. This section briefly describes the common types of malicious programs targeting mobile phones. There are four broad categories[1] of mobile malware in addition to backdoor and worm malware. Backdoor helps other malware to enter the system without user knowledge by evading the system protections [61]. Worms make their copies and spread those copies through network or removable media.

(i) Trojans and Viruses

Viruses, worms, Trojans, and bots are all malware[2]. Trojans are those malware which look like some legitimate application but have hidden harmful malicious code which when executed inflicts serious damage to the device [62]. Trojanized apps are the biggest threat to the android devices as they can control the browser and steal account details including the bank login information. Trojans are viruses, which can be installed in different ways and can inflict damages ranging from simply annoying to highly-destructive and irreparable. Mobile viruses can root the device and gain unauthorized access to sensitive files and memory.

(ii) Spyware and Adware

Spyware are those malware which secretly steal user's data

1https://www.veracode.com/blog/2013/10/common-mobile-malware-types-cybersecurity-101
2 https://www.cisco.com/c/en/us/about/security-center/virus-differences.html

and shares with third parties for various purposes including the future attacks. In some cases these may be advertisers or marketing firms [63], which is why spyware is sometimes referred to as “adware”. Adware are those applications which are using ad libraries. They gather the user’s data to show relevant ads to the users for marketing purpose. Ad libraries cause privacy leaks and can frustrate the user by showing unwanted image or notifications repeatedly on the screen [64]. Spyware and Adware are typically installed without user consent by disguising itself as a legitimate app or by infecting its payload on a legitimate app.

(iii) Phishing Apps

Mobile phishing apps use same conventional web phishing techniques to infect the mobile devices. There are mobile phishing websites which look harmless but they covertly steal user’s credentials. The smaller screen of mobile devices is making malicious phishing techniques easier to hide from users. Some phishing schemes use Trojanized mobile apps, disguising their malicious action as a system update, marketing offer or game.

(iv) Botnets

A bot is a type of malware that allows an attacker to take control over an affected mobile device. Bots are usually part of a network of infected mobile phones, known as a “botnet”, which is typically made up of victim mobile phones that stretch across the globe. They allow hackers to take control of many mobile phones at a time, and turn them into "zombie" phones, which operate as part of a powerful "botnet" to spread viruses, generate spam, and commit other types of online crime and fraud. Botnets infect the device by accessing the device's resources and data; helping botnet masters to control the device. They exploit the system vulnerabilities and un-patched devices. They keep spreading over other devices by sending text messages or emails to the contacts of the infected device. Hidden processes can secretly run executable or contact bot masters for new instructions without user's knowledge. Future botnet are envisaged to have more serious damages and can completely hijack and control infected devices.

2.4.2 Malware Propagation

Malware use different sophisticated methods to spread over mobile devices [65]. Some of the widely used malware propagation methods are:

(i) **Infected websites**

Cybercriminals design malicious websites that exploit system vulnerabilities to spread the malware easily [66]. Mobile devices are infected when their users access such websites from the device.

(ii) **Third party app markets**

Third-party app stores have loose security controls over the applications developed and uploaded by unknown parties [67].

Malicious developers can upload Trojanized apps which can be downloaded by user, if the app has some appealing functionality. Third party stores also distribute the repackaged apps which are some popular apps installed with some malicious code, repackaged and distributed.

(iii) **Spam Emails and Botnets**

Propagating a malware by spam email is simple and effective propagation method. Attackers may send emails to the victims which appear to come from trusted sources such as the user’s bank, Amazon, Paypal or from own contacts. They contain links to some malicious website, compelling them to change their password and then sending the login information to a cybercriminal, or they may have infected attachments that immediately begin collecting data on their own once opened. Bots also propagate malware by sending text messages or e-mails to the contacts of infected user with a malicious link.

(iv) **Worms**

Mobile worms are similar to viruses in that they replicate themselves and can cause the damage. Unlike viruses, worms are standalone software and do not require an infected file or human help to propagate. They propagate over other devices through different exploits and system vulnerabilities.

(v) **Onscreen Adware**

Some attractive ads are run on user's screen as a sidebar with some game or other app, which when clicked by the user lands him on some malicious website.

(vi) **Dynamic Payload**

Hiding some malicious code in the APK resources file and executing it with Dex Class Loader API after installing it with the main application.

(vii) **App Updates**

Malicious code is hidden in updates which if installed by the user can infect the device.

2.5 Malware Detection Systems

Smartphone security and malware detection is an emerging research field where topics of publications are scattered within this domain. In this section, we present different most cited works on Android malware detection. We present related state of the art research studies, and systems developed by different researchers in Table 2.1.

2.5.1 Malware Analysis

A number of studies focus on analyzing Android's security mechanisms. Felt *et al.* [4] analysed the real mobile malware and carried out a ccomprehensive survey of behaviour of 46 malware samples related to three smartphone platforms (Android, iOS and Symbian) emerged between 2009 and 2011. In a similar type of study, Zhou et al. [7] covers 1260 Android malware samples distributed among 49 different malware families. Their findings confirm the increase in sophistication and obfuscation by the malware. Some researchers [59, 60, and 61] have proposed to rely on code clone detection techniques to identify similarity in repackaged and piggybacked apps. The piggybacked apps are those benign apps which are unpacked by malware writers and inserted with some malicious code then repackaged and distributed for free. A significant amount of research has been conducted into privacy leaks and ad libraries [62, 63, 64 65, 66, 67, 68, 69, 70]. Most of the proposed approaches are based on permission usage and other security risks, such as the potential to load and execute arbitrary byte code through the ad interface.

Due to known limitations of signature based methods, behaviour analysis has gained attention from anti-malware research community. One of such work is Risk Ranker [71] which targets zero-day malware samples. It examines the apps for presence of dangerous behaviours like using root exploits and SMS sending and classifies them according to the associated risk levels. Crowdroid [72] is also behaviour based approach which observes the run-time system calls to generate app profile and applies machine learning algorithms to distinguish between the malware and benign apps. Droidchamleon [73] evaluated the performance of ten commercial mobile anti-malware products against the common obfuscation techniques. Andromaly [74] used different network statistics for detecting deviations in application’s network behaviour. AppGuard [75] facilitates the enforcement of user-customizable security policies on untrusted Android apps. MADAM [76] proposed the analysis of kernel level features (CPU usage, system calls, memory usage etc) and user level features (key strokes, called numbers, SMS etc) to detect malware. Droidscope [77] provides sandboxed monitoring of app features at hardware, OS, and Dalvik Virtual Machine levels. PScout [78] proposd permission based behaviour analysis of malicious apps. Xmandroid [79] dynamically analysed the transitive permission usage to detect covert channels. Woodpecker [80] combined static and dynamic analysis to identify explicit and implicit leakage.

A considerable effort has been focussed on understanding the Android permission model as well as using it for the malware detection. Kirin [81] is more towards blocking the installation of apps that request dangerous combination of permissions, while Sarma et al. [82] assessed the permissions usage by the apps to evaluate the level of associated risks and [83] used the requested permissions to rebuild the malicious behaviours of apps to categorize them according to their security perceptive.

Another Android based security research track is towards the Inter-Component Communication (ICC) mechanism. Erika et al. [84] investigated the inter-application communication to verify the possible attacks and exploits of interacting components. Their tool ComDroid could be used by application developers to detect the application communication vulnerabilities. Long et al. [85] studied the component hijacking vulnerability of Android apps by inspecting the data flow activities and developed a static analysing tool CHEX which detects the component hijacking vulnerability. A similar tool EPICC was devised by Damien [39] to detect the ICC vulnerabilities, while [86] worked to prevent confused deputy and collusion attacks.

2.5.2 State of the Art approaches for Malware Detection

Some of the most cited works are highlighted in Table 2.1 along with the methodologies, advantages / disadvantages and years of publication. Most of the existing approaches are based on analysis of permissions, APIs or system calls. Permissions have been widely analyzed by the researchers, but intents were relatively untouched till the start of this work in 2013. Permissions and intents are either analyzed separately or combined with randomly selected features such as API calls, network statistics and memory usage etc.

Summarizing, there is no such a study which had investigated the correlation between permissions and intents. Our work is the first which investigates the inter-dependence of permissions and intents and how this correlation could be exploited for detecting the stealthy malicious activities. Our approach exploits the inherent inter-correlation and inter-dependence of these two mechanisms. Our approach benefits in terms of accuracy and efficiency by relying on low-dimensional and most relevant set of features. This work fills up the gap in the Android malware research.

The closest latest works are Marvin, Drebin and Droidmat since these approaches are using permissions and intents in addition to many other features like API calls, network statistics, components etc. However, there is no clear rational or study which uses intent exclusively as a feature for malware detection. Drebin and DroidMat use same feature set: Permissions, API calls, components, IPC, intent messages related to activate components only whilst Marvin uses permissions, API calls, dynamic loaded codes and intents related to broadcast receivers only.

Table 2.1 Overview of existing work

S No	Reference	Year	Methodology	Contributions
1.	AndroDialysis[48]	2017	Intents analysis to investigate their effectiveness to detect malware	Validation of intent as a decisive feature for malware detection
2.	Deep Android [189]	2017	It uses deep convolutional neural network (CNN). Malware classification with static analysis of the raw opcode sequence from a disassembled program	Applications need to be disassembled for analysis
3.	Stormdroid [161]	2016	Analysing static features (permissions, API calls), sequences and dynamic behaviours using ML techniques	It requires both static and dynamic analyses and root access for run time process analysis
4.	ICCDetector [185]	2016	Analysis of intents to detect the Inter component communication vulnerabilities	Focuses only to find out communication vulnerabilities
5.	APK Auditor[93]	2015	Permission-based malware detection using static analysis	Analysis is done on central server. Client needs an internet connection with the server for the malware analysis and detection
6.	CopperDroid [107]	2015	It carries out VMI-based dynamic analysis to reconstruct the behaviours of malware.	Needs root access to monitor the system calls
7.	TaintDroid[62]	2014	Dynamic taint tracking of API calls.	This can handle only privacy violations
8.	DroidMiner[63]	2014	Detection by generating Programming logics based behaviours of malicious apps	Non résilient to code transformation techniques

9.	DenDroid [64]	2014	Text mining and information retrieval based classification	System is unable to handle Code obfuscation
10.	Apposcopy [65]	2014	Semantics-Based Detection	Signature based approach thus detection scope limited to certain known malware.
11.	AndroSimilar [60]	2014	Signatures based AV solution to detect similar Android applications	Detection of repackaged applications only.
12.	AdRob[61]	2013	Permissions analysis	Study on impact of Android Application Plagiarism
13.	VetDroid[66]	2013	Permission Analysis	High computation cost and complex design
14.	AppProfiler[67]	2013	Static and dynamic analyses of API calls and permissions.	Can detect privacy leaks only.
15.	AppPlayground [53]	2013	Dynamic analysis of system calls, API calls and taint tracing.	Root access required
16.	Secloud[56]	2013	A cloud based system offering different detection techniques: SYS call monitoring, AV scanning and file integrity check	Root access required
17.	Epicc[68]	2013	Static analysis of ICC ,APIs and Intent	Limited to ICC vulnerabilities
18.	DroidChameleon [73]	2013	Different code transformation techniques implemented to evaluate the performance of ten anti-malware products for their resilience against malware transformations	Scope of work is to evaluate the anti-malware products and not the malware detection.

19.	Andromaly[74]	2013	Behaviour monitoring in terms of CPU usage, battery consumption and number of sent packets on WIFI.	Analysis is done on self-made malware apps only as they couldn't find any real world malware samples.
20.	DroidMOSS [59]	2012	Detection of repackaged applications with fuzzy Hashing technique.	Repackaged applications can only be detected
21.	MADAM [76]	2012	Detection with System calls and permissions	Limited dataset. In total 60 apps monitored (10 malicious and 50 benign)
22.	AppGuard [75]	2012	Permission misuse analysis	No detection of Malware
23.	DroidScope [77]	2012	Semantics based detection	Root access required
24.	RiskRanker [71]	2012	Analysis of root exploits, permissions, API calls, crypto, dynamic code, IPC.	Root-exploit detection scheme depends on signatures, which implies that it can detect only known exploits and may also miss encrypted or obfuscated exploits.
25.	PScout[78]	2012	Permissions analysis	
26.	Dr. Android and Mr. Hide[83]	2012	Permissions analysis	
27.	AdSplit[46]	2012	Permissions analysis	Separating smartphone advertising from applications
28.	AndroidLeaks [47]	2012		Privacy leaks only
29.	Woodpecker [69]	2012	Uses CFG and permission analysis	Complex design

30.	RobotDroid[70]	2012	API calls	Root access required
31.	PiOS [70]	2011	Uses CFG	Privacy leaks only
32.	Xmandroid[80]	2011	ICC analysis	
33.	Crowdroid [87]	2011	Monitors SYS calls, list of running applications and the device information.	Root access required
34.	Paranoid android[88]	2010	Dynamic analysis of API calls and Permissions.	Root access required
35.	Kirin [81]	2009	Detection by analysing certain combinations of permissions and API calls.	Detects privacy leaks only.

Chapter 3

Investigating Permissions and Intents for Malware Detection

3.1 Introduction

Permission model is a vital security mechanism which guards against the misuse of hardware and software resources; however, it relies on the user’s response and other built-in features such as intent, which is a communication mechanism which facilitates the use of different functionalities offered by the components of same application or other applications [95]. Intent spoofing and permission collusion are few examples of attacks due to misuse of intents [96, 97]. Although, a significant research work has been carried out to investigate the permission model and API calls for detection of mobile malware but less work is done on Intents.

This chapter investigates Android permissions and intents to understand their role in basic functionality of apps and how that role can be exploited by the cyber criminals for malicious attacks. Such an understanding will help in devising a novel malware detection solution based on permissions and intents to effectively detect the malicious activities. Malware analysis by combining the permissions and intents is carried out to deduce the usage pattern of these vital features that can be exploited to distinguish between the malware and benign apps.

We present an automated malware detection algorithm: AndroPIn which is based on permissions and intents declared in the Manifest file. Once declared, these vital features cannot be altered by code obfuscation or encryption, hence making our proposed approach resilient to code obfuscation.

The rest of the chapter is organized as follows: Section 3.2 presents the basic background information about permissions and intents, Section 3.3 discusses the investigation findings, and Section 3.4 presents the statistical testing details carried out to understand the correlation between permissions and intents. Section 3.5 presents the AndroPIn malware detection algorithm and Section 3.6 discusses its implementation aspects. Section 3.7 gives the details of experiments and result. Finally the Section 3.8 summarizes the chapter.

3.2 Background

Android has 117 permissions and 227 intents in version 4.4, API level 19 - an API level is an integer value which identifies the application’s compatibility with the Android versions. The earliest Android version: API level 1, contains only 76 permissions and 124 intents. Google adds new permissions and intents into every upcoming version. This trend is depicted in Table 3.1, where monotonic increment in permission and intents against the API levels is obvious. The increased number of permissions and intents has not only added new features but also opened the doors for malware. A meticulous analysis of permissions and intents used by the apps will help to construct the behavioural image of apps for malware detection.

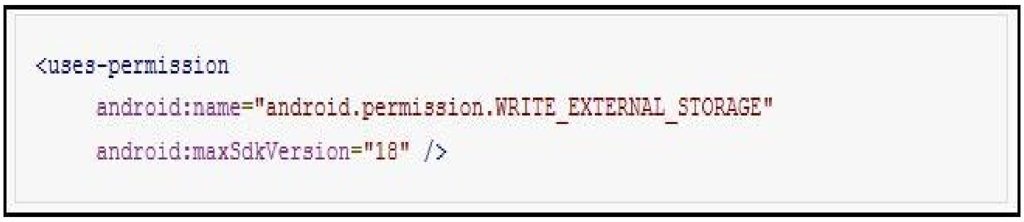
Table 3.1: Number of permissions and intents in API levels

API Level	No of Permissions	No of Intents
23	135	252
22	124	243
21	123	238
20	118	227
19	117	227
18	106	221
17	103	214
16	103	203
15	99	201
14	99	191
13	97	180
12	96	180
11	96	176
10	96	167
09	95	167
08	92	167
07	88	161
06	88	158
05	88	158
04	87	146
03	83	136
02	78	124
01	76	124

It is found during investigation of apps that certain permissions and intents are repetitively used by malware apps which can distinguish them from benign ones.

3.2.1 Android Permissions

Permission model is the basic security feature of android system which provides access to the vital organs of android based devices [98]. These are embedded in the manifest file of applications and declared as shown in Figure 3.1 [6].



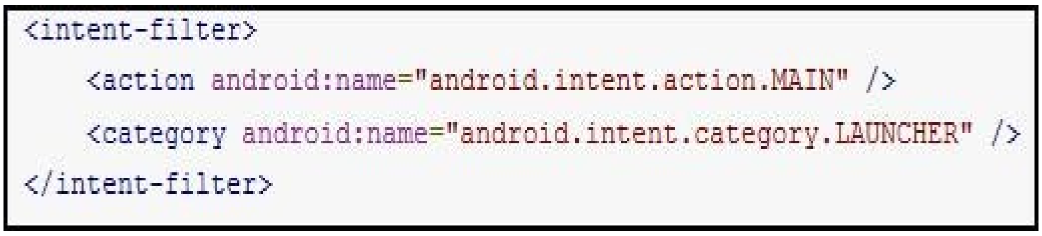
```
<uses-permission
     android:name="android.permission.WRITE_EXTERNAL_STORAGE"
     android:maxSdkVersion="18" />
```

Figure 3.1: Declaration of Android permission.

In earlier versions of Android, permissions were required to be granted as a whole and not in parts [99]. There was no choice to select the permissions from the offered ones; user had to accept all the permissions and install the app or reject and didn’t install. Once granted, these permissions would remain effective for the lifetime of the installed app until changed through an update [100]. However, on Android version 6.0 and above, user can control the installation of permissions with capability compromises.

