Universität Bremen

A Malware Detection System For Android

by

M.Sc. Franklin Tchakounté

Thesis

Submitted in partial Fulfillment of the

Requirements for the Degree of Dr.-Ing.

Fachbereich Mathematik / Informatik

Universität Bremen

July 2015

Supervisors:
Prof. Dr. Karl-Heinz Rödiger, University of Bremen
Prof. Dr. David Békollé, University of Ngaoundéré

Acknowledgements

On completion of my Ph.D. thesis I would like to sincerely thank all those who supported me in realising and finishing my work.

I am heartily thankful to my supervisor Prof. Dr.-Ing. Karl-Heinz Rödiger spending time and effort on me. Throughout all the stages of my thesis, he helped me to keep track on the right research direction, patiently discussed and seriously revised all my work

I would like to honestly thank Prof. Dr. David Békollé for his permanent assistance in my research career and for for acting as an expert in writing a report about my thesis.

I thank Dr. Jean Michel Nlong who is my research mentor at the University of Ngaoundéré since I was a Master student. I received lot of suggestions about the evolution of my work