3.2.2 Android Intents

Android intent is the basic communication mechanism used for exchanging inter and intra application messages. Functionalities and capabilities of different apps can be combined with the use of intents [101]. A malicious app may trick the user to install some other collaborating app for getting additional features. User is then prompted two different sets of permissions by these two different apps but because they share their functionalities cowardly through sending /receiving intents, the user being ignorant of this feature might install both apps which would harm his device [102]. Intents are embedded in the manifest file and declared as shown in Figure 3.2 [34].



```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

Figure 3.2: Declaration of Android Intents.

3.3 Analysis of Permissions and Intents

We carried out a comprehensive review of Android security framework and existing research work on malware detection to establish the distinguishing key features of Android apps which could facilitate the malware detection.

A total of 500 apps (270 malware and 230 benign) were analysed which are collected from well-known sources such as Google Playstore[1], Mobango[2]. Contagiodump[3], Genome[4], Virus Total[5], theZoo[6] and MalShare[7]. These sources contain the datasets of already known malware samples. The benign apps are selected from different categories such as social, news, entertainment, finance, education, games, sports, music, and audio, telephony, messaging, shopping, banking, and weather. Selection

1Google Play, Web: https://play.google.com/store?hl=en
2Web: http://www.mobango.com/
3Contagio Mobile: mobile malware mini dump, Web: http://contagiominidump.blogspot.co.uk/
4Android Malware Genome Project, Web: http://www.malgenomeproject.org/
5VirusTotal for Android, Web: https://www.virustotal.com/en/documentation/mobile-applications/
6theZoo aka Malware DB, Web: http://ytisf.github.io/theZoo/
7MalShare project, Web: http://malshare.com/about.php

of malware and benign samples is carefully done to learn the malicious and normal behaviour of apps.

Our investigation of Android security framework, the existing state-of-the-art malware detection approaches, Android features used by most of these approaches (permissions, intents, API calls, system calls, ICC) and analysis of benign and malware samples resulted in interesting finding: identification of key features: permissions and intents used for malware attacks and propagation.

We also establish that certain permissions and intents which are frequently used by malware apps are seldom used by benign apps. Malware families use a particular set of permissions and intents targeting specific capabilities and resources. Almost all the malware samples belonging to that particular family use a unique set of permissions and intents. This study resulted in some very interesting findings which are discussed in this section.

3.3.1 Permission usage by the Applications

Most famous benign apps like Facebook, YouTube, Skype and Viber tend to use on average 8-16 permissions while this number goes down to 3-6 for the least famous applications. Some trend prevails in malware apps- most harmful malware apps use on average more than 16 permissions and least harmful use 3-6 permissions as depicted in Figure 3.3.

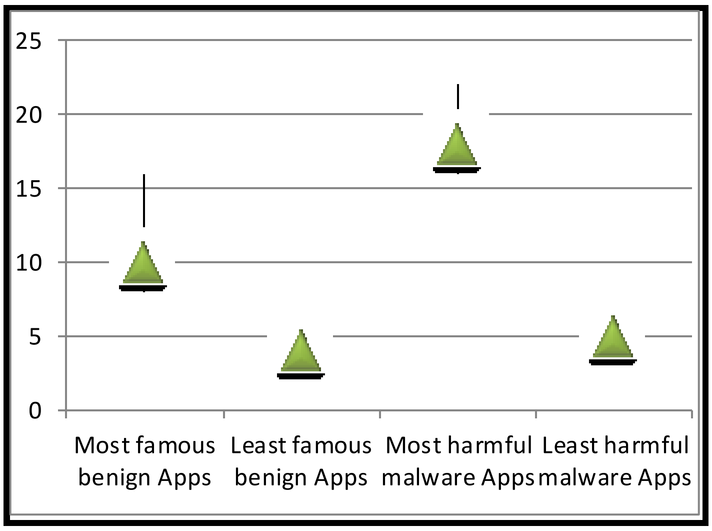


Figure 3.3. Number of permissions used by the benign and malware apps.

3.3.2 Permission Groups

There are 35 permissions out of a total of 145 which are frequently used by apps, whereas remaining 110 are hardly ever used. We can group the repeatedly used permissions into normal and dangerous categories depending on their usage and associated risk levels. Examples of frequently used permissions by benign apps are Full Network access, Create/Add/remove/user accounts, Delete/Modify USB contents, Read/write/modify contacts. Malware apps prefer using Read phone status & ID, Access Network state, Send SMS/MMS, Receive boot complete, Receive SMS, Delete/Modify USB contents, your locations permissions etc.

There are a few permissions, which are scarcely used by benign apps but frequently by malware apps e.g., Access Network state, Receive boot complete, Restart packages, Mount/Unmount File system, Set wallpapers, Read/write history bookmarks of browser, Write APN settings.

Set wallpapers permission is frequently used by adware to display coupons and ads for malicious or marketing websites whether user want them to or not. These ads promote the installation of additional unwanted contents such as browser extensions or optimization utilities and to generate pay-per-click revenue for the originator.

3.3.3 Intent usage by the Applications

Intent is a message passing system which is used to link components of same or different applications [103]. Applications with the same user ID could invoke functionalities of each other without declaring permissions individually for those functionalities thus gaining extra privileges. We categorize malware apps into most harmful and least harmful apps depending on the ease of access to sensitive resources and data regarding used permissions and intents. The most harmful malicious apps are those who are accessing more sensitive resources and data and may provide monetary damages to the users like sending premium rate SMSs, making calls, and accessing bank accounts details. The least dangerous malicious apps are those who can access some useful data and resources, but they may not cause financial or serious damage to the user or device.

Most famous benign apps tend to use on average 1-3 intents, the least famous apps use 1-2. Most harmful malware use min 3 intents while this number goes up to 7. Least harmful malware apps witnessed to use at least 2 or 3 intents as depicted in Figure 3.5. Benign apps are seen to use only ACTION_MAIN, CATEGORY_LAUNCHER and

CATEGORY_DEFAULT intents whereas malware apps are tending toward adding more intent to gain extra capabilities. Most common intents used by the malware apps are ACTION_BOOT_COMPLETED, ACTION_CALL, ACTION_BATTERY_LOW, ACTION_SMS_RECIEVE and ACTION_NEW_OUTGOING_CALL.

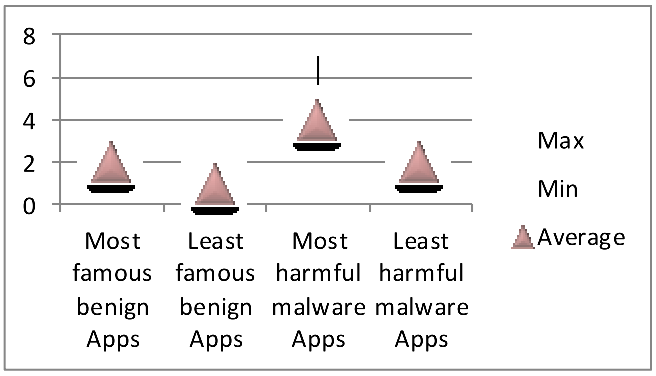


Figure 3.5 Number of intents used by the benign and malware apps.

3.3.4 Combining Permissions and Intents for Malware detection

Android apps are exhibiting a consistent usage pattern of permissions and intents. Figure 3.6 gives an overall trend of how android apps are using these attributes in a clearly distinguishable manner. Real malware apps are corroborated to use few of the normal permissions and intents whilst they use a greater number of dangerous permissions and intents. Benign apps have shown a similar trend of using only normal permissions and intents, whereas the grey ware are those benign apps which are using unnecessary permissions along with the normal permissions and intents, to expand their modus operandi.

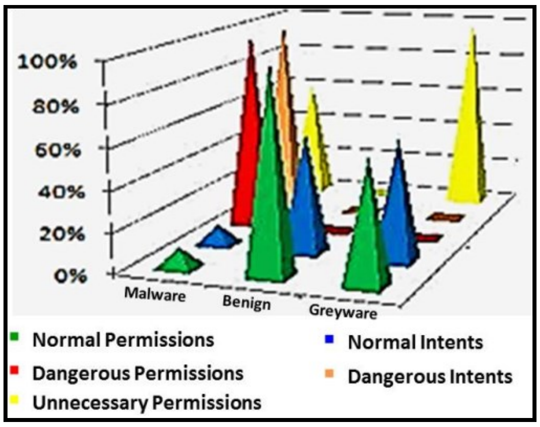


Figure 3.6 Permissions and intents usage pattern by Android apps.

3.4 Correlation between permissions and intents

A Correlation is a statistical technique that is used to measure and describe the strength and direction of the relationship between two variables. Different correlation coefficient methods: Pearson correlation coefficient, Intra-class correlation and Rank correlation. Correlation is defined as a single number known as correlation coefficient that quantifies a type of correlation and dependence, meaning statistical relationships between two or more values in fundamental statistics. We used Pearson correlation coefficient to find the statistical correlation between permissions and intents since it is widely used and more reliable method for the purpose.

Pearson product-moment correlation coefficient, also known as r, R, or Pearson's r, a measure of the strength and direction of the linear relationship between two variables that is defined as the (sample) covariance of the variables divided by the product of their (sample) standard deviations. The most common is the Pearson correlation coefficient that is a statistical measure of the strength of a linear relationship between two variables. It is denoted by “r" and is calculated by dividing the covariance of two variables with product of their standard deviations. Pearson's correlation coefficient has a value between -1 (perfect negative correlation) and 1 (perfect positive correlation) [104].

Suppose we have *n* malware applications, each application is using *X* dangerous permissions written as $x_i = \{x_1, x_2, \ldots, x_n\}$ and *Y* dangerous intents such that $y_i = \{y_1, y_2, \ldots, y_n\}$, then the Pearson correlation coefficient (r) can be calculated using equation (3.1).

$$r_{\mathrm{xy}} = \frac{n\sum x_{\mathrm{i}}y_{\mathrm{i}} - \sum x_{\mathrm{i}}\sum y_{\mathrm{i}}}{\sqrt{n\sum x_{\mathrm{i}}^2 - (\sum x_{\mathrm{i}})^2}\,\sqrt{n\sum y_{\mathrm{i}}^2 - (\sum y_{\mathrm{i}})^2}}. \qquad (3.1)$$

Two sets of malware apps are used to measure the strength of correlation between dangerous permissions and dangerous intents. One set consists of 200 malware apps which are randomly selected from different malware families and the other consists of 20 malware apps from same malware family.

For the first set, the correlation coefficient (r) equals 0.74, indicating a strong relationship between dangerous permissions and dangerous intents for the significance level: $p < 0.001$. For the other set, the correlation coefficient (r) equals to 0.94, indicating a very strong correlation between dangerous permissions and intents in the case of samples belonging to the same malware family. The strong correlation between the dangerous permissions and intents supports our conjecture about the association between permissions and intents to carry out the malicious activity.

The Pearson correlation coefficients of 0.74 for different malware families and 0.94 for same malware family confirm the positive correlation between permissions and intents. However, we need to perform a significance test to decide whether or not there is any evidence which supports or contradicts the presence of a linear correlation in the whole population of malware apps. We use the hypothesis testing, for which we test the null hypothesis, *H0*, and alternate hypothesis, *H1* as

H0 : malware and benign applications use the same set of permissions and intents,

H1 : malware and benign applications don't use the same set of permissions and intents.

For hypothesis testing, we use the Mann-Whitney $U-$test with the p-value of 0.05. We calculate *U1* and *U2* values for both the permissions and intents respectively using equations 3.2 and 3.3, respectively. In following equations, *R1* and *R2* are the sums of ranks for permissions and intents, respectively, and *n1* and *n2* are the sample sizes for both the variables.

$$U_1 = R_1 - \frac{n_1(n_1+1)}{2}; \tag{3.2}$$

$$U_2 = R_2 - \frac{n_2(n_2+1)}{2}. \tag{3.3}$$

We take the smallest of U and compare it with the critical value obtained from the Mann-Whitney critical values table [105]. We use Mann-Whitney critical values table for a small number of malware samples and Z-test for large samples of malware apps due to limitations of the number of entries in the Mann-Whitney critical value table. With samples from same malware family (n1= 20, n2 = 19, p=0.05, critical value = 119), the smallest U value obtained is 87 which is less than the critical value of 119, we would reject the null hypothesis for the malware apps belonging to same family. For a large sample of apps belonging to different malware families (n1 = n2 = 200, p=0.05, Z-critical value = 1.64), we calculate z-score with Z test. We obtain z-score of 13.0594 which is greater than Z-critical value hence suggesting the rejection of null hypothesis H0. We have very strong statistical evidence to accept the alternate hypothesis H1, which suggests that the malware and benign apps use a different set of permissions and intents. This conjecture is further verified with classification analysis using different machine learning algorithms.

3.5 AndroPIn: Malware Detection Algorithm

In this section, we present an automated malware detection algorithm: AndroPIn which is based on identification of distinct usage pattern of permissions and intents declared in the manifest file. Once declared, these vital features cannot be altered by code obfuscation or encryption. AndroPIn is an implementation of the methodology proposed in Chapter 4 and validating the effectiveness of algorithm for detection of malware apps including the obfuscated ones. It extracts the permissions and intents of an app and classifies it as malicious or benign by comparing against certain combinations of permissions and intents. These combinations form a distinct usage pattern of malicious apps which distinguishes them from the benign apps. These features, which play a major role in sharing user data and device resources cannot be obfuscated or altered. These vital features are well suited for resource constrained smartphones. Experimental evaluation on a corpus of real-world malware and benign apps demonstrate that the proposed algorithm can effectively detect malicious apps with low run-time overheads and is resilient to common obfuscations methods.

3.5.1 Design

A malware detection system for Android can be architected in a variety of ways. It could be designed as a complete client based anti-malware scanner app or a client and server based solution to efficiently process the analysis and classification of malware apps. A complete client based solution would have to overcome a number of challenges for efficiently and accurately detecting the malware apps. We develop our solution using client and server architecture which is efficient and accurate since it does not rely on the limited resources of mobile phones. The client-server architecture further has multiple choices to select for the implementation of different tasks either on server or on client. Such sub tasks include extraction of features, comparison of learned behaviour against the normal or malicious etc. We studied different design options and discuss here the selected one in which we use server for the analysis and detection processes to gain efficiency. However, it is possible to design the whole architecture on the phone at a little bit cost of efficiency.

The work flow of AndroPIn is shown in Fig. 3.7. It consists of three main stages: Feature extraction, Detection engine and data logger / reporter. First of all the apk file is decompiled and the required features: permissions and intents of an app are extracted and stored in a separate file for analysis.

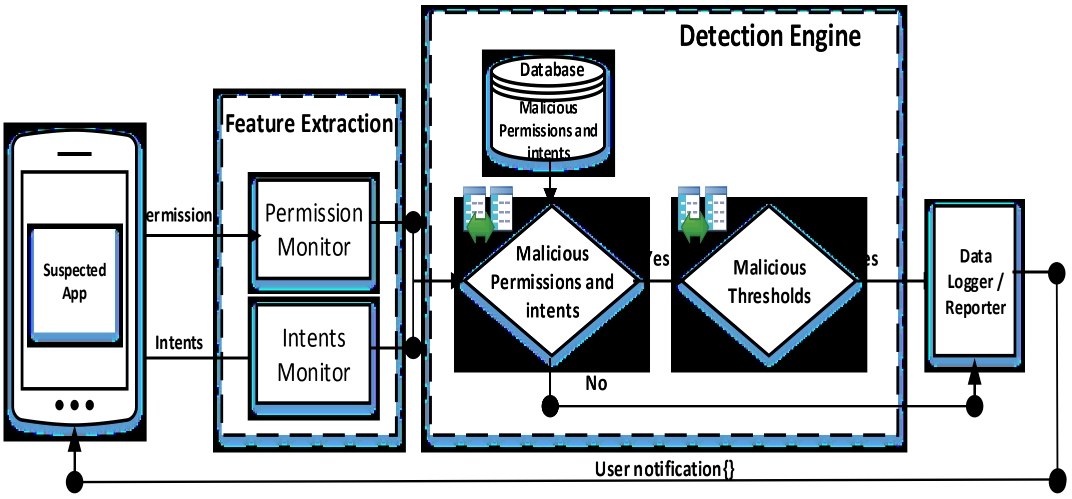


Fig. 3.7: AndroPIn architecture

The analysis/detection engine consists of two steps: In first step, the extracted features of a suspected app are checked against the pre-defined template T, which consists of four arrays of malicious permissions, normal permissions, malicious intents and normal intents. Second step involves the testing against the malicious threshold. If the app is within the malicious threshold, it is labelled as malware. Third stage is the data logger and reporter which makes logs of results and generates notifications for the user.

3.5.2 Implementation

We present an algorithm for checking whether an app is malware or benign. We have implemented our algorithm as a malware detection tool. The proposed algorithm consists of two phases: identifying the malicious permissions and intents (Algorithm 1) and classifying the app as either malware or benign (Algorithm 2).

The crux of AndroPIn is the component responsible for classifying an application's behaviour as either benign or malicious. We use Androguard[1] tool to extract permissions and intents of an app from its AndroidManifest.xml file. Androguard is a python based reverse engineering tool, which can run on Linux/Windows/OSX. It is used to disassemble and to decompile android apps and to statically analyse apps. The Androguard’s commands get_permissions() and get_intents() lists the permissions and intents declared by an app. After extracting the permissions, the program automatically saves the information in a temporary output file: ‘output3.txt’. We use Python to develop the algorithm, which first defines the malicious and normal feature set in the algo.py file (Algorithm 1). There are four arrays, each defining the malicious permissions, normal permissions, malicious intents and normal intents. Permissions and intents of suspected app saved in ‘output3.txt’ are compared against the four arrays. If the under test app contains the malicious permissions and malicious intents, these are printed on the screen as shown in Fig.3.8.

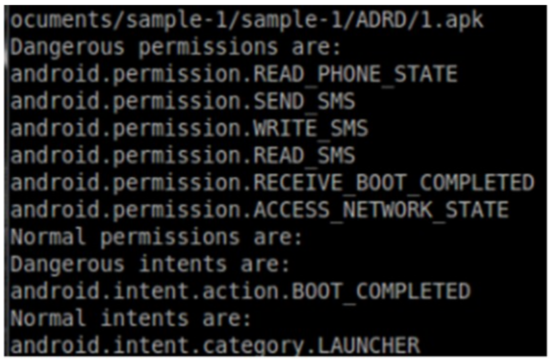
[1] https://github.com/androguard/androguard

Algorithm 1: Identification of Malicious permissions and intents

Input : List of malicious permissions,
List of normal permissions,
List of malicious intents,
List of normal intents

Output : List of malicious and normal permissions & intents of suspected app

```
1:  apk                     > an incoming app
2:  Malicious_Permission [n1]: = List of n1 malicious permissions
3:  Normal_Permission [n2 ]: = List of n2 normal permissions
4:  Malicious_Intent  [n3 ]: = List of n3 malicious intents
5:  Normal_Intent  [ n4]: = List of n4 normal intents
6:  Create an object O of the APK class for Suspected_Malware.apk
7:  Call O. get_android_manifest_xml() to generate AndroidManifest.xml
8:      for i: = 1 to n1
9:          if (Malicious_Permission [i] exist in AndroidManifest.xml)
10:             Print Malicious_Permission [i]
11:         end if
12:     end for
13:     for i: = 1 to n2
14:         if (Normal_Permission [i] exist in AndroidManifest.xml)
15:             Print Normal_Permission [i]
16:         end if
17:     end for
18:     for i: = 1 to n3
19:         if (Malicious_Intent[i]  exist in AndroidManifest.xml)
20:             Print Malicious_Intent  [i]
21:         end if
22:     end for
23:     for i: = 1 to n4
24:         if (Normal_Intents [i] exist in AndroidManifest.xml)
25:             Print Normal_Intent  [i]
26:         end if
27:     end for
```



```
ocuments/sample-1/sample-1/ADRD/1.apk
Dangerous permissions are:
android.permission.READ_PHONE_STATE
android.permission.SEND_SMS
android.permission.WRITE_SMS
android.permission.READ_SMS
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.ACCESS_NETWORK_STATE
Normal permissions are:
Dangerous intents are:
android.intent.action.BOOT_COMPLETED
Normal intents are:
android.intent.category.LAUNCHER
```

Fig. 3.8: Matched permissions and intents

The malware detection Algorithm 2 is responsible for classifying an application's behaviour as either benign or malicious. If the Algorithm 1 confirms the presence of malicious permissions and malicious intents in the under test app, Algorithm 2 verifies against the thresholds of maliciousness to avoid the false positives. If the app falls into the malicious criteria then it is labelled as malware and user is notified.

Algorithm 2 Malware Detection Process

LEGEND:
- **MP:** Malicious_ Permissions
- **NP:** Normal_Permissions
- **MI :** Malicious_Intents
- **NI :** Normal_ Intents

Input: Malicious_ Permissions[i]
Normal_Permissions[i]
Malicious_Intents[i]
Normal_ Intents[i]

Output: Malware notification

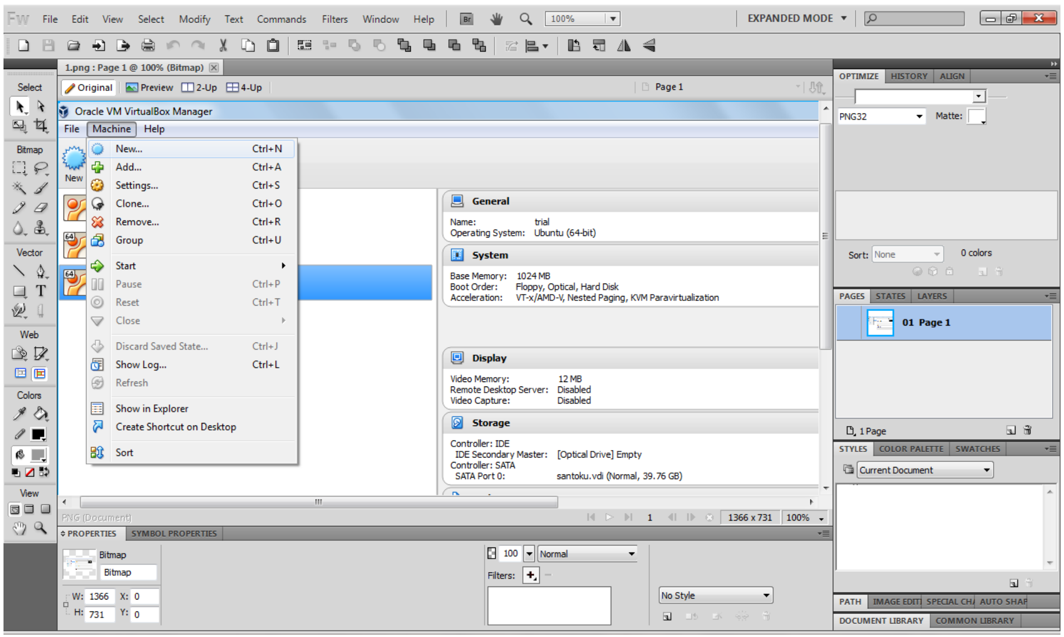
```
1:  Scan for (Malicious_ Permissions[i], Normal_Permissions[i],
                  Malicious_Intents[i], Normal_ Intents[i])
2:  if (MP >= 1 && MI >= 1) //Filter malicious permissions and malicious intents
3:     then
4:           if (MP + + NP = = 3) && (MI + + NI >=1)
5:                 Print Malware detected
6:     else
7:                 Print Goodware detected
8:     end if
```

3.6 EXPERIMENTAL SETUP AND RESULTS

We evaluate our implementation of algorithm against real world malware and benign apps. The experiments aimed to validate the effectiveness of algorithm for detection of malware apps including the obfuscated ones with a low false positive rate.

3.6.1 Experimental Setup

The experiments are carried out on an Intel Core i7-3520 M CPU @ 2.90 GHz, 2901 MHz machine with 8GB RAM. Machine was configured with different Android reverse engineering tools. Androguard can be run on terminal by directly downloading from the project website or it can be run on Virtual machine environments such as Santoku or Android Reverse Engineering (A.R.E) virtual machines. Both of these VMs are installed with all modules required to run Androguard. We use Santoku VM due to its preference by the Androguard creators. Santoku is a dedicated to mobile forensics, analysis, and security. It is a Linux distribution. We download the full pack of Virtual machine with all modules required to run the tool and installed with default settings. First step is to create a new virtual machine to carry out our analysis as shown in Fig. 3.9. Then the configurations of resources for the VM are done as shown in Figures 3.10 and 3.11 respectively.



General

Name: trial
Operating System: Ubuntu (64-bit)

System

Base Memory: 1024 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, KVM Paravirtualization

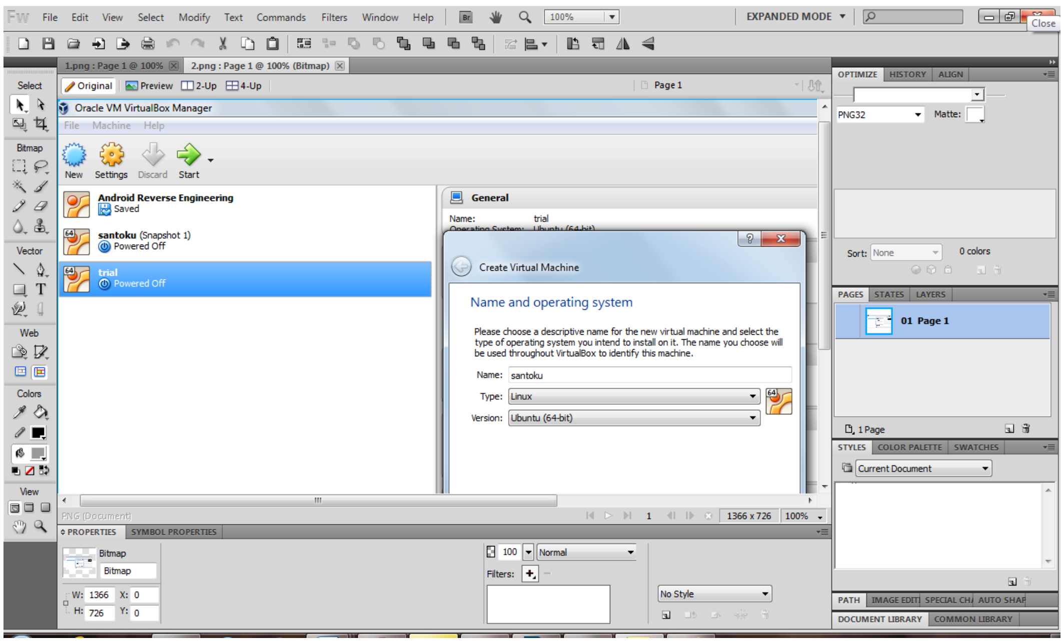
Display

Video Memory: 12 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage

Controller: IDE
IDE Secondary Master: [Optical Drive] Empty
Controller: SATA
SATA Port 0: santoku.vdi (Normal, 39.76 GB)

Fig. 3.9 AndroPIn: creation of new VM



Android Reverse Engineering
Saved

santoku (Snapshot 1)
Powered Off

trial
Powered Off

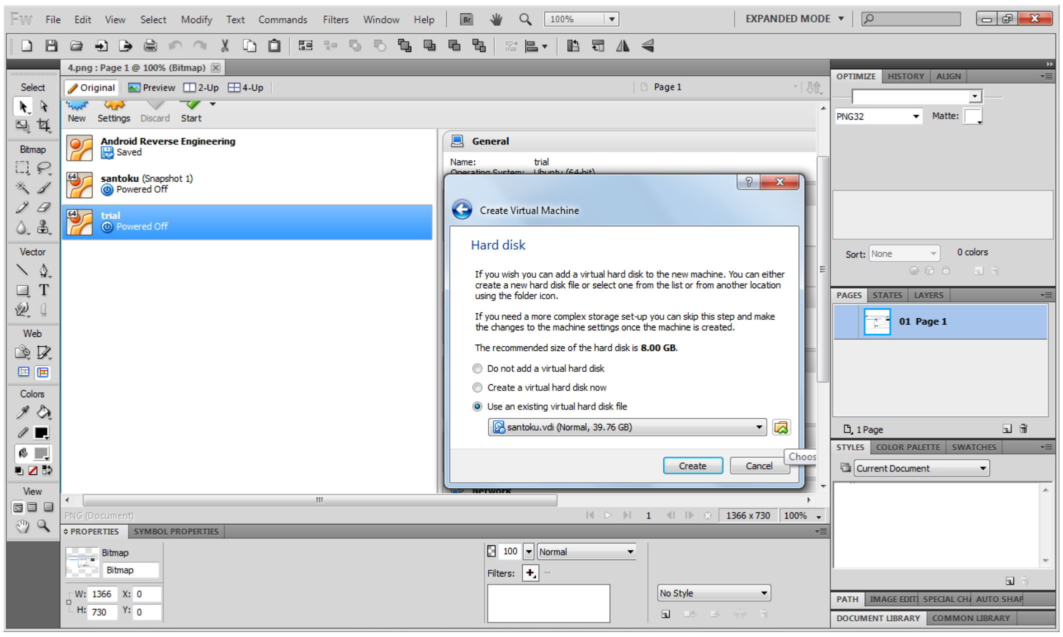
Create Virtual Machine

Name and operating system

Please choose a descriptive name for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: santoku
Type: Linux
Version: Ubuntu (64-bit)

Fig. 3.10 AndroPIn: configuration of new VM



Create Virtual Machine

Hard disk

If you wish you can add a virtual hard disk to the new machine. You can either create a new hard disk file or select one from the list or from another location using the folder icon.

If you need a more complex storage set-up you can skip this step and make the changes to the machine settings once the machine is created.

The recommended size of the hard disk is **8.00 GB**.

Do not add a virtual hard disk
Create a virtual hard disk now
[✓ Use an existing virtual hard disk file]

santoku.vdi (Normal, 39.76 GB)

Create Cancel

Fig. 3.11: AndroPIn: configuration of new VM continued

Once the VM is created, starting the machine will display the main analysis environment as shown in Fig. 3.12.

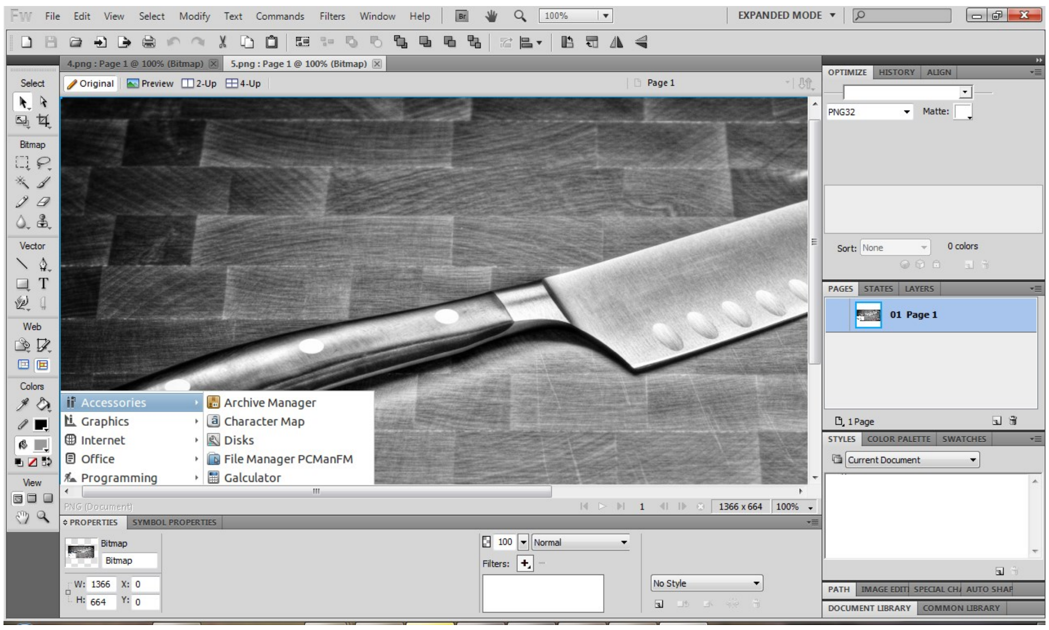
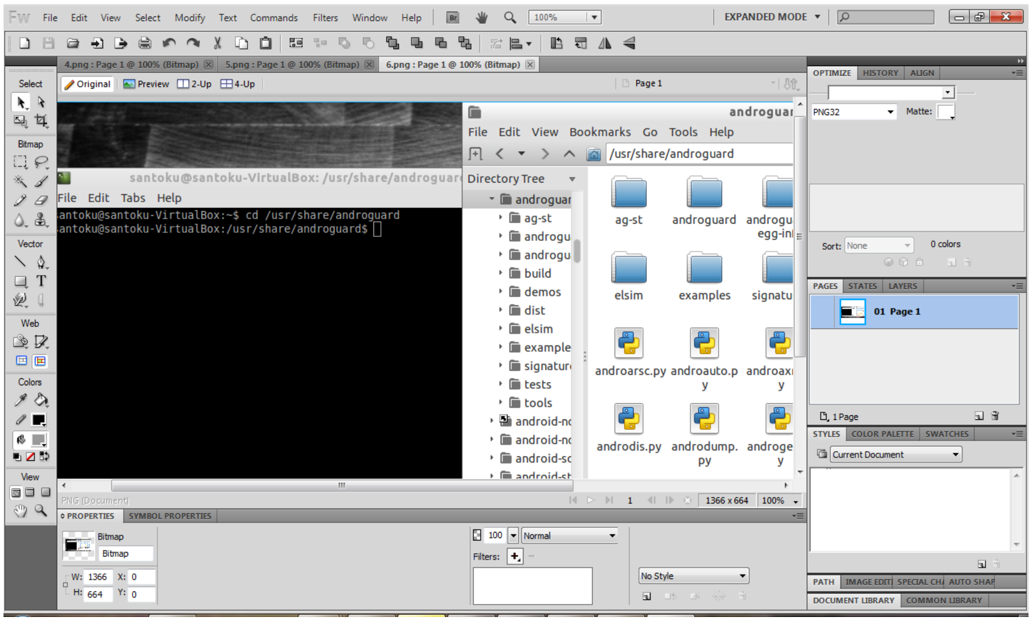


Fig. 3.12: Analysis of app

Clicking on the Knife and going to accessories to open the LXTerminal as shown in Fig. 3.13. With the following command we start the Androguard in our VM:

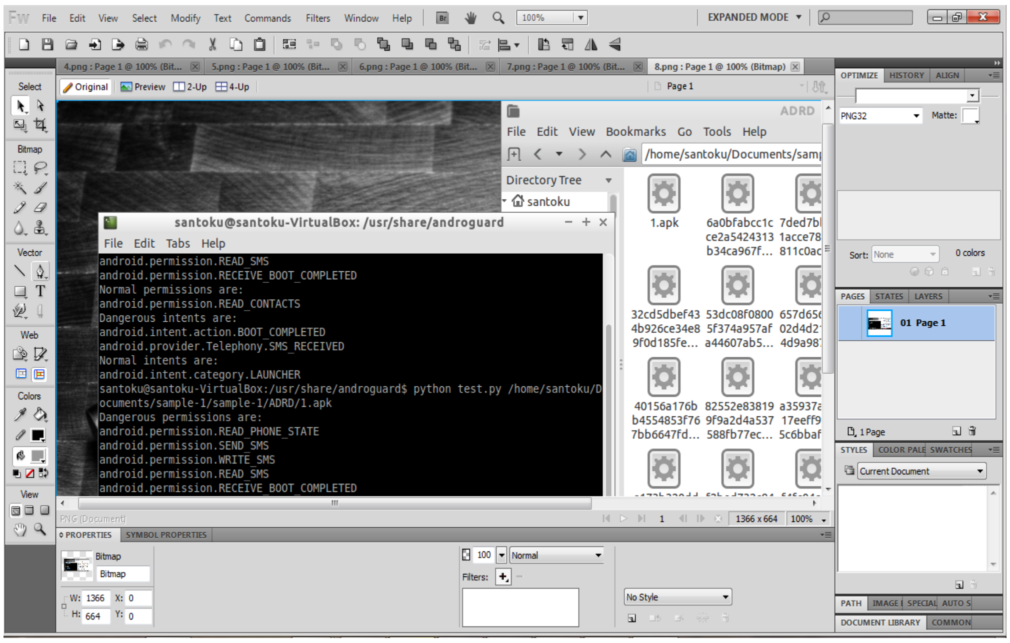
```
cd /usr/share/androguard
```



```
antoku@santoku-VirtualBox:~$ cd /usr/share/androguard
antoku@santoku-VirtualBox:/usr/share/androguard$ 
```

Fig. 3.13: Calling the Androguard

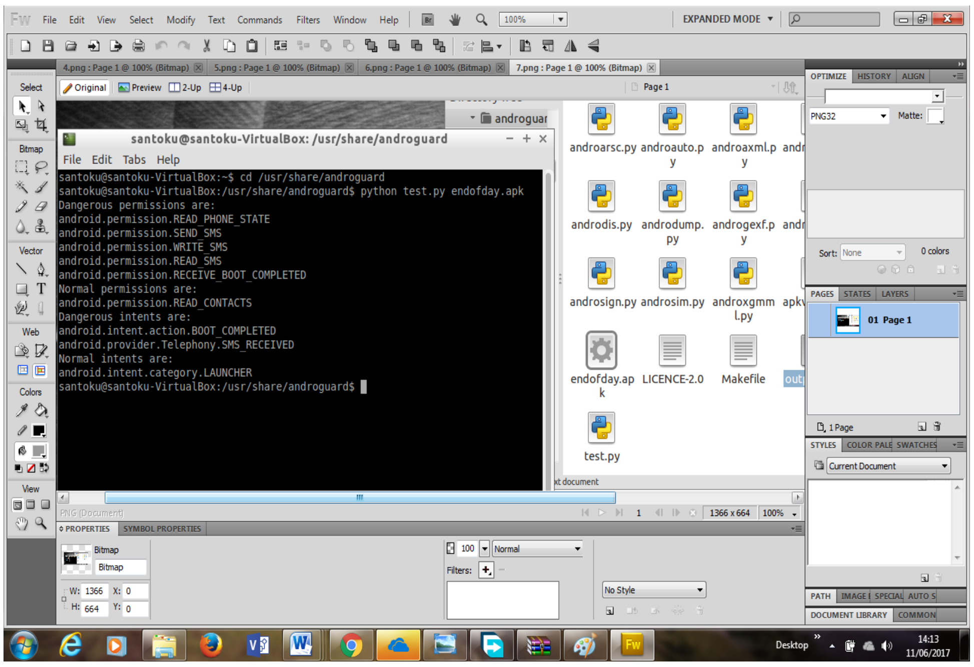
Using the file manager (next to the Knife icon) and starting the /usr/share/androguard for placing the algorithm file: test.py in that directory as shown in Fig. 3.14.



```
android.permission.READ_SMS
android.permission.RECEIVE_BOOT_COMPLETED
Normal permissions are:
android.permission.READ_CONTACTS
Dangerous intents are:
android.intent.action.BOOT_COMPLETED
android.provider.Telephony.SMS_RECEIVED
Normal intents are:
android.intent.category.LAUNCHER
santoku@santoku-VirtualBox:/usr/share/androguard$ python test.py /home/santoku/D
ocuments/sample-1/sample-1/ADRD/1.apk
Dangerous permissions are:
android.permission.READ_PHONE_STATE
android.permission.SEND_SMS
android.permission.WRITE_SMS
android.permission.READ_SMS
android.permission.RECEIVE_BOOT_COMPLETED
```

Fig. 3.14: AndroPIn algorithm in VM

Once the algorithm file is setup in the environment, the apk files are verified as shown Fig. 3.15.



```
santoku@santoku-VirtualBox:~$ cd /usr/share/androguard
santoku@santoku-VirtualBox:/usr/share/androguard$ python test.py endofday.apk
Dangerous permissions are:
android.permission.READ_PHONE_STATE
android.permission.SEND_SMS
android.permission.WRITE_SMS
android.permission.READ_SMS
android.permission.RECEIVE_BOOT_COMPLETED
Normal permissions are:
android.permission.READ_CONTACTS
Dangerous intents are:
android.intent.action.BOOT_COMPLETED
android.provider.Telephony.SMS_RECEIVED
Normal intents are:
android.intent.category.LAUNCHER
santoku@santoku-VirtualBox:/usr/share/androguard$ 
```

Fig. 3.15: Analysis results

3.6.2 Dataset

A total of 145 malware and 125 benign apps are verified with the algorithm which is collected from well-known sources described in Chapter 3. These samples are rigorously selected from known malware families and different categories of benign apps. Details of samples from the malware families and benign categories are shown in Tables 3.2 and 3.3 respectively.

Table 3.2: Details of Malware samples with obfuscation

Malware Family	No of samples	Malware Family	No of samples
Basebridge	5	DroidKungFu	5
FakeDolphin	5	Locker	5
VDLoader	5	FakeBank	5
GinMaster	5	Boxer	5
JIFake	5	SNDApps	5
OpFake	5	FakeInst	5
FakePlayer	5	BgServ	5
Plankton	5	Geinimi	5
AnserverBot	5	PjApps	5
GoldDream	5	DroidSheep	3
CopyCat	3	DroidDream	5
DroidKungFu	5	Keji	3
HolyBible	3	Obad	2
Nickispbby	2	RuFraud	3
Jsmshider	3	Zitmo	3
AngryBird	5	KMin	5
DroidKungFua	3	DroidKungFuaa	2

Table 3.3: Categories of benign apps

Category	No of samples	Category	No of samples
Social Media	5	Mail	5
Education	5	Banking	5
Entertainment	5	Sports	5
Shopping	5	Finance	5
News	5	Weather	5
Games	5	Medical	5
Fitness	5	Media	5
Casual	5	Music	5
Books	5	Travel	5
Lifestyle	5	Simulation	5
Transportation	3	Misc	20

3.6.3 Results and Performance Analysis

The detection results are shown in Table 3.4. The TPR of 0.98 and FPR of 0.02 are achieved with the experimental dataset. These results can further be improved with optimization of algorithm.

Table 3.4: Performance Results

Method	TPR	FPR
AndroPIn	0.98	0.02

3.7 Summary

Android security framework relies on permissions and intents to control the access to vital hardware and software resources. These two features have never been used in tandem for malware detection. In this paper, we proposed Andropin, a novel and efficient malware detection algorithm based on these two vital security features, which was evaluated on real world malicious and benign apps. Malware samples selected for the experiments represent different types of malicious families with diversified real-world threats. The experiment results demonstrate that the proposed algorithm can accurately detect malware apps. We also evince with experiments that the proposed algorithm is particularly effective for detection of obfuscated malware apps due to its reliance on the unalterable features. Our future work aims to optimize the algorithm to improve on the false positive rate and validation on larger malware dataset.

Chapter 4

PInDroid: Permissions and Intents based Malware detection

4.1 Introduction

In this chapter, we discuss our methodology of PInDroid, which is built on the study presented in Chapter 3. Since malware use more sophisticated obfuscation and evasion techniques, it is provident to use the obfuscation resilient methods and features for malware detection. For this reason, PInDroid uses those features of manifest file, which are resilient to code obfuscation and there is no complexity involved in extraction of these features. The selected features are the basic essence of any Android app; without these features apps cannot do any good or bad or things.

Permissions are a vital security mechanism which guards against the misuse of hardware and software resources [106]; however, it relies on the user’s decision to accept the permissions of an app and other built-in features mainly the intents deprecate such security protections. Most malware need to use some permission to achieve their malicious goals which they must declare in the Manifest file and ask the users for approving the permissions before installation of app. Similarly, malware apps use intents to carry out malicious actions which they must declare in the Manifest file but do not ask the users for approval. This design flaw on Android is exploited by malware apps to carry out stealthy malicious actions. Intents have extended a number of known and unknown legitimate covert channels to malware app. Although, a significant research is done on permissions and API calls for detection of malware, however, intents remained almost untouched before starting this research work. There was no published works where intent was used as a key feature for detection of malware at the start of PhD research in 2013.

Our research on permissions and intents led us to a feature set that helps in accurately detecting malicious apps. After separately investigating the potentials of permissions and intents in detecting the maliciousness of apps, the author combined these two features to study their effectiveness in pinpointing the possible risky behaviours of apps. We argue that since many of the stealthy malicious actions are not possible without combining the permissions and intents by the malware developers, thus many of the malware apps cannot be detected without combining these features. Our experiments strongly validate our heuristics.

The rest of the chapter is organized as follows: Section 4.2 presents the overview of the methodology which includes the system architecture, samples dataset, reverse engineering techniques used to analyse the apps, extraction of features and pre-processing, building the classification models using different ML algorithms, comparison of performance of these algorithms on the dataset and feature set. Section 4.3 discusses the experimental setup and Section 4.4 presents the results. Section 4.5 compares the approach with the related state-of-the-art approaches. Section 4.6 discusses the application of ensemble learning methods and Section 4.7 summarizes the chapter.

4.2 Overview

The aim of this work is to validate a set of simple and effective features which should be easily extracted, applied and combined to classify the malware apps. Different machine learning algorithms and ML based malware detection approaches [64, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148 and 149] are investigated for classification of malicious apps. The work flow of this methodology is divided into two phases: training and testing. In the training phase, a set of features (Permissions and Intents) are extracted from the manifest files of a large sample of malware and benign apps. The extracted features are represented in a vector format executable by the data mining software Weka[1]. Our goal is to build a model which can distinguish malware from

1Weka, http://www.cs.waikato.ac.nz/ml/weka/

benign applications efficiently based on Android permissions and intents.

Six machine learning classifiers: Naive Bayesian, Decision Tree, Decision Table, Random Forest, Sequential Minimal Optimization and Multi-lateral Perceptron (MLP) are trained with the datasets to build classification models.

In the testing phase, the same set of features is extracted from a sample of benign and malware to be tested and classified by the learned models from the training phase. Performance of ML algorithms is validated against various performance measures.

4.2.1 System Architecture

The system architecture is shown in the Figure 4.1. It consists of four main stages: Apk de-compilation, Feature extraction, Pre-processing, and Classification. The first stage is for decompiling of target app to get AndroidManiFest.xml. The second stage analyses the manifest file and extracts the permissions and intents. This stage comprises of two monitors that are used to measure: (i) type of permissions (normal or dangerous) and their numbers and, (ii) type of intents (normal or dangerous) and their number. Permissions and intents are labelled into four groups: normal permissions, normal intents, dangerous permissions and dangerous intents. Dangerous permissions and intents are frequently used by malware apps whilst normal permissions and intents are frequently used by benign apps. The pre-processor stage transforms the extracted features from each app into vector dataset in an ARFF file format that can be applicable for machine learning algorithms. Each app is represented as a single instance with discrete vector of features and a class label indicating whether the app is benign or malicious. The generated dataset is randomized using unsupervised instance randomization filter for better accuracy and sent to the classifier stage. The last stage is for the classification of app as either malware or benign. The classifier is trained with the known samples and the learned models are used to detect whether a given app is malicious or benign. The classifier takes each vector as input and classifies the data set using trained classifier. Finally, the reporter stage generates notifications for the user based on the classifier results.

Details of implementation, datasets, used tools, main features of interest, and the ML algorithms are given in subsequent paragraphs.

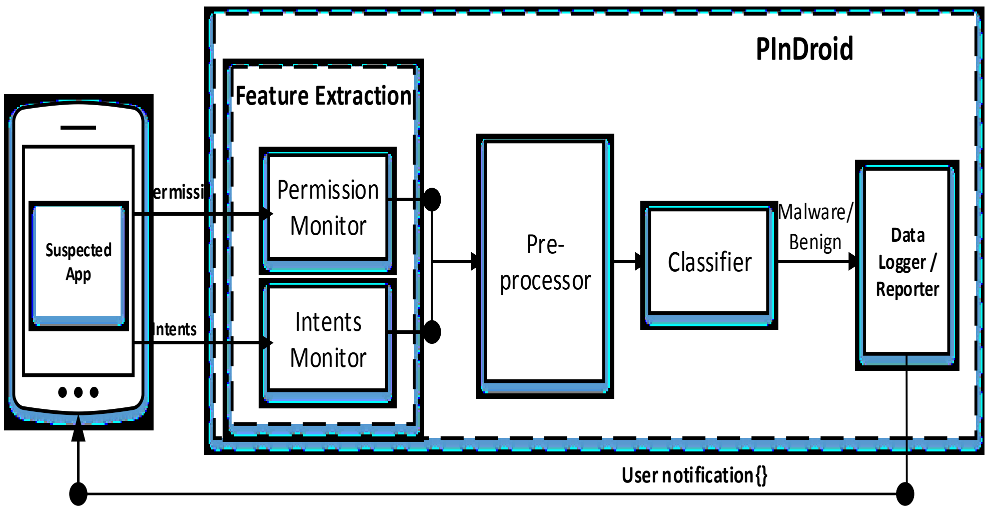


Figure 4.1: Diagram of proposed system

4.2.2 Data Collection

A total of 1300 malware and 445 benign apps were analysed, which are collected from well-known sources such as Google Play store[1], Contagiodump[2], Genome[3], Virus Total[4], theZoo[5], MalShare[6], and VirusShare[7]. Samples were selected to ensure that the dataset represent the behaviour of broad categories of benign apps and families of malware apps. Table 4.1 depicts the details of malware samples collected from each source. These sources contain the datasets of already known malware samples. Maliciousness of these samples is also confirmed with Virus Total service integrated with ten detection engines. We labelled the app as malware, if it was detected as malicious by two of the engines. Cryptographic hashes (SHA-1) of files were also checked with a tool: HashTab[8] to ascertain the uniqueness of samples. Details of known malware families, their malicious activities and number of analysed samples from each family are shown in Table 4.2.

To validate our method, we also downloaded 445 benign apps from known app stores such as Google Play, AppBrain[9], F-Droid[10], Getjar[11], Aptoid[12], and Mobango[13]. The benign apps are selected from different categories such as social, news, entertainment, finance, education, games, sports, music, and audio, telephony, messaging, shopping, banking and weather to learn the normal behaviour of benign apps. Table

1Google Play, Web: https://play.google.com/store?hl=en
2Contagio Mobile: mobile malware mini dump, Web:http://contagiominidump.blogspot.co.uk/
3Android Malware Genome Project, Web: http://www.malgenomeproject.org/
4VirusTotal for Android,Web: https://www.virustotal.com/en/documentation/mobile-applications/
5theZoo aka Malware DB, Web: http://ytisf.github.io/theZoo/
6MalShare project, Web: http://malshare.com/about.php
7Web: https://virusshare.com/
8HashTab, Web: http://implbits.com/products/hashtab/
9Web: http://www.appbrain.com/
10Web: https://f-droid.org/
11Web: http://www.getjar.com/
12Web: https://www.aptoide.com/
13Web: http://www.mobango.com/

4.3 depict the details of categories of benign apps, number of analysed apps from each category and the corresponding app stores.

Table 4.1: List of Malware samples

Malware family	No of samples	Malware type
Basebridge	11	Botnet, Information stealing
DroidKungFu	11	Botnet, Information stealing
DroidKungFu	10	Botnet, Information stealing, Backdoor
FakeDolphin	4	Adware
Locker	2	Ransomware
VDLoader	3	Backdoor, Information stealing
FakeBank	5	Trojan Banker, Money stealing, Information stealing
GinMaster	7	Information stealing, Backdoor
Boxer	2	Sends SMS
JIFake	3	Sends SMS
SNDApps	1	Information stealing
OpFake	4	Sends SMS
FakeInst	3	Installer
FakePlayer	3	Sends SMS
BgServ	7	Botnet, Information stealing, Trojan Installer, backdoor
Plankton	7	Money stealing, Botnet, Information

		stealing, Backdoor, Trojan installer
Geinimi	9	Botnet, Information stealing, Root access
AnserverBot	13	Information stealing
PjApps	9	Botnet, backdoor
GoldDream	10	Trojan, Information stealing
DroidSheep	7	Session hijacker
CopyCat	4	Adware
DroidDream	10	Information stealing, Adware
DroidKungFu	11	Botnet, Information stealing, Root access
Keji	4	Information stealing, Trojan Installer
HolyBible	5	Adware, Backdoor
Obad	2	Botnet, Information stealing, Botnet, Trojan Installer, backdoor, SMS, Location
Nickispbby	5	Spying, Information stealing
RuFraud	3	SMS sending

Jsmshider	3	Information stealing
Zitmo	3	Money, Information stealing, Backdoor
AngryBird	13	Botnet, Information stealing
KMin	10	Exploit, Information stealing

Table 4.2: Sources of malware samples

Source	No of malware samples
Contagio	60
Drebin	100
Genome	1000
Virus Total	70
theZoo	20
MalShare	25
VirusShare	25

Table 4.3: Categories and sources of benign samples

Category	No of samples	App Market
Social Media	11	Google Play store
Mail	4	Google Play store
Education	10	Google Play store
Banking	4	Google Play store
Entertainment	15	Google Play store
Sports	8	Google Play store
News	8	Google Play store
Weather	8	Google Play store
Games	15	Google Play store

Weather	8	Google Play store
Games	15	Google Play store
Medical	10	Google Play store
Fitness	11	Google Play store
Media	11	Google Play store
Casual	15	Google Play store
Music	15	Google Play store
Books	5	Google Play store
Travel	5	Google Play store
Lifestyle	15	Google Play store
Simulations	7	Google Play store
Misc	15	AppBrain
Misc	10	F-Droid
Misc	10	Getjar
Misc	15	Aptoid
Misc	15	Mobango

4.2.3 Feature Extraction

The collected samples are apk files that were analysed and transformed into the format suitable for the Machine learning algorithms. Each apk file is decompressed to extract the manifest file, which is investigated to for the desired features: permissions and intents. The extracted features are then processed to build a dataset in an ARFF file format. Each instance of dataset represents either a malware or benign app. Feature datasets and examples of feature vector set are shown in Tables 4.4 and 4.5 respectively.

Table 4,4: Selected features

Features	Category	Sub Features
Permissions	Normal Permissions	WRITE_SETTINGS CREATE ACCOUNTS ADD ACCOUNTS REMOVE ACCOUNTS USE ACCOUNTS
	Dangerous Permissions	INTERNET READ_PHONE_STATE SEND_SMS INSTALL PACKAGES RECEIVE _SMS WRITE_SMS READ_SMS RECEIVE_BOOT_COMPLETED MOUNT_UNMOUNT_FILESYSTEM
Intents	Dangerous Intent	BOOT_COMPLETED SMS_RECEIVED PHONE_STATE NEW_OUTGOING_CALLS UNINSTALL_SHORTCUT HOME
	Normal Intent	MAIN LAUNCH VIEW BROWSABLE

Table 4.5: Examples of features Vector set

Normal Permission	Normal Intent	Dangerous Permission	Dangerous Intent	Classification
1	1	3	2	Malware
10	2	2	0	Benign
9	3	0	1	Benign
4	2	0	0	Benign
3	2	7	3	Malware
5	2	2	0	Benign
4	0	1	1	Benign
6	2	11	4	Malware
2	1	0	0	Benign
3	0	2	1	Benign

4.3 Experimental Settings

The experiments were carried out on an Intel Core i7-3520 M CPU @ 2.90 GHz, 2901 MHz machine with 8GB RAM. Machine was configured with different machine learning algorithms (WEKA software), Android development and testing modules, apk file parser as well as some open source analysis tools. Each of the classifiers is evaluated with the 10-fold cross-validation method. In 10-fold cross-validation, the data is divided into ten subsets, and the method is repeated ten times. In each round, one subset is taken as test set and the remaining nine subsets are combined to form the training set. Errors of all the ten rounds are averaged out to obtain a final output. This method ensures that each instance is included at least once in the test set and nine times in the training set. The final model is the average of all ten iterations. Basically, we applied the classifier to data 10 times and every time with 90:10 ratios (90% for training and 10% for testing). The final model is the average of all 10 iterations as depicted in Figure 4.2.

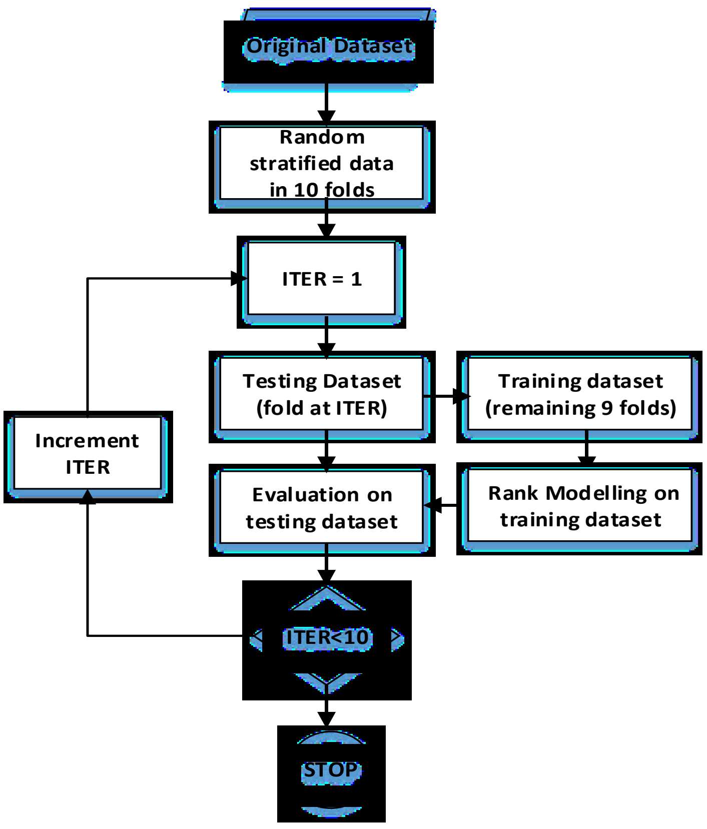

Original Dataset
Random stratified data in 10 folds
ITER = 1
Testing Dataset (fold at ITER)
Training dataset (remaining 9 folds)
Increment ITER
Evaluation on testing dataset
Rank Modelling on training dataset
ITER<10
STOP

Figure 4.2: Flowchart for 10-fold experiments

4.3.1 ML Classifiers

Performance of the following six ML classifiers is compared against different measures.

(i) Naive Bayesian

(ii) Decision Tree (J48)

(iii) SMO

(iv) Random Forest

(v) Neural Networks Multi-Layer Perceptron (MLP)

(vi) Decision Table (DT)

(i) Naïve Bayesian

It is a conditional probabilistic classifier based on Bayes' theorem with an assumption of independence between the features to predict the class. A Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. Naive Bayesian model is easy to build and particularly useful for very large data sets. It provides a way of calculating posterior probability P (c|x) from P (c), P (x) and P (x|c) [1].

$$\text{Posterior (P (c|x)} = \frac{\text{Prior P(c) x Likelihood P(x)}}{\text{Evidence P (x|c)}} \qquad (3.4)$$

(ii) Decision Tree

Decision tree uses a decision tree predictive model to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). In classification trees (the target variable can take a discrete set of

1 https://en.wikipedia.org/wiki/Naive_Bayes_classifier

values) leaves represent class labels and branches represent conjunctions of features that lead to those class labels[2].

(iii) Sequential minimal optimization (SMO)

SMO is an algorithm for solving the quadratic programming (QP) problem that arises during the training of support vector machines. SMO is widely used for training support vector machines and is implemented by the popular LIBSVM tool that is simpler than the previously available methods for SVM training and it required expensive third-party QP solvers[3].

(iv) Random Forest

Random forests is an ensemble learning method for classification that constructs multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) of the individual trees. Random decision forests correct for decision trees' habit of over fitting to their training set [4].

(v) MLP

MLP is an artificial neural network algorithm consisting of at least three layers of nodes. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. It is s multiple

2https://en.wikipedia.org/wiki/Decision _Tree
3 https://en.wikipedia.org/wiki/Sequential_Minimal_Optimization
4 https://en.wikipedia.org/wiki/Random_Forest

layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable[5].

(vi Decision Table

Decision tables are a visual representation for specifying which actions to perform depending on given conditions. They are algorithms whose output is a set of actions. The information expressed in decision tables could also be represented as decision trees or in a programming language as a series of if-then-else and switch-case statements. Each decision corresponds to a variable, relation or predicate whose possible values are listed among the condition alternatives. Each action is a procedure or operation to perform, and the entries specify whether (or in what order) the action is to be performed for the set of condition alternatives the entry corresponds to[6].

4.3.2 Performance Comparison of ML Classifiers

Performance of the six classifiers is compared in terms of True Positive Rate (TPR), False Positive Rate (FPR), accuracy, F1-score and Area Under Curve (AUC). These metrics are calculated using the confusion matrix as shown in Table 4.6.

5 https://en.wikipedia.org/wiki/Multilayer_Perceptron
6https://en.wikipedia.org/wiki/Decision_table

Table 4.6 is generated from the four measures: True Positive (TP) - the number of correctly classified instances that belong to the class, True Negative (TN) - the number of correctly classified class instances that do not belong to the class, False Positive (FP) - instances which were incorrectly classified as belonging to the class and False Negative (FN) - instances which were not classified as class instances.

$$TPR = \frac{TP}{TP + FN}; \tag{3.5}$$

$$FPR = \frac{TP + TN}{TP + FN + FP + TN}; \tag{3.6}$$

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN}; \tag{3.7}$$

$$F1 - Score = 2. \frac{Precision\ .\ Recall}{Precision + Recall}; \tag{3.8}$$

Table 4.6: Confusion Matrix

Actual Class	Classified as Malware	Classified as Benign
Malware	TP	FN
Benign	FP	TN

4.4 Results and Discussion

Table 4.7 lists the TPR, FPR, Precision, F1-score, recall, AUC and processing time. All the analysed classifiers perform well with an accuracy of 0.90 or more. However, MLP and Decision table dominate with an accuracy of 0.993. In terms of time, Nave Bayesian, Decision Tree and

Decision Table are more efficient than MLP and Random forest. Overall, Decision Table produces the best results.

4.4.1 Performance Comparison of different Classifiers

Performance of the six widely used classifiers is compared in terms of TPR, FPR, Precision, recall, AUC and time taken to build the model. Table 4.7 lists the results obtained for TPR, FPR, Precision and recall. Decision Table outperformed all classifiers in detecting the correct class of malware applications with an accuracy of 99.3% whilst SMO performs worst in terms of measured parameters.

Table 4.7: Comparison of classification algorithms

Algorithm	TPR	FPR	Precision	Recall	Time
Decision Table	0.993	0.006	0.99	0.99	0.23
MLP	0.992	0.008	0.99	0.99	1.18
Decision Tree	0.9861	0.011	0.98	0.98	1.24
Nave Bayesian	0.982	0.012	0.98	0.98	0.95
Random Forest	0.97	0.07	0.97	0.97	0.43
SMO	0.67	0.033	0.67	0.67	0.94

Additionally, the classifiers are evaluated in terms of time taken to build up the model. The Decision Table takes less time than all other classifiers. Overall, results demonstrate that the Decision Table is efficient and accurate classifier as compared to other five algorithms.

4.4.2 Area Under Curve (AUC)

Accuracy of detection is measured by the area under the curve. An area of 1 represents a perfect detection; an area of 0.5 represents a worst detection. Traditionally accepted values[1] for AUC are shown below:-

- 0.90-1 = excellent (A)
- 0.80-0.90 = good (B)
- 0.70-0.80 = fair (C)
- 0.60-0.70 = poor (D)
- 0.50-0.60 = fail (F)

Table 4.8 depicts the AUC values obtained with different classifiers. Decision Table, Decision Tree and MLP have "Excellent" AUC values compared to Random Forest and Naïve Bayesian which have "Good" AUC values. SMO's performance performs poorly of all the classifiers.

Table 4.8 AUC comparison of classifiers

Classifier	**AUC**
Decision Table	99%
Decision Tree	98%
MLP	98%
Random Forest	89%
Naïve Bayesian	87%
SMO	50%

1https://sonoworld.com/fetus/page.aspx?id=1698

4.5 Comparison with related approaches

We compare the performance of PIndroid against relevant approaches which use some of the similar features and analysing the samples acquired from same sources: Google Playstore, Genome and Contagiodump. These are known repositories of malware and benign apps and the performance of most of the state of the art malware detection approaches are tested on these samples with a difference of number of samples tested. The most relevant approaches are Drebin [154], DroidMat [155] and Marvin [156].

Drebin [154] examines the manifest file and decomposed code of app to check the permissions, API calls, hardware resources, app components, filtered intents and network addresses. It uses support vector machines (SVM) for malware classification. Although, they used the largest dataset of 129,013 apps, it consists only 4.5% of malware samples thereby may not be able to learn malware patterns. It used many features opposed to our work which uses only two most effective features. It achieved 94% malware detection rate with 0.01 false positive rate whereas our approach achieved 99% detection accuracy with 0.006 FPR. Drebin [154] requires extensive processing for extraction and execution of a large number of features from the manifest file and app code, it takes more time to analyse the app and therefore is less efficient than our method. It takes on average 10 seconds to analyse an app, whereas our approach takes less than 1 second. Its use of a large number of features may also result in more false alarms as the efficiency and accuracy of feature based detection approaches highly depend on the selection of more relevant and less number of features.

DroidMat [155] analyses some features from the manifest file and smali files of disassembled codes. The extracted features include permissions, components deployments, intent messages and API calls. It applies K-means algorithm for clustering and Singular Value Decomposition (SVD) method for low-rank approximation. The minimized clusters are processed with a kNN algorithm for classification into malware or benign apps. It achieves an accuracy of 97.6% with no reported false positive rate. They analysed 1738 apps consisting of 1500 benign and only 238 malware samples. Malware samples are only 13% of total dataset, which is a non-representative data set for capturing the malware usage patterns. The accuracy is less than our method and the processing time is higher as it needs to perform the execution of smali files and manifest files. Since Smali files are much larger than manifest files, the overall cost of methods which analyse smali files forgoes higher. This holds true for Drebin [154] and DroidMat [155].

Marvin [156] uses off-device static and dynamic analysis for malware detection. It uses around 490,000 features extracted from the manifest files and disassembled codes. Its high-dimensional feature set includes permissions, intents, API calls, network statistics, components, file operations, phone events, app developer IDs, package serial numbers and bundles of other features. It uses a linear classifier to detect malware app and assign a malicious score to the app on a scale from 0 to 10, with 0 being benign and 10 being malicious. They used the largest dataset of

150,000 apps in which only 10% are malware samples. It classifies with an accuracy of 98.24% and false positive rate of 0.04%. Although this approach classifies with the malicious score, this is not an efficient approach considering the high dimensionality of features and regular updating requirement of the database to maintain the detection performance. Since, both the analyses are done off-the-device; the mobile app is just to provide an interface to upload the apk to the analysis server. The static and dynamic analyses of an app take several minutes depending on the size of smali files. This approach is less efficient and less accurate than our approach.

We further compared the detection rate of PIndroid on the unlabelled set of 100 apps against these approaches. PInDroid significantly outperforms the other approaches with TPR of 0.98 and FPR of 0.1. The other approaches provide a detection rate between 0.90 to 0.93 with FPR between 0.7 to 1. Detection performance of compared approaches is shown in Fig. 4.3.

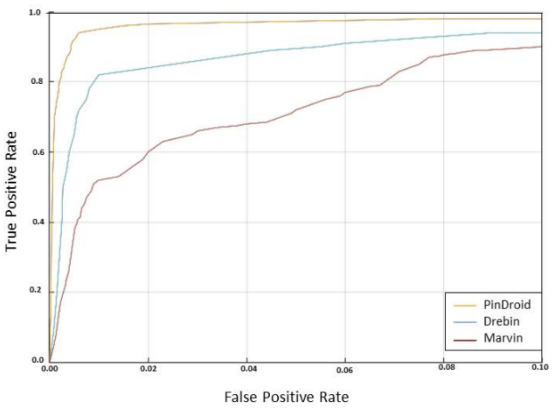


Figure 4.3: Comparison with relevant approaches

The compared approaches are less efficient than our approach in analysing the apps due to their dual processing time. PInDroid gives more accurate results due to the use of most relevant feature set to model the malicious behaviour

4.6 Application of Ensemble Learning Methods

In this section, a study on effect of applying different ensemble methods is presented. The motivation to apply ensemble methods is to ascertain if the performance of poorly performing classifiers can be improved by applying ensemble techniques. If the predictions from each non-over fitting model are combined, then the final aggregated prediction will be less noisy than the single opinion of individual model and there will be no over fitting. We have used different ensemble methods such as boosting, bagging and stacking for combining multiple trained classification algorithms. The predictions from combined classifiers are processed with the help of well know ensemble schemes such as majority voting, average of probability and product of probability.

4.6.1 Ensemble Learning

Ensemble methods combine the results of multiple machine learning algorithms to improve the predictive performance [159, 160]. We use three ensemble methods namely Boosting, Bagging and Stacking to improve the detection results of classification algorithms.

4.6.2 Boosting

In boosting, a base classifier is trained on the training dataset followed by the subsequent stages of classifiers which concentrate on the incorrectly classified instances by the previous classifier. Classifier stages are added till the time there is a limit in the number of models or accuracy [161, 162]. We use popular boosting meta-algorithm AdaBoost, introduced in 1995 by Freund and Schapire.

4.6.3 Bootstrap Aggregating (Bagging)

In bagging, the training dataset is sub-divided into multiple training datasets and each dataset is used to train a classifier as shown in Figure 4.4. Finally, the outputs of all the classifiers are combined by averaging out or majority voting method [162].

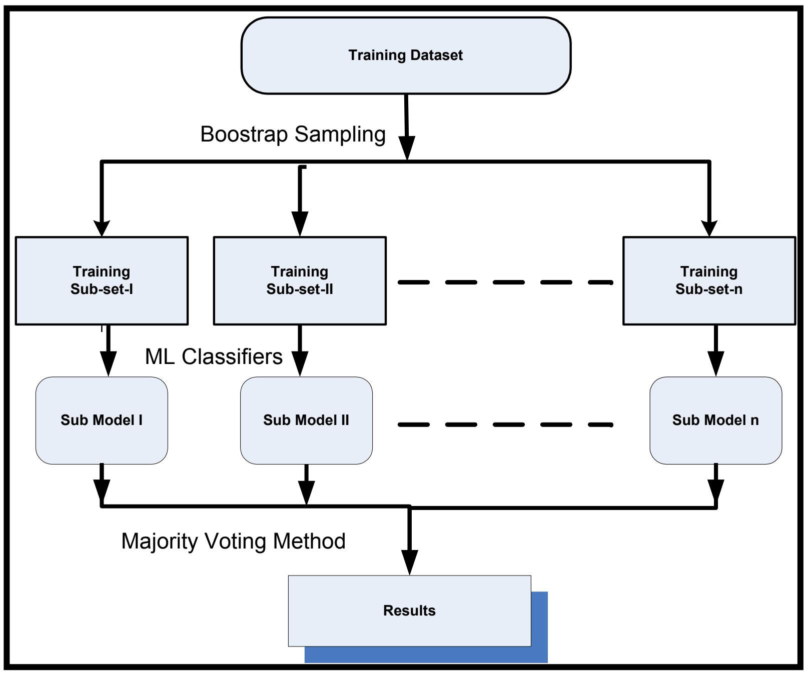


Figure 4.4: Bagging Process

4.6.4 Blending / Stacking

In stacking, multiple algorithms are trained individually with the training dataset and the outputs from the classifiers are sent to a meta-classifier which combines the results of the base classifiers using any of the three schemes: an average of probabilities, a product of probabilities and majority voting (Figure 4.5).

Decision Table, MLP, and Decision Tree classifiers are applied in first stage and their results are combined with different schemes as mentioned above.

(a) Average of probabilities

It takes an average of the probabilities of each class from the individual classifiers (*k=3* for three classifiers) and compares which class has greater probability such that,

$$Malware, if \quad P_{avg}\sum_{k=1}^{3} Class_{malware} < P_{avg}\sum_{k=1}^{3} Class_{benign}; \tag{3.9}$$

$$Benign, if \quad P_{avg}\sum_{k=1}^{3} Class_{malware} > P_{avg}\sum_{k=1}^{3} Class_{benign}. \tag{3.10}$$

(b) Product of probabilities

Product of probabilities is taken from each of the classifiers and highest probability of class is assigned as:

$$Malware, if \quad P_{avg}\prod_{k=1}^{3} Class_{malware} < P_{avg}\prod_{k=1}^{3} Class_{benign}; \tag{3.11}$$

$$Benign, if \quad P_{avg}\prod_{k=1}^{3} Class_{malware} > P_{avg}\prod_{k=1}^{3} Class_{benign}. \tag{3.12}$$

(c) Majority vote

The final result is decided based on the results obtained from the majority of the results. Results of ensemble classification are depicted in Tables 4.9 to 4.11. The product of probabilities method yields the best results.

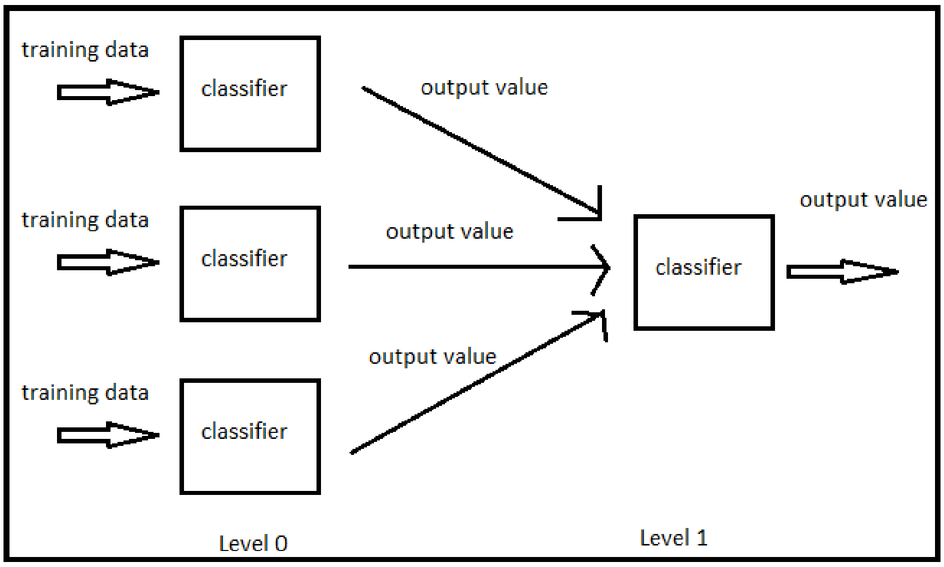


Figure 4.5: Stacking Process

4.6.5 Results of Ensemble methods

Ensemble methods are applied on different datasets and a considerable amount of improvement is noticed in the performance of model. Following four cases are particularly noticeable due to their distinct nature.

(i) The worst model with SMO with an accuracy of 67% improved to an accuracy of 94.6% after applying stacking method (Table 4.9).

Table 4.9: Accuracy gain in SMO model with Stacking

Dataset	**Meta. AdaBoo**	**Meta. Bagging**	**Meta. Stacking**
Classification	67.84	63.12	94.6

(ii) The minimum accuracy obtained with Decision table without ensemble methods was 99.3% which increased to 99.5 with bagging method (Table 4.10).

Table 4.10: Accuracy gain in Decision Table model with Bagging

Dataset	**Meta. AdaBoo**	**Meta. Bagging**	**Meta. Stacking**
Classification	99.3	99.51	99.38

(iii) The minimum accuracy obtained with Naïve Bayesian before applying ensemble methods was 98.2% which improved to 99% with stacking and 98.31% with boosting method (Table 4.11).

Table 4.11: Accuracy gain in Naïve Bayesian model with Boosting and Stacking

Dataset	**Meta. AdaBoo**	**Meta. Bagging**	**Meta. Stacking**
Classification	98.35	98.31	99

Ensemble methods combine results from multiple machine learning algorithms to improve the predictive performance [159]. It is not necessary that the performance of ensemble learning be better than the individual classifiers. The stacked performance depends on the selection of classifiers and methods used to combine the output predictions [160].

We apply three ensemble methods: Boosting, bagging, and stacking to further improve the detection accuracy. Stacking gives the better results as compared to boosting and bagging.

4.7 Summary

Android security model relies on permission and intent based mechanisms for controlling access to vital hardware and software components. However, so far these two features have not been combined together for detection of malware.

In this chapter, Android permissions and intents are investigated for using them to detect the malware apps. We also investigated the correlation between permissions and intents by applying statistical testing methods. It was also studied how effective is to combine permissions and intents analysis for malware detection. Permissions and intents are found to be most effective features of Android for characterizing malware as they are easy to extract from the manifest files of apps and require less processing time and complexity. This work proposed a novel malware detection method—PInDroid which is based on these two key features. Various well known classification algorithms were applied on the dataset. Application of classification algorithms have given very encouraging results to further advance the work.

In this chapter, we also applied different ensemble methods such as Boosting, Bagging and stacking on six well Machine Learning classifiers: Naïve Bayesian, Decision Table, Decision Tree, Neural Network (MLP), SMO and Random Forest. All the classifiers demonstrated an increase in performance at different ensemble methods. Accuracy of some of the classifiers improved with bagging method and some of them improved with stacking method. These methods have improved the overall results significantly thus increasing the confidence level.

It was observed during repeating the experiments that it is not necessary to get good or equal results with different ensemble methods. Results of ensemble learning depend on the classifier itself and the combination of classifiers chosen for cascading stages.

Chapter 5

Detection of Colluding Applications

5.1 Introduction

Android being the most popular platform for mobile devices is under proliferated malicious attacks. A recent threat is from app collusion; in which two or more apps collaborate to perform stealthy malicious operations by elevating their permission landscape using legitimate communication channels. Each app requests for a limited set of permissions which do not seem dangerous to users. However, when combined, these permissions potentially inflict a number of malicious attacks. Mobile users are generally unaware of this type of permission augmentation, they consider each app separately. Hence, their decision to install apps is limited in perspective due to unawareness of such type of capability escalation [164].

Android implements sandbox and permission based access control to protect resources and sensitive data, however, being open source and developer-friendly architecture, it facilitates sharing of functionalities across multiple apps. It supports useful collaboration among apps for the purpose of resource sharing; however, it also introduces the risk of app collusion when the collaboration is done with malicious intention. Cyber criminals exploit this vulnerability to launch distributed malicious attacks [165].

This chapter investigates Android application collusion and intents related attacks with an intention to furnish a feasibility study of using our permissions and intents based methodology for detection of malicious colluding apps. Most of the recent works on app collusion investigate permissions and IPC mechanism to understand their role in app collusion. Our preliminary investigations confirm that permissions and intents can be exploited to detect malicious app collusion.

The rest of the chapter is organized as follows: Section 5.2 presents an overview of app collusion, Section 5.3 investigates the technical details of app collusion and covert channels, and Section 5.4 presents the IPC and intents related attacks. Section 5.5statistical testing details carried out to understand the correlation between permissions and intents. Section 3.5 presents the challenges faced in detecting app collusion and recommend potential measures. Section 5.7 proposes a possible generic framework for detection of colluding apps and finally Section 5.8 summarizes the chapter.

5.2 Overview

Application collusion is possible with Inter Process Communication (IPC), covert channels or system vulnerabilities. Malicious colluding apps are explicitly designed by cyber criminals by using different tactics which includes the development of apps with same User ID. Such apps have more chances for a successful collusion attack. In some cases, mis-configured apps also participate in the collusion attack with a complete obliviousness of colluding app [166]. One of the collusion scenarios is illustrated in Figure 5.1: App 'A' has no permission to access the internet; however it has permissions for camera. Similarly, App 'B' has no permission for the camera but can access the internet. Assuming that the components of both apps are not protected by any access permission, they could collude to capture the pictures and upload on a remote server through the Internet.

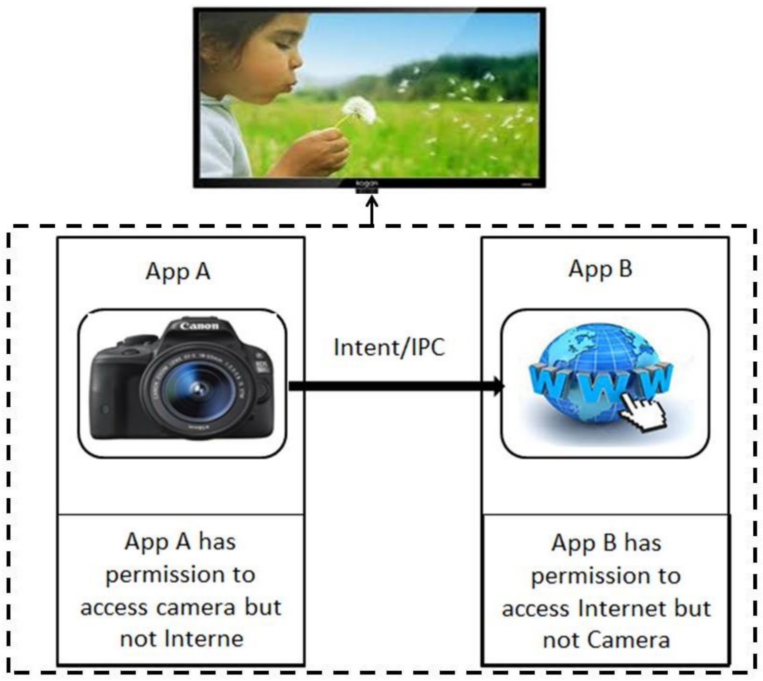

App A
Intent/IPC
App B
App A has permission to access camera but not Interne
App B has permission to access Internet but not Camera

Figure 5.1: Application Collusion Scenario

Until recently, a small scale research is done on app collusion primarily due to non-availability of known samples of colluding apps for analysis and experimentations [167]. Most of the works accentuated on rummaging of covert channels and development of experimental colluding apps. As a result of this innovative approach, the research on collusion gained a little momentum and there are now a few collusion detection approaches available each with a limited scope. Despite the growing research interest, detection of malicious colluding apps has been a challenging task [168].

The fact that permissions and intents (which are the key features of our detection model) are the main features behind the application collusion, our proposed malware detection model is particularly suitable for detection of colluding applications in addition to other types of malware applications.

In Android, all applications are treated as potentially malicious. They are isolated from each other and do not have access to each other's' private data. Each app runs in its own process and by default can only access own files. This isolation is enforced with the sandbox, in which each app is assigned with a unique user identifier (UID) and own Virtual Machine (VM). App developers are required to sign the apps with a self-certified key. Apps signed with the same key can share UIDs and can be placed in a same sandbox [169].

Android app comes as .apk file, which contains the byte code, data, resources, libraries and a manifest file. Manifest file declares the permissions, intents, features and components of an app. The components that can be handled by an app are declared with intent filters. System resources and user data are protected through permissions. Figure 5.2 illustrates the communication between apps in a sandbox environment. App 1 can use only those system resources and user data for which it has permissions. Similarly, app 2 is also limited to use certain resources. Although both apps have limited permissions to access the resources but through IPC, they are able to augment their permissions and get over-privileged access to system resources and user data.

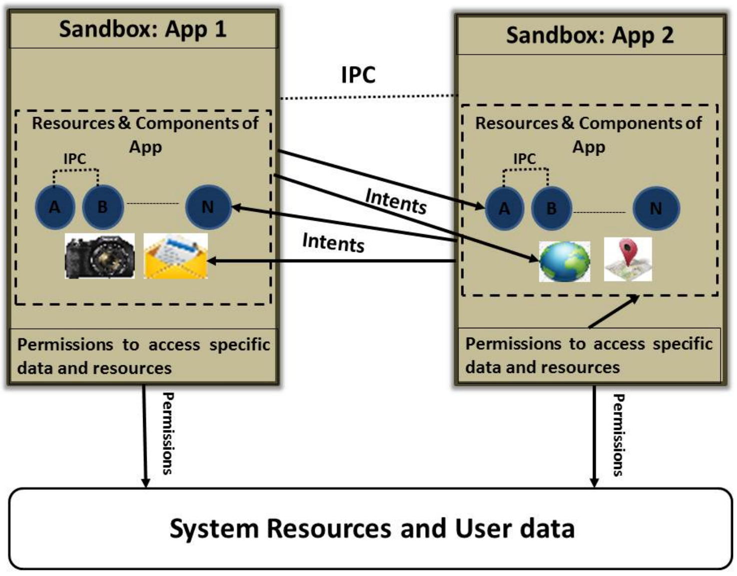

Sandbox: App 1
Sandbox: App 2
IPC
Resources & Components of App
IPC
A
B
N
Resources & Components of App
IPC
A
B
N
Intents
Intents
Permissions to access specific data and resources
Permissions to access specific data and resources
Permissions
Permissions
System Resources and User data

Figure 5.2 Inter Process Communication

5.3 Investigations of Application Collusion

Colluding applications are those applications that cooperate in some manner to perform extended operations which they would independently be unable due to their respective permission restrictions. These applications can perform covert operations even without breaking the security framework or exploiting any system vulnerabilities [170].

Application collusion can inflict serious damages to the user by stealing user’s data or device resources. Following elements of Android architecture directly or indirectly contribute in app collusion:-

- **Permissions**

  Permissions are used to restrict the access of system resources and user data on the device.

- **Shared User ID**

  Android assigns a unique user ID to each app to ensure that it runs in its own process and can only access the allocated system resources. Apps with shared User IDs (shared Userid) can access each other's data and can run in same process, thereby limiting the effectiveness of isolation provided with user ID.

- **Components**

  Components are the basic modules that are run by apps or the system. There are four types of components: Activities, Services, Content Providers and Broadcast Receivers.

- **Intents**

  Intents are messages used to communicate between the components of apps. These messages are used to request actions or services from other application components. Intents declare the intention to perform an operation [166]. Intents are of two types: Explicit and Implicit. Explicit intent specifies the component exclusively by class name. Implicit intent does not specify a particular component by name. Apps with implicit intent only specify the required action without specifying particular apps or component. System selects the app from device which can perform the requisite task. Implicit intents are vulnerable to exploits as they can combine operations of various applications, if they are not handled properly.

- **Sandboxing**

Sandboxing isolates an app from other apps and system resources. Each app has a unique identifier and has access to the allocated System files and resources against the unique identifier. An app can also access files of other apps that are declared as readable/writeable/executable for others.

- **Access Control Mechanism**

In Android, the access control mechanism of Linux prevails. It controls access to files by process ownership. Each running process is assigned a UserID and for each file, access rules are specified. File access rules are defined for a user, group and everyone, thus granting permissions to read / write / execute the file.

- **Application Signing**

Cryptographic signatures are used for verification of app source and for establishing trust among apps. Developers are required to sign the app to enable signature based permissions, and to allow apps from the same developer to share the UserID. A self-signed certificate of the signing key is enclosed into the app installation package for validation at installation time.

5.3.1 Covert Communication Channels

A covert channel is a stealthy mechanism which exploits resources and uses them to exchange information between apps in a manner that it cannot be detected [171]. There are two types of covert channels: Timing and Storage. Timing channels modulate the time spent on execution of some task or using some resource. Storage channels relate to modifying the data item such as configuration changes etc. Example of covert channel is sending user data to a remote server by encoding it as network delays over the normal network traffic [172]. Figure 5.3 depicts a covert channel, where a file of 20 bytes containing some data is sent through a normal communication channel. The file size is covert information. This information might not be of any importance to the receiver but significantly valuable for the malicious party.

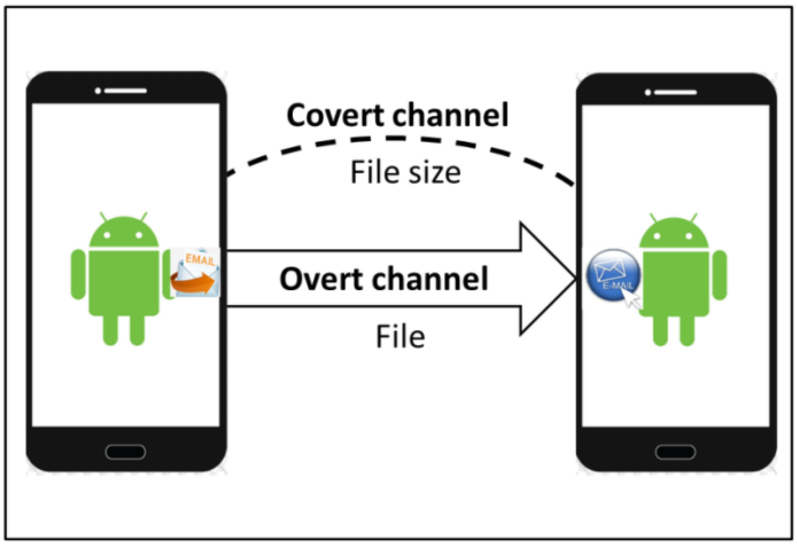


Figure 5.3: Overt and covert channel

Covert channel typically exploit the shared resources to read, store and modify data as a medium for communication between two malicious entities. This type of information exchange is different from IPC based resource sharing. App collusion through covert channels is investigated by implementing high throughput covert channels in [165].

5.4 IPC related Attacks

Android security builds upon sandbox, application signing and permission mechanism. However, these protections fail if the resource and task sharing procedures provided through IPC are used with malicious intentions. In this section, we discuss most common IPC related attacks on Android devices.

5.4.1 Application Collusion Attack

In application collusion attack, two or more apps collude to perform a malicious operation which is broken into small actions [165]. Each of the participating apps communicates using legitimate communication channels to perform the part assigned to them. Apps do not need to break any security framework or exploit the system vulnerabilities for carrying out a collaborative operation [168]. Colluding attack help in malware evasion as the current commercially available anti-malware solutions do not have capability of simultaneously analyzing multiple apps to detect collusion.

5.4.2 Privilege Escalation Attack

In privilege escalation attack, an application with less permissions access components of more privileged application [172].This attack is prevalent in misconfigured apps mainly from the third party market. The default device applications of phone, clock and settings were also vulnerable to this attack [173]. Confused deputy attack is a type of privilege escalation attack. A compromised deputy may potentially transmit the sensitive data to the destination specified in the spoofed intent (Fig. 5.4). Consider an app which is processing some sensitive information like bank details at the time of receipt of spoofed intent. It is likely that such information may be passed on to the url or phone number defined in the malicious intent.

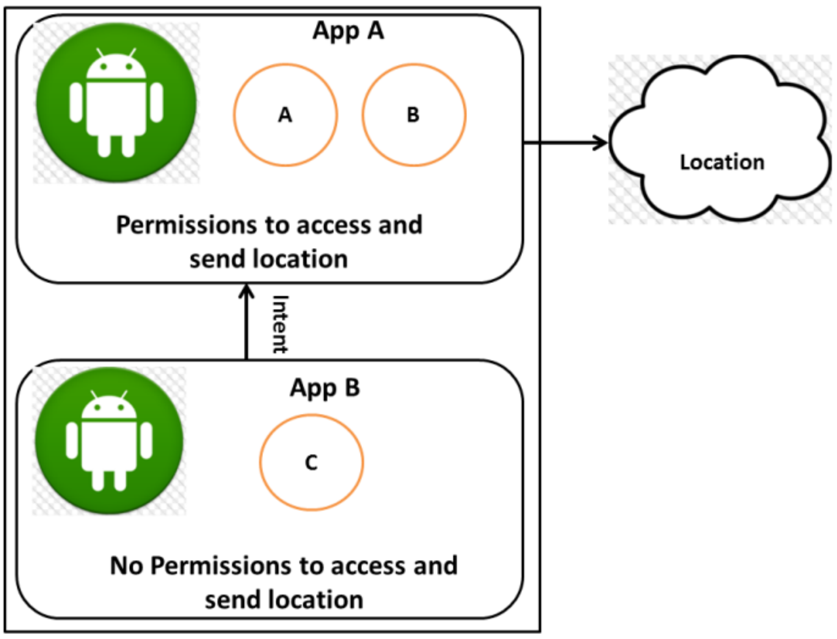

App A
A
B
Permissions to access and send location
Location
Intent
App B
C
No Permissions to access and send location

Figure 5.4: Confused Deputy Attack

5.4.3 Intents related Attacks

Explicit and implicit intents may potentially assist in colluding attacks. Although, explicit intents guarantee the success of collusion between apps, implicit intents can also be intercepted by the malicious apps with matching intent filters. We discuss some of the known intents related attacks.

- **Broadcast Theft**

  A public broadcast sent by application is vulnerable to interception. As shown in Figure 5.5, a malicious app 'M' can passively listen to the public broadcasts while the actual recipient is also listening. If a malicious receiver registers itself as a high priority receiver in ordered broadcasts and receives the broadcast first, it could stop the further broadcasting to the recipients. The ordered broadcasts are serially delivered messages to the recipients that follow an order according to the priority of receivers. Public and ordered broadcasts may cause eavesdropping and Denial of Service (DoS) attacks [167].

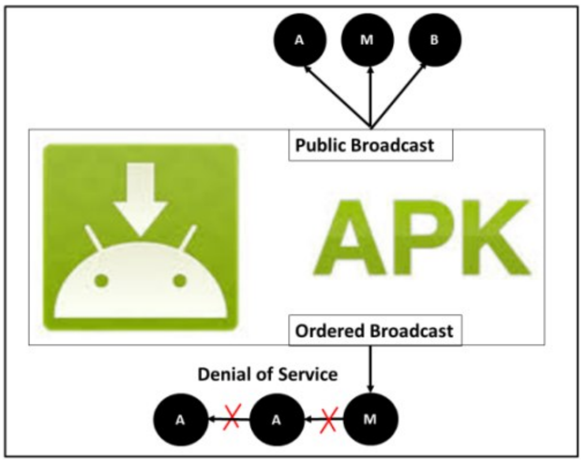


Figure 5.5: Broadcast Theft Attack

- **Activity Hijacking**

If a malicious app registers to receive the implicit intent, it may launch activity hijacking attack on successful interception of intent. With activity hijacking, a malicious activity can illegally read the data of the intent before relaying it to the recipient [165]. It can also launch some malicious activity instead of the actual one. Consider a scenario, in which an activity is required to notify the user for the completion of certain action. The malicious user can falsely notify the user for the completion of uncompleted activity like un-installation of app or transaction completed.

- **Service Hijacking**

If an exported service is not protected with permissions, it can be intercepted by an illegitimate service, which may connect the requesting app with a malicious service instead of the actual one [5]. In this attack, the malicious user hijacks the implicit intent which contains the details of service and start the malicious service in place of the expected one. Implicit intents are not guaranteed to reach to the desired recipient because it does not exclusively specify the recipient. A malicious app can intercept an un-protected intent and access its data by declaring a matching intent filter [6]. This type of attack may be used for Phishing, Denial of Service (DoS) and component hijacking attacks are possible with unauthorized intent receipt.

- **Intent Spoofing**

In Intent spoofing attack, the malicious app controls the unprotected public component of a vulnerable app. It starts performing as the deputy of the controlling app and carries out the malicious activity on behalf of the controlling app [3]. This type of attack is also known as confused deputy attack as the deputies (victim apps) are unaware of their participation in the malicious activities. Figure 5.4 illustrates the confused deputy attack. A malicious broadcast injection is also possible with spoofed intent when a broadcast receiver that is registered to receive the system broadcasts trusts an incoming malicious broadcast as a legitimate one and performs those actions which need system triggers.

5.5 Detection of colluding applications

Detection of app collusion is a very complex proposition. There are a number of challenges in designing a solution to detect the malicious colluding apps and there remain big question marks over efficacy of such solutions. This is the prime reason that we don't have a lot of reliable choices available for such detections.

5.5.1 Challenges

First challenge in detection is classification of IPC into benign and malicious groups. Android is an open source platform, which encourages resource sharing among apps by re-using the components. IPC is mainly used by apps to interact with different inter and intra components. The main problem is to distinguish between the benign collaboration and malicious collusion. Such a distinction is likely to come up with a cost of very high false positives. Keeping the false positive rate to lowest is another problem.

Secondly, considering the substantial number of apps available in the Android market (more than 2 Million apps by Feb 2016), there is a difficulty of analyzing pairs of apps. It is computationally challenging and cost exorbitant to analyze all possible pairs of apps to detect the malicious collusion between sets of apps given the search space. Analysis of all possible app pairs of total of N apps would require N2 pairs. Similarly, to analyze sets of three colluding apps, it would require analyzing N3 apps. An effective collusion detection tool must be capable of isolating potential sets of apps and carrying out further investigations.

Another glaring challenge is the presence of a number of covert channels in the system. Detection of covert channels is an NP-hard problem as it would require monitoring of all the possible communication channels [174]. Covert channels are difficult to detect because they use overt channels for conveying stealthy information. Lastly, known malicious colluding apps are not available for analysis. The non-availability of known samples of colluding apps, makes it difficult to validate the experiment results. Analysis and validation of collusion detection is a quandary, we need known samples of colluding apps to validate the detection method, but to find the samples, a reliable detection method is mandatory, which itself is not available in an authenticated form.

An effective collusion detection system must overcome the aforementioned challenges and encompasses an integrated solution. The detection of IPC based collusion have been recently proposed in a few research papers [175], [176], [177], and [178]. The proposed approaches have a number of limitations and the accuracy and efficiency of these methods is questionable due to non-availability of universally accepted dataset of malware colluding apps.

The solution proposed in [174] is to re-design the security model of Android system to mitigate the risk of collusion. However, this would involve a big cost and complexity in re-writing the OS components and ensuring their compatibility and smooth functioning in conjunction with already available millions of apps in the Android market.

Another approach [175] is limited to the detection of collusion based on intents only. It analyzes the interaction of components through intent filters only and analyzes only two apps at a time. Currently, this approach suffers with a high false positive rate. It is a memory consuming approach which may not be feasible for mobile phones keeping in view the limited memory of phones. It is likelihood that extensive memory consumption may deteriorate the overall performance of device. Similarly, [176] is also mainly based on intent messages. This approach faces the challenges of conventional rule based methods that are prone to evasion with obfuscation and evasion. Scalability is a major drawback of their approach.

Malware collusion detection tool [177] supports the latest API versions only, hence analysis of apps developed under earlier versions is not possible. Technical details of the tool are not available for performance verifications and evaluations. It generates a high number of false alarms mainly due to its reliance over information flows.

The detection of covert channels is still an under explored research area. So far, there are two works [173] [178], which attempts to detect the covert channels based app collusion. Currently, [173] has a limited scope of detecting only covert channels related to shared resources such as reading of the voice volume, change of the screen state and change of vibration settings etc. However, the approach can be investigated for inclusion of other covert channels. Similarly, [178] handles only data flows.

5.5.2 Potential Measures

The complexity and challenges of collusion detection merit a hybrid framework. As a result of our analysis, we recommend an integrated approach for detection of app collusion. We also suggest that a covert channel may not be detected in isolation, but its existence may be realized whilst analyzing the IPC related security breaches. We argue that any mobile user downloads a limited number of apps as opposed to available millions of apps. A user cannot install millions of apps on a single device; hence, there is no need to analyze the millions of app pairs or triplets for possible collusion. On the average, a mobile user installs 20 to 30 apps. A system capable of analyzing 502 or 503 apps is sufficient for a common mobile user. This solution may also be augmented with a cloud based analysis engine if the number of concurrently analyzed apps is increased to 4, 5 or more. Cloud based analysis is an efficient and cost effective approach for high computational operations. We argue that adopting such an approach is essentially required to facilitate the identification of sets of colluding apps from a dataset of millions of apps.

Since permissions and intents facilitate inter and intra app communication and collaboration. Analysis of usage pattern of permissions and intents has potentials to detect app collusion through IPC and covert channels. Adding shared user IDs and publicly declared intents is also recommended as the collaborating apps may use same User IDs to make sure that the attack is successful.

5.6 Proposed framework for Detection of Colluding Apps

There are different methods with which applications can collude; however shared user ID and public declaration of intent are two key features of Android OS which are more vulnerable to collusion attacks.

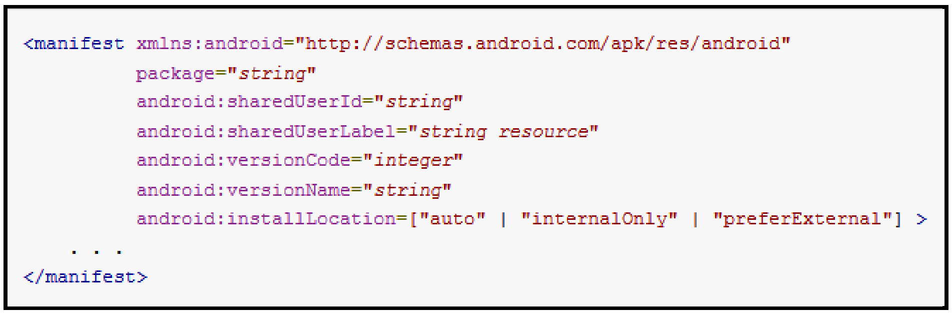
(a) Shared User ID

In the UID assignment step, sharing UID is checked in the manifest. If the sharing UID exists, Android checks other applications' share User Id. If they match to each other, this application is assigned with the existing UID. If no applications match or no sharing UID in the manifest, a new UID is assigned to this application. In the permission assignment step, if the UID is new, this UID will have all permissions requested in the manifest if the users approve. If the UID is shared, this application will not only have its own requested permissions, but also the permissions of other applications with the same UID.

Application sandbox is a means to isolate the applications from each other in the Android system by assigning a UID and a set of permissions [179]. When the application is installed on the device, it runs in its own sandbox and other applications cannot access or interfere. An application can only access its own files, unless other applications explicitly assign the access permissions to this application. For example, if the applications are created by the same developers, the developers can make these applications share the same UID, then these applications will run in the same sandbox and share the resources in that sandbox.

Application signing is used to ensure the application security. It creates a certification between developers and their applications. Before placing an application into its sandbox, the application signing creates a relationship between the UID and the application. The applications couldn't be run on the Android without signing. With the same UID, that is, running in the same sandbox, the applications can share the permissions and communicate with each other. By using application signing, the application update process can be simplified. Since different versions of the same application have the same certificate, the package manager can verify this certificate. Then, the old version is replaced; the new version can have the permissions already granted to the old version. What's more, the application signing can also ensure that an application cannot communicate with another app unless using the ICC. But if the author is the same, the author can use the same application signing to enable the direct communication among his/her applications.

Android OS assigns a unique user id to each application to ensure that it is run in its own process and resources created against that id. However, aapplications can share their user ids if they are developed with same signature or certificate and applications with the same user ids (shared Userid) can access each other's data and can be run in the same process.Shared UserIds are declared in application’s Manifest file as shown in Figure 5.6.

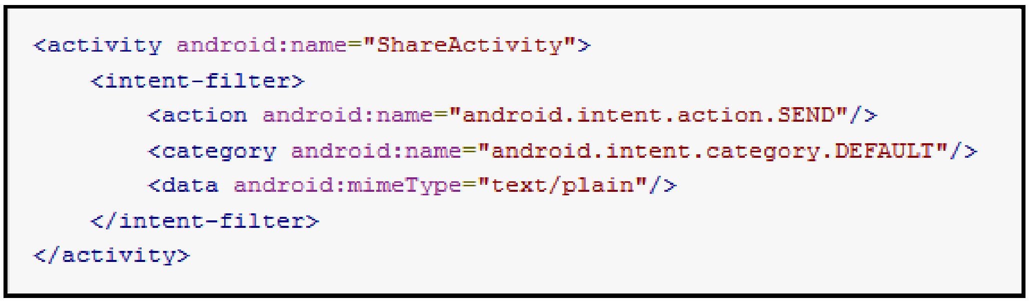


```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="string"
          android:sharedUserId="string"
          android:sharedUserLabel="string resource"
          android:versionCode="integer"
          android:versionName="string"
          android:installLocation=["auto" | "internalOnly" | "preferExternal"] >
    . . .
</manifest>
```

Figure 5.6: Declaration of Shared User ID in Manifest file.

(b) Implicit Intent declaration

Implicit intents specify the action it needs to perform without specifying particular apps/component which can only be used for that action. Implicit intents are vulnerable to exploits as they can combine operations of various applications, if they are not handled appropriately. Applications can receive implicit intents from other apps if they advertise/declare their components with an intent filter. If the declared intent filter of app matches all the fields of requesting intent then system will pass on the implicit intent the declaring app. Intrinsic intents are declared in the manifest file as depicted in Fig. 5.7.



```
<activity android:name="ShareActivity">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

Figure 5.7:. Declaration of Public Intrinsic Intent.

Adding these two key features in our detection model enables the further classification of malware apps as colluding apps or vice versa. This can be realized with an additional layer of classifier or through adding few lines into the code. Figure 30 depicts the additional layer for the detection of colluding apps.

The proposed system is shown in Figure 5.8. In first stage, apps are analyzed to identify those which share user IDs as they have more potential to collude successfully. In second stage, permissions and intents are extracted and analyzed for source permission, source intent, sink permission and sink intent. Pairwise communication mappings of apps are generated from the source and sink permissions and intents. The identified communicating pairs of apps are further analyzed to check if their communication is limited to each other or more apps. The classifier stage is used to classify the app into colluding or non-colluding ones and users are notified for possible collusion. In the proposed approach, permissions and intents are grouped into four categories: source permissions, source intents, sink permissions and sink intents. Source permissions or intents are those that initiate some operation, whereas the sink permissions and intents are those which act upon to complete the required operation [164].

With additional policy refinements, the identified colluding apps can be classified into benign and malicious apps. This approach may be integrated with the methodologies proposed in [178] and [173] to monitor the data flow sources and sinks of IPC and tracking of shared resources. Information flow system proposed in [178] to monitor the data flow sources and sinks in IPC is a good trade-off for detecting the covert channels however, it lacks the tracking of shared resources. Mapping structure of [173] helps in tracking the shared resources used by two interacting apps.

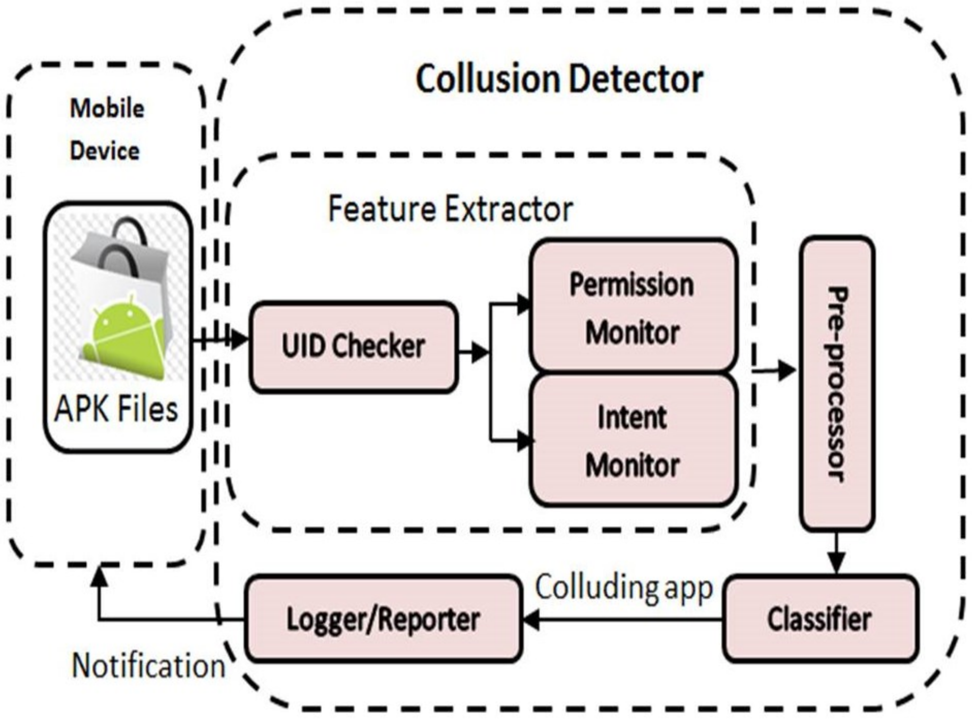


Figure 5.8: Collusion Detection Model

Effective detection of app collusion requires monitoring of IPC and all possible covert communication channels: shared resources and data flow sources and sinks. An integrated system comprising of the proposed framework and Taintdroid [178] for analyzing the covert channels is a good starter towards a comprehensive detection system.

5.7 Related Work

IPC and intents have not been explored the way permissions have been investigated. Most of the existing IPC based studies focus on finding the IPC related vulnerabilities. [179] investigated the IPC framework and interaction of system components. [166] detects the IPC related vulnerabilities. [180] suggested improvement in ComDroid by segregating the communication messages into inter and intra-applications groups so that the risk of inter-application attacks may be reduced. [181] characterized Android components and their interaction. They investigated risks associated with misconfigured intents. [182] examined vulnerable public component interfaces of apps. [183] generated test scenarios to demonstrate the ICC vulnerabilities. [184] performs information flow analysis to investigate the communication exploits. [185] investigated intents related vulnerabilities and demonstrated how they may be exploited to insert the malicious data. Their experiments found 29 out of a total of 64 investigated apps as vulnerable to intent related attacks. Similarly, [186] investigated the ICC vulnerabilities. All of these works focus on finding communication vulnerabilities, and none of them used IPC and intents for malware detection.

5.8 Summary

The concept of colluding apps has emerged recently. App collusion can cause irrevocable damage to mobile users. Detection of colluding apps is quite a challenging task. Some of the challenges are: distinction between the benign and malicious collaboration, false positive rate, presence of covert channels and concurrent analysis of millions of apps. Existing malware detection system is designed to analyse each app in isolation. There is no commercially available detection system which can analyse multiple apps concurrently to detect the collusion. We have carried out a preliminary study to evaluate the applicability of our proposed approach for detection of collaborating apps.

In this chapter, we discussed the current state and open challenges to detection of colluding apps. To address the problem, we have proposed an integrated approach to detect app collusion. However, due to non-availability of real colluding app samples, it was not possible to validate the framework. The complexity of problem merits collaborative large scale investigations to mitigate a very large number of known and unknown communication channels between apps besides known IPC and covert channels. Our future work aims to validate the proposed framework on real colluding apps.

Chapter 6

Conclusions and Future work

6.1 Introduction

This chapter concludes the author's work by revisiting the thesis goals, contributions and achieved objectives. This includes the author's work: PInDroid, the permissions and intents based solution that can detect malicious apps accurately and efficiently.

This work also validates different well known classification and clustering algorithms for comparing their performance in malware detection. We found that classification algorithms are more accurate as compared to clustering algorithms for malware detection.

Different ensemble methods are also applied on the models to ascertain the margin of performance improvement. Detection accuracy of proposed model is further optimized with boosting, bagging and blending methods.

The author also demonstrated the usefulness of PInDroid methodology by implementing it through an automated malware detection algorithm: ***AndroPIn***.

The author also described two additional studies, which investigated the usefulness of the proposed methodology to detect the malicious colluding apps and obfuscated malware apps.

Lastly, the author discusses possible future directions of research based on top of research performed by the author after systematically reviewing the work related to Android malware.

6.2 Restating Research Problems and Research Goals

In Chapter 2, the author performed an extensive survey on the existing work related to the analysis, detection, and classification of Android malware to identify the research gaps. This resulted in four thesis goals discussed below:

Goal 1 outlined the importance of analysing the features extractable from the manifest file such as permissions and intents as they are widely used by apps to perform basic operations and can help in understanding the behaviours of malicious apps. As these features do not need run-time analysis, the static approach is used for efficiency purpose.

Goal 2 outlines the use of best classifier for achieving the best accuracy. The selection of the classification algorithm is done after comparing different algorithms against globally accepted performance metrics.

While highly detailed data tends towards higher accuracy, excessive or redundant data increases performance costs and decreases the efficiency of a framework. Thus, we introduced Goal 3, which influenced a smaller, more concentrated, set of features to work with. This allowed the author to improve accuracy with less performance sacrifices, a trade-off issue common when dealing with large datasets.

Lastly, despite of efficient and accurate framework, it is ineffective if malware can evade analysis or detection. We discovered this to be a problem with several frameworks, as they were vulnerable to obfuscation and evasion. Thus, we introduced Goal 4 to develop frameworks that were resilient to code obfuscation.

6.3 Research Contributions

In the introduction chapter, the author stated the contributions of this work and the novel research aspect of each contribution. In this section, we elaborate on the contributions of this work.

6.3.1 Android Malware Detection: PInDroid

In Chapter 4, we proposed PInDroid, a permissions and intents based methodology to distinguish between malicious and benign apps. Android security model relies on permission and intent mechanisms for controlling access to vital hardware and software components. However, these features were never used jointly to investigate their effectiveness in the malware detection. This work is the first one that proposes a novel malware detection method based on these two vital security features.

The basic work of investigations on the identified features: permissions and intents were completed in chapter 3, which resulted in identification of the usage patterns of permissions and intents by malware and benign apps. The author's role in identifying the effective combinations of permissions and intents and automatically performing extraction has been instrumental. This resulted in novel, efficient and accurate permissions and intents based Android malware detection solution. The resulting model fulfilled the author's goals of a robust, efficient, and accurate analysis solution.

6.3.2 Malware Classification using suitable ML Classifier

Different well known classification and clustering algorithms were investigated for comparing their performance in malware detection. We found classification algorithms more efficient and useful as compared to clustering algorithms for malware detection.

Using the author's work in PInDroid, the author then provided a novel feature set to feed to six ML classifiers. Performance of classifiers was then compared in terms of false positive rate, true positive rate, precision, recall, and accuracy. In order to further optimize the classification results, different ensemble methods were also applied to ascertain the margin of performance improvement. It was ascertained through experiments that the performance of PInDroid can further be increased with boosting, bagging and blending ensemble methods.

6.3.3 Implementation of Methodology through an Algorithm: AndroPIn

The author also implemented the proposed approach in form of an algorithm to automatically detect the malware. The algorithm: AndroPIn is implementable as either a client end or a cloud based solution. The algorithm finds dangerous permissions and dangerous intents in the malware app and verifies against the malicious threshold.

6.3.4 Investigation for Detection of Obfuscated and Colluding apps

The last segment of work in this thesis comprises of two studies. First study investigated the effectiveness of the proposed approach of PInDroid for detection of obfuscated malicious apps and the second study explored if the approach can be used for detection of colluding apps. We investigated the techniques used by apps for possible collusion and found that permission model and Intent are the basic essence of collusion. This fact strengthens our approach for possible detection of collusion apps. The ancillary work shows that the permissions and intents based solution can be used to detect the colluding apps. Furthermore, the work shows possible applications for detection of obfuscated malware, which are difficult to detect with other solutions.

6.4 Future Work

There are many directions to advance the work that has been presented in this thesis. First of all, as a future work, we aim to validate our

PInDroid approach on more malware and benign samples to evaluate its performance on diversified malware families and benign categories.

The second area for future work is to implement the methodology for detection of colluding apps. It can be integrated with other state-of-the-art solutions as discussed in chapter 5 and to validate the integrated solution on real colluding apps.

Another possible future work could be to determine whether there are better machine learning methods than the ones used in this approach. Many available machine learning and deep learning approaches have not yet been tested for the most appropriate method.

Similarly, multi-class classification would be an interesting area of work. Applications can be classified into three categories: malware, benign and greyware thereby giving mobile users more flexibility to draw the peripheries between the applications.

AndroPIn implementation can be improved for better performance. More samples need to be tested to validate the algorithm. Malicious scoring of malware apps is another area which can further improve and widen the malware detection.

6.5 Concluding Remarks

Android, and Android malware are rapidly evolving as of the year 2017. Therefore, it is imperative to continue the research on emerging malware threats and their mitigation solutions. In this thesis, a comprehensive survey on the existing work on Android malware detection and classification is presented and research gaps have been identified. The culmination of these observations lead to a novel malware detection approach: PInDroid, which efficiently and accurately detect most of the malware from permissions and intents analysis. The approach is implemented as an algorithm: AndroPIn to automatically detect the malware. The permissions and intents based system has potential to detect the malicious colluding apps besides the obfuscated malware apps.

Bibliography

[1] Wei, Xuetao, L. Gomez, I. Neamtiu, and M.Faloutsos. "Permission evolution in the android ecosystem." In 28th Computer Security Applications Conf., pp. 31-40, 2012.

[2] Enck, William, D. Octeau, P. McDaniel, and S.Chaudhuri, "A Study of Android Application Security," USENIX Security Symposium, pp. 21–37, 2011.

[3] Seo, Seung-Hyun, A. Gupta, A. M.Sallam, E.Bertino, and K.Yim, "Detecting mobile malware threats to homeland security through static analysis," J. Network and Computer Applications, pp. 43-53. 2014.

[4] Felt, A. Porter, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," In Proc. of the 2nd USENIX conf. on Web application development, pp. 7-7,2011.

[5] Sarma, Bhaskar Pratim, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. "Android permissions: a perspective combining risks and benefits." In Proceedings of the 17th ACM Symp. Access Control Models and Tech., pp.13-22, 2012.

[6] Do, Quang, Ben Martini, and Kim-Kwang Raymond Choo. "Exfiltrating data from Android devices." Computers & Security 48 (2015): 74-91.

[7] Tan, Darell JJ, Tong-Wei Chua, and Vrizlynn LL Thing. "Securing android: a survey, taxonomy, and challenges." ACM Computing Surveys (CSUR) 47, no. 4 (2015): 58.

[8] Wu, L., Grace, M., Zhou, Y., Wu, C., & Jiang, X., “The impact of vendor customizations on android security”, In Proc. of ACM conf. on Computer & communications security, 2014, pp. 623-634.

[9] Krutz, Daniel E., Mehdi Mirakhorli, Samuel A. Malachowsky, Andres Ruiz, Jacob Peterson, Andrew Filipski, and Jared Smith. "A dataset of open-source Android applications." In IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 522-525.

[10] Avdiienko, Vitalii, et al. "Mining apps for abnormal usage of sensitive data." Proc. of the 37th Int. Conf. on Software Engineering-Volume 1. IEEE Press, 2015.

[11] Maiorca, Davide, et al. "Stealth attacks: An extended insight into the obfuscation effects on android malware." Computers & Security 51 (2015): 16-31.

[12] Vidas, Timothy, and Nicolas Christin. "Sweetening android lemon markets: measuring and combating malware in application marketplaces." Proceedings of the third ACM conference on Data and application security and privacy. ACM, 2013.

[13] Maggi, Federico, Andrea Valdi, and Stefano Zanero. "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. ACM, 2013.

[14] Penning, Nicholas, et al. "Mobile malware security challeges and cloud-based detection." Int. IEEE Conf. on Collaboration Technologies and Systems, 2014.

[15] Faruki, Parvez, et al. "Android security: a survey of issues, malware penetration, and defenses." IEEE communications surveys & tutorials 17.2 (2015): 998-1022.

[16] Maier, Dominik, Tilo Müller, and Mykola Protsenko. "Divide-and-conquer: Why android malware cannot be stopped." Availability, Reliability and Security (ARES), 2014 Ninth International Conference on. IEEE, 2014.

[17] Sheen, Shina, R. Anitha, and V. Natarajan. "Android based malware detection using a multifeature collaborative decision fusion approach." Neurocomputing 151 (2015): 905-912.

[18] Feizollah, Ali, et al. "A review on feature selection in mobile malware detection." Digital Investigation 13 (2015): 22-37.

[19] Felt, A. Porter, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," In Proc. of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, pp. 3-14, 2011.

[20] Zhou, Yajin, and X. Jiang, "Dissecting android malware: Characterization and evolution," In IEEE Symposium on Security and Privacy, pp. 95-109, 2012.

[21] Spreitzenbarth, Michael, et al. "Mobile-sandbox: having a deeper look into android applications." Proceedings of the 28th Annual ACM Symposium on Applied Computing. ACM, 2013.

[22] Suarez-Tangil, Guillermo, et al. "Evolution, detection and analysis of malware for smart devices." IEEE Communications Surveys & Tutorials 16.2 (2014): 961-987.

[23] Liu, Xing, and Jiqiang Liu. "A two-layered permission-based Android malware detection scheme." Mobile cloud computing, services, and engineering (mobilecloud), 2014 2nd ieee international conference on. IEEE, 2014.

[24] Fedler, Rafael, Julian Schütte, and Marcel Kulicke. "On the effectiveness of malware protection on android." Fraunhofer AISEC 45 (2013).

[25] Yuan, Zhenlong, et al. "Droid-Sec: deep learning in android malware detection." ACM SIGCOMM Computer Communication Review. Vol. 44. No. 4. ACM, 2014.

[26] Benats, G., Bandara, A., Yu, Y., Colin, J. N., & Nuseibeh, B., "PrimAndroid: privacy policy modelling and analysis for android applications." In IEEE Int. Symp. on Policies for Distributed Systems and Networks, 2011, pp. 129-132.

[27] Sanz, Borja, et al. "Instance-based anomaly method for android malware detection." Security and Cryptography (SECRYPT), 2013 International Conference on. IEEE, 2013.

[28] Li, Li, et al. "Potential component leaks in Android apps: An investigation into a new feature set for malware detection." Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on. IEEE, 2015.

[29] M. Zheng, P. P. Lee, and J. C. Lui, “ADAM: an automatic and extensible platform to stress test android anti-virus system,” in Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), 2012, pp. 82–101.

[30] Zonouz, S., Houmansadr, A., Berthier, R., Borisov, N., and Sanders, W. (2013). Secloud: A cloud-based comprehensive and lightweight security solution for smartphones. Computers & Security.

[31] Suarez-Tangil, G., Lombardi, F., Tapiador, J. E., and Pietro, R. D. (2014a). Thwarting obfuscated malware via differential fault analysis. IEEE Computer, 47(6):24–31.

[32] Suarez-Tangil, G., Tapiador, J. E., Peris, P., and Ribagorda, A. (2014b). Evolution, detection and analysis of malware for smart devices. IEEE Communications Surveys & Tutorials, 16(2):961–987.

[33] Sanz, Borja, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. "Puma: Permission usage to detect malware in android." In International Joint Conference CISIS’12-ICEUTE´ 12-SOCO´ 12 Special Sessions, pp. 289-298. Springer Berlin Heidelberg, 2013.

[34] http://developer.android.com/reference/.

[35] Wei, Xuetao, L. Gomez, I. Neamtiu, and M.Faloutsos. "Permission evolution in the android ecosystem." In 28th Computer Security Applications Conf., pp. 31-40, 2012.

[36] Timothy Vidas and Nicolas Christin. Evading Android runtime analysis via sandbox detection. In Proceedings of the 9th ACM

symposium on Information, computer and communications security, pp 447–458, ACM, 2014.

[37] Zhang, Yuan, M. Yang, B. Xu, Z. Yang, G.Gu, P.Ning, X. S. Wang, and B.Zang, "Vetting undesirable behaviors in android apps with permission use analysis," In Proc. of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 611-622, 2013.

[38] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D.Wagner, “Android permissions:user attention, comprehension, and behavior," in Proc. of the 8th ACM Symp. on usable Privacy and Security, 2012, pp. 3:1-3:14.

[39] A. P. Felt, K. Greenwood, and D. Wagner, “The effectiveness of application permissions," in Proc. of the 2nd USENIX conf. on Web application development, 2011, pp. 7-14.

[40] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, “Android permissions: a perspective combining risks and benefits," in Proc. of 17th ACM symp. on Access Control Models and Technologies, 2012, pp. 13-22.

[41] Wijesekera, Primal, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. "Android Permissions Remystified: A Field Study on Contextual Integrity." In USENIX Security Symposium, pp. 499-514. 2015.

[42] Benton, Kevin, L. Jean Camp, and Vaibhav Garg. "Studying the effectiveness of android application permissions requests." IEEE Int. Conf. on IEEE Pervasive Computing and Communications Workshops, 2013.

[43] Armando, A., Carbone, R., Costa, G., & Merlo, A., “Android permissions unleashed”. In 28th IEEE Computer Security Foundations Symposium, 2015, pp. 320-333.

[44] Peiravian, Naser, and Xingquan Zhu. "Machine Learning for Android Malware Detection Using Permission and API Calls," In International Conf. on Tools with Artificial Intelligence (ICTAI), pp. 300-305. IEEE, 2013.

[45] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, “Droidmat: Android malware detection through manifest and API calls tracing,” in Proc. of Asia Joint Conf. on Information Security, 2012, pp. 62–69.

[46] Shekhar, Shashi, Michael Dietz, and Dan S. Wallach. "AdSplit: Separating Smartphone Advertising from Applications." In USENIX Security Symposium, pp. 553-567. 2012.

[47] Gibler, Clint, Jonathan Crussell, Jeremy Erickson, and Hao Chen, “AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale”, Springer Berlin Heidelberg, 2012.

[48] Feizollah A, Anuar NB, Salleh R, Suarez-Tangil G, Furnell S. “AndroDialysis: analysis of android intent effectiveness in malware detection”, computers & security. 2017, 31;65:121-34.

[49] Yang, Zhemin, et al. "Appintent: Analyzing sensitive data transmission in android for privacy leakage detection." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013.

[50] Ye H, Cheng S, Zhang L, Jiang F. “Droidfuzzer: Fuzzing the android apps with intent-filter tag”, In Proc. of Int. ACM Conf. on Advances in Mobile Computing & Multimedia, 2013, p. 68.

[51] Sasnauskas, R., & Regehr, “Intent fuzzer: crafting intents of death”. In Proc. Int. ACM Workshop on Dynamic Analysis and Software and System Performance Testing, Debugging, and Analytics, pp. 1-5.

[52] Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., & Wang, X., “Appintent: Analyzing sensitive data transmission in android for privacy leakage detection”, In Proc. of ACM SIGSAC conf. on Computer & communications security pp. 1043-1054.

[53] V. Rastogi, Y. Chen, and W. Enck, “Appsplayground: automatic security analysis of smartphone applications,” in Proc. 3rd ACM conference on Data and application security and privacy. ACM, 2013, pp. 209–220.

[54] T. Vidas, N. Christin, and L. Cranor, “Curbing Android permission creep," in Proc.of the Web 2.0 Security and Privacy, May 2011.

[55] Loorak, M. H., Fong, P. W., & Carpendale, “Papilio: Visualizing android application permissions”. In Computer Graphics Forum Vol. 33, No. 3, pp. 391-400, 2014.

[56] S. Zonouz, A. Houmansadr, R. Berthier, N. Borisov, and W. Sanders, “Secloud: A cloud-based comprehensive and lightweight security solution for smartphones,” Computers & Security, 2013.

[57] Sato, Ryo, Daiki Chiba, and Shigeki Goto. "Detecting Android malware by analyzing manifest files." Proceedings of the Asia-Pacific Advanced Network36 (2013): 23-31.

[58] Rastogi, Vaibhav, Yan Chen, and William Enck. "AppsPlayground: automatic security analysis of smartphone applications." Proceedings of the third ACM conference on Data and application security and privacy. ACM, 2013.

[59] Zhou, Wu, Yajin Zhou, Xuxian Jiang, and Peng Ning. "Detecting repackaged smartphone applications in third-party android marketplaces." In Proceedings of the second ACM conference on Data and Application Security and Privacy, pp. 317-326. ACM, 2012.

[60] Faruki, Parvez, Vijay Ganmoor, Vijay Laxmi, Manoj Singh Gaur, and Ammar Bharmal. "AndroSimilar: robust statistical feature signature for Android malware detection." In Proceedings of the 6th International Conference on Security of Information and Networks, pp. 152-159. ACM, 2013.

[61] Gibler, Clint, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. "Adrob: Examining the landscape and impact of android application plagiarism." In Proceeding of the 11th annual international conference on Mobile systems, applications, and services, pp. 431-444. ACM, 2013.

[62] Enck, William, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones."Communications of the ACM 57, no. 3 (2014): 99-106.

[63] Yang, Chao, et al. "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android

applications." Computer Security-ESORICS 2014. Springer International Publishing, 2014. pp.163-182.

[64] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. B. Alis, “Dendroid: A text mining approach to analyzing and classifying code structures in android malware families,” Expert Systems with Applications, 2013, in Press.

[65] Feng, Yu, S.Anand, I.Dillig, and A. Aiken, "Apposcopy: Semantics-Based Detection of Android Malware," In submission.

[66] Zhang, Yuan, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. "Vetting undesirable behaviors in android apps with permission use analysis." In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 611-622. ACM, 2013.

[67] S. Rosen, Z. Qian, and Z. M. Mao, “Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users,” in Proc. 3rd ACM conference on Data and application security and privacy. ACM, 2013, pp. 221–232.

[68] Octeau, Damien, P. McDaniel, S.Jha, A.Bartel, E.Bodden, J. Klein, and Y. L.Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," In Proc. of the 22nd USENIX Security Symposium, 2013.

[69] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, “Systematic detection of capability leaks in stock android smartphones,” in Proc. 19th Annu. Symp. on Network and Distributed System Security, 2012.

[70] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in Proc. Network and Distributed System Security Symp., 2011.

[71] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in Proc. 10th int. conf. on Mobile systems, applications, and services. ACM, 2012, pp. 281–294.

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

[73] Rastogi, Vaibhav, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," In 8th ACM SIGSAC symp.on Information, computer and communications security, pp. 329-334, 2013.

[74] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""andromaly": a behavioral malware detection framework for android devices," J. of Intelligent Information Systems, vol. 38, pp. 161–190, 2012.

[75] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. Styp-Rekowsky, "Appguard —-real–time policy enforcement for thirdparty applications," Universitats- und Landesbibliothek, Postfach 151141, 66041 Saarbracken, Tech. Rep., 2012.

[76] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: a multi-level anomaly detector for android malware," in Proc. 6thint. conf. on

Mathematical Methods, Models and Architectures for Computer Network Security: computer network security, ser. MMMACNS' 12. Springer-Verlag, 2012, pp. 240–253.

[77] L. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in Proc. 21st USENIX conf. on Security symp. USENIX Association, 2012, pp. 29–29.

[78] Au, Kathy Wain Yee, Yi Fan Zhou, Zhen Huang, and David Lie. "Pscout: analyzing the android permission specification." In Proceedings of the 2012 ACM conference on Computer and communications security, pp. 217-228. ACM, 2012.

[79] Bugiel, Sven, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. "Xmandroid: A new android evolution to mitigate privilege escalation attacks." Technische Universität Darmstadt, Technical Report TR-2011-04 (2011).

[80] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in Proc. 19th Annu. Symp. on Network and Distributed System Security, 2012.

[81] Enck, William, Machigar Ongtang, and Patrick McDaniel. "On lightweight mobile phone application certification." In Proceedings of the 16th ACM conference on Computer and communications security, pp. 235-245. ACM, 2009.

[82] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang and Binyu Zang. Vetting undesirable behaviors

in Android apps with permission use analysis. Computer & communications security, pp 611–622, ACM, 2013.

[83] Jeon, Jinseong, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. "Dr. android and mr. hide: fine-grained permissions in android applications." In Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices, pp. 3-14. ACM, 2012.

[84] Chin, Erika, A. Porter, Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," In Proc. of the 9th international conf. on Mobile systems, applications, and services, pp. 239-252, 2011.

[85] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “Chex: statically vetting android apps for component hijacking vulnerabilities,” in Proc. 2012 ACM conf. on Computer and communications security. ACM, 2012, pp. 229–240.

[86] Octeau, Damien, P. McDaniel, S.Jha, A.Bartel, E.Bodden, J. Klein, and Y. L.Traon, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," In Proc. of the 22nd USENIX Security Symp., 2013.

[87] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, “Crowdroid: behaviorbased malware detection system for android,” in 1st ACM workshop on Security and privacy in smartphones and mobile devices. ACM, 2011, pp. 15–26.

[88] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, “Paranoid android: versatile protection for smartphones,” in Proc. 26th Annu.Computer Security Applications Conf., 2010, pp. 347–356.

[89] Marforio, Claudio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. "Analysis of the communication between colluding applications on modern smartphones." In Proceedings of the 28th Annual Computer Security Applications Conference, pp. 51-60. ACM, 2012.

[90] Marforio, Claudio, and Aurélien Francillon. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Department of Computer Science, ETH Zurich, 2011.

[91] Portokalidis G., Homburg P., Anagnostakis K., Bos H. “Paranoid android: versatile protection for smartphones,” In: Proc. of 26th Annual Computer Security Applications Conference, pp. 347-356, 2010.

[92] Chun B.G., Ihm S., Maniatis P., Naik M., Patti A., “Clonecloud: elastic execution between mobile device and cloud,” In: Proc. of the 6th conf. on Computer systems, pp. 301-314, 2011.

[93] Kabakus Abdullah Talha, Dogru Ibrahim Alper and Cetin Aydin, “APK Auditor: Permission-based Android malware detection system”, Digital Investigations, Elsevier Journal on, PP0 (13): pp 1–14, Elsevier, 2015.

[94] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang and David Lie. Pscout: analyzing the Android permission specification. Proceedings of conference on Computer and communications security, pp 217–228, ACM, 2012.

[95] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee and Guofei Jiang. Chex: statically vetting Android apps for component hijacking vulnerabilities. Proceedings of conference on Computer and communications security, pp 229–240, ACM, 2012.

[96] Davi, Dmitrienko, Sadeghi, and Winandy] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In Information Security, pp 346–360, Springer, 2011.

[97] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen and Martin C Rinard. Information Flow Analysis of Android Applications in DroidSafe. pp 1–16, NDSS, 2015.

[98] Faruki, Bharmal, Laxmi, Ganmoor, Gaur, Conti, and Rajarajan] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Gaur, Mauro Conti and Raj Muttukrishnan. Android security: A survey of issues, malware penetration and defenses. Communications Surveys & Tutorials, 170 (2): pp 998–1022, IEEE, 2014.

[99] William Enck, Machigar Ongtang and Patrick McDaniel. On lightweight mobile phone application certification. Computer and communications security, pp 235–245, ACM, 2009.

[100] Tenenboim-Chekina, L., Barad, O., Shabtai, A., Mimran, D., Rokach, L., Shapira, and Elovici, Y. (2013). Detecting application update attack on mobile devices through network featur. In 2013 IEEE Conference on Computer Communications Workshops, pp. 91-92.

[101] Wei, X., Gomez, L., Neamtiu, I., and Faloutsos, M. (2012). Permission evolution in the android ecosystem. In Proc. of the 28th ACM Annual Computer Security Applications Conference, pages 31-40.

[102] L.-K. Yan and H. Yin, “Droidscope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis,” in Proc. of USENIX Security Symposium, 2012.

[103] Octeau, Damien, et al. "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis." (2013).

[104] Henry B Mann and Donald R Whitney, “On a test of whether one of two random variables is stochastically larger than the other”, the annals of mathematical statistics, pp 50-60, 1947.

[105] Jacob Cohen. A power primer. Psychological bulletin, 1120 (1): pp 155, 1992.

[106] Zhang, Fangfang, et al. "ViewDroid: Towards obfuscation-resilient mobile application repackaging detection." Proc. of ACM conf. on Security and privacy in wireless & mobile networks. ACM, 2014.

[107] Tam, Kimberly, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. "CopperDroid: Automatic Reconstruction of Android Malware Behaviors." In NDSS. 2015.

[108] Geneiatakis, Dimitris, Igor Nai Fovino, Ioannis Kounelis, and Paquale Stirparo. "A Permission verification approach for android mobile applications." Computers & Security 49 (2015): 192-205.

[109] Bagheri, H., Sadeghi, A., Garcia, J., & Malek, S. (2015). Covert: Compositional analysis of android inter-app permission leakage. IEEE transactions on Software Engineering, 41(9), 866-886.

[110] Klieber, W., Flynn, L., Bhosale, A., Jia, L., & Bauer, L., " Android taint flow analysis for app sets", In Proc. of the 3rd ACM Int. Workshop on the State of the Art in Java Program Analysis, pp. 1-6.

[111] Chan, P. P., Hui, L. C., & Yiu, S. M., "Droidchecker: analyzing android applications for capability leak", In Proc. of the 5th ACM conf. on Security and Privacy in Wireless and Mobile Networks, pp. 125-136.

[112] Wei, F., Roy, S., & Ou, X., "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps", In Proc. of ACM SIGSAC Conference on Computer and Communications Security, pp. 1329-1341.

[113] Jo, M. J., & Shin, J. S., "Study on Security Vulnerabilities of Implicit Intents in Android", Journal of the Korea Institute of Information Security and Cryptology, 24(6), 2014, 1175-1184.

[114] Yang T, Yang Y, Qian K, Lo DC, Qian Y, Tao L., "Automated detection and analysis for android ransomware", In 12th Int. Conf. on Embedded Software and Systems, 2015, pp. 1338-1343.

[115] Gordon MI, Kim D, Perkins JH, Gilham L, Nguyen N, Rinard MC. "Information Flow Analysis of Android Applications in DroidSafe", InNDSS 2015.

[116] Li L, Bartel A, Bissyandé TF, Klein J, Le Traon Y., "Apkcombiner: Combining multiple android apps to support inter-app analysis", In IFIP Int. Information Security Conf., 2015, pp. 513-527.

[117] Zheng, Min, Mingshen Sun, and John CS Lui. "Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware", 12th IEEE Int. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom), 2013.

[118] Aung, Zarni, and Win Zaw. "Permission-based android malware detection." International Journal of Scientific & Technology Research 2.3 (2013): 228-234.

[119] Elish, Karim O., et al. "Profiling user-trigger dependence for Android malware detection." Computers & Security 49 (2015): 255-273.

[120] Sanz, Borja, et al. "Puma: Permission usage to detect malware in android." International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions. Springer Berlin Heidelberg, 2013.

[121] Sanz, Borja, et al. "MAMA: manifest analysis for malware detection in android." Cybernetics and Systems 44.6-7 (2013): 469-488.

[122] Zhang, Yuan, et al. "Vetting undesirable behaviors in android apps with permission use analysis." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013.

[123] Samra, Aiman A. Abu, Kangbin Yim, and Osama A. Ghanem. "Analysis of clustering technique in android malware detection." Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2013 Seventh International Conference on. IEEE, 2013.

[124] Huang, Chun-Ying, Yi-Ting Tsai, and Chung-Han Hsu. "Performance evaluation on permission-based detection for android malware." Advances in Intelligent Systems and Applications-Volume 2. Springer, Berlin, Heidelberg, 2013. 111-120.

[125] Cen, Lei, et al. "A probabilistic discriminative model for android malware detection with decompiled source code." IEEE Transactions on Dependable and Secure Computing 12.4 (2015): 400-412.

2014.

[126] Li, Li, et al. "Iccta: Detecting inter-component privacy leaks in android apps." Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, 2015.

[127] Liang, Shuang, and Xiaojiang Du. "Permission-combination-based scheme for android mobile malware detection." Communications (ICC), 2014 IEEE International Conference on. IEEE, 2014.

[128] Zhongyang, Yibing, et al. "DroidAlarm: an all-sided static analysis tool for Android privilege-escalation malware." Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. ACM, 2013.

[129] Yerima, Suleiman Y., Sakir Sezer, and Igor Muttik. "High accuracy android malware detection using ensemble learning." IET Information Security 9.6 (2015): 313-320.

[130] Sarwar, Golam, et al. "On the Effectiveness of Dynamic Taint Analysis for Protecting against Private Information Leaks on Android-based Devices." SECRYPT. 2013.

[131] Ping, Xiong, et al. "Android malware detection with contrasting permission patterns." China Communications 11.8 (2014): 1-14.

[132] Xiaoyan, Zhao, Fang Juan, and Wang Xiujuan. "Android malware detection based on permissions." (2014): 2-063.

[133] Sanz, Borja, et al. "Mads: malicious android applications detection through string analysis." International Conference on Network and System Security. Springer, Berlin, Heidelberg, 2013.

[134] Wang, Wei, et al. "Exploring permission-induced risk in android applications for malicious application detection." IEEE Transactions on Information Forensics and Security 9.11 (2014): 1869-1882.

[135] Ham, Hyo-Sik, and Mi-Jung Choi. "Analysis of android malware detection performance using machine learning classifiers." ICT Convergence (ICTC), 2013 International Conference on. IEEE, 2013.

[136] Chuang, H. Y., & Wang, S. D., "Machine learning based hybrid behavior models for Android malware analysis". In IEEE Int. Conf. on Software Quality, Reliability and Security (QRS), 2015, pp. 201-206.

[137] Mas' ud, Mohd Zaki, et al. "Analysis of features selection and machine learning classifier in android malware detection." Information Science and Applications (ICISA), 2014 International Conference

[138] Allix, Kevin, et al. "Machine Learning-Based Malware Detection for Android Applications: History Matters!." University of Luxembourg, SnT, 2014.

[139] Yerima, Suleiman Y., Sakir Sezer, and Igor Muttik. "Android malware detection using parallel machine learning classifiers." Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on. IEEE, 2014.

[140] Narudin, Fairuz Amalina, et al. "Evaluation of machine learning classifiers for mobile malware detection." Soft Comput. 20.1 (2016): 343-357.

[141] Canfora, Gerardo, Francesco Mercaldo, and Corrado Aaron Visaggio. "A classifier of malicious android applications." 8th Int. IEEE Conf. on Availability, Reliability and Security, 2013.

[142] Yerima, S. Y., Sezer, S., McWilliams, G., and Muttik, I. (2013). A new android malware detection approach using bayesian classification. In 2013 IEEE 27th Int. Conf. on Advanced Information Networking and Applications, pp.121-128.

[143] Kang, Jang, Mohaisen, and Kim] Hyunjae Kang, Jae-wook Jang, Aziz Mohaisen, and Huy Kang Kim. Detecting and classifying Android malware using static analysis along with creator information. International Journal of Distributed Sensor Networks, pp 1–9, 2015.

[144] Samra, A.A.A., K. Yim ; Ghanem and O.A, “Analysis of Clustering Technique in Android Malware Detection,” Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2013, pp. 729 – 733.

[145] Glodek, W. and Harang, R., “Rapid Permissions-Based Detection and Analysis of Mobile Malware Using Random Decision Forests,” Military Communications Conference, MILCOM, pp. 980 – 985.

[146] Canfora, G., Mercaldo, F. and Visaggio, C.A. “A Classifier of Malicious Android Applications,” Availability, Reliability and Security (ARES), 2013.

[147] Suarez-Tangil, Guillermo, et al. "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families." Expert Systems with Applications 41.4 (2014): 1104-1117.

[148] Yerima SY, Sezer S, McWilliams G, Muttik I. “A new android malware detection approach using bayesian classification”, 27th IEEE Int. Conf. on In Advanced Information Networking and Applications, 2013, pp. 121-128.

[149] Feldman, Stephen, Dillon Stadther, and Bing Wang. "Manilyzer: automated android malware detection through manifest analysis." Mobile Ad Hoc and Sensor Systems (MASS), 2014 IEEE 11th International Conference on. IEEE, 2014.

[150] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In Proc. of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS ’10.

[151] Gartner says worldwide sales of mobile phones of first quarter of 2017." http://www.gartner.com/newsroom/id/3725117, November 2017.

[152] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, \A methodology for empirical analysis of permission-based security models and its application to android," in Proc. of the 17th ACM conference on Computer and communications security, pp. 73-84, 2010.

[153] Peiravian, N., & Zhu, X., “Machine learning for android malware detection using permission and api calls”. In 25th IEEE Int. Conf. on Tools with Artificial Intelligence, 2013, pp. 300-305.

[154] Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K, Siemens CE. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. InNDSS 2014 Feb 23.

[155] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee and Kuo-Ping Wu, “DroidMat: Android Malware Detection through Manifest

and API Calls Tracing," Information Security (Asia JCIS), 2012, pp. 62 – 69.

[156] Martina Lindorfer, Matthias Neugschwandtner and Christian Platzer. MARVIN: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis. Computer Software and Applications, 39th Annual Conference, pp 422–433, IEEE, 2015.

[157] Dietterich, Thomas G. "Ensemble methods in machine learning." In Multiple classifier systems, pp. 1-15. Springer Berlin Heidelberg, 2000.

[158] Drucker, Harris, Corinna Cortes, Lawrence D. Jackel, Yann LeCun, and Vladimir Vapnik. "Boosting and other methods." Neural Computation 6, no. 6 (1994): 1289-1301.

[159] Bennett, Kristin P., Ayhan Demiriz, and Richard Maclin. "Exploiting unlabeled data in ensemble methods." In Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 289-296. ACM, 2002.

[160] Thomas G Dietterich. Ensemble methods in machine learning. In Multiple classifier systems, pp 1–15, Springer, 2000.

[161] Chen, Sen, et al. "Stormdroid: A streaminglized machine learning-based system for detecting android malware." Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. ACM, 2016.

[162] Karim O Elish, Danfeng Yao and Barbara G Ryder, (2015) On the Need of Precise Inter-App ICC Classification for Detecting Android Malware Collusions. Proc. Of IEEE Mobile Security Technologies.

[163] Claudio Marforio, Hubert Ritzdorf, Aurelien Francillon, Srdjan Capkun (2012) Analysis of the communication between colluding applications on modern smartphones. Proc. of the 28th Annual Computer Security Applications Conference. pp. 51-60.

[164] Erika Chin, Adrienne Porter Felt, Kate Greenwood, David Wagner (2011) Analyzing inter-application communication in Android. Proc. of the 9th ACM conf. on Mobile systems, applications and services. pp.239-252.

[165] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, Bhargava Shastry (2012) Towards Taming Privilege-Escalation Attacks on Android. NDSS

[166] Fauzia Idrees, Muttukrishnan Rajarajan (2014) Investigating the android intents and permissions for malware detection. Proc. of IEEE Wireless and Mobile Computing, Networking and Communications. pp. 354-358.

[167] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steven Hanna, Erika Chin (2011) Permission Re-Delegation: Attacks and Defenses. USENIX Security Symposium.

[168] Fauzia Idrees, Muttukrishnan Rajarajantitle, Mauro Conti, Thomas M. Chen and Rahulamathavan Yogachandran (2017) PIndroid: A novel Android malware detection system using ensemble learning methods. Computers & Security. vol. 68, Elsevier, pp.36-46.

[169] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, Sam Malek (2015) Covert: Compositional analysis of android inter-app permission leakage. IEEE Transactions on Software Engineering no. 9, pp. 866-886.

[170] Wade Gasior, Li Yang (2011) Network covert channels on the Android platform. Proc.of the Seventh Annual ACM Workshop on Cyber Security and Information Intelligence Research, pp. 61-67.

[171] Davi, L., Dmitrienko, A., Sadeghi, A. R., Winandy, M. (2010) Privilege escalation attacks on android. In Int. Conf. on Information Security. pp. 346-360.

[172] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi (2011) Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical Report:Technische University at Darmstadt.

[173] Atif M Memon, Ali Anwar (2015) Colluding Apps: Tomorrow's Mobile Malware Threat. IEEE Security & Privacy, no. 6, pp. 77-81.

[174] Shweta Bhandari, Vijay Laxmi, Akka Zemmari, Manoj Singh Gaur (2016) Intersection automata based model for Android application collusion. Advanced Information Networking and Applications. pp. 901-908.

[175] Irina Asavoaeca, Blasco Jorge, Thomas Chen, Harsha Kumara , Igor Muttik, Hoang Nga Nguyen, Markus Roggenbach, Siraj Shaikh (2016) Towards Automated Android App Collusion Detection. arXiv preprint arXiv:1603.02308.

[176] Ravitch Tristan, Creswick E Rogan, Tomb Aaron, Foltzer Adam, Elliott Trevor, Casburn Ledah (2014) Multi-app security analysis with fuse: Statically detecting android app collusion. Proc. of the 4th Program Protection and Reverse Engineering Workshop. pp. 4.

[177] W. , P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, A. N. Sheth (2010) TaintDroid: an information flow tracking system for realtime privacy monitoring on smartphones. Proc. of the 9th USENIX Conf on Operating Systems Design and Implementation. (OSDI'10), pp. 1-6.

[178] William Enck, Machigar Ongtang, Patrick McDaniel (2009) Understanding Android Security. IEEE Security and Privacy. 7:50-57.

[179] David Kantola, Erika Chin,Warren He, DavidWagner (2012) Reducing attack surfaces for intra-application communication in Android. Proc. of second ACM workshop on Security and privacy in smartphones and mobile devices. pp. 69-80.

[180] Amiya Maji, Fahad Arshad, Saurabh Bagchi, Jan Rellermeyer (2012) An empirical study of the robustness of inter-component communication in Android. Int. Conf. on Dependable Systems and Networks. pp. 1-12.

[181] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, Guofei Jiang (2012) Chex: statically vetting Android apps for component hijacking vulnerabilities. Proc. of conf. on Computer and communications security. pp. 229-240.

[182] Andrea Avancini, Mariano Ceccato (2013) Security testing of the communication among Android applications. Proc. of 8th IEEE International Workshop on Automation of Software Test. pp. 57-63.

[183] Michael I Gordon, Deokhwan Kim, Je H Perkins, Limei Gilham, Nguyen Nguyen, Martin C Rinard (2015) Information Flow Analysis of Android Applications in DroidSafe. NDSS. pp. 1-16.

[184] Daniele Gallingani, Rigel Gjomemo, VN Venkatakrishnan, Stefano Zanero, “Practical Exploit Generation for Intent Message Vulnerabilities in Android.” Proc. of the 5th ACM Conf. on Data and application security, 2015, pp. 155-157.

[185] Xu K, Li Y, Deng RH., “ICCDetector: ICC-based malware detection on Android”, IEEE Transactions on Information Forensics and Security, 2016, 11(6):1252-64.

[186] Martina Lindorfer, Matthias Neugschwandtner and Christian Platzer, “MARVIN:Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis”, Computer Software and Applications Conf., pp. 422-433, IEEE, 2015.

[187] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon and Konrad Rieck, “DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket”, NDSS, 2014.

[188] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu, “Droidmat: Android malware detection through manifest and api calls tracing”, Information Security (Asia JCIS), pp. 62-69. IEEE, 2012.

[189] McLaughlin, Jesus M., BooJoong K., Suleiman Y., Paul M., Sakir S., Yeganeh S., "Deep Android Malware Detection", Proc. Seventh Conf. Data and Application Security and Privacy, pp. 301-308. ACM, 2017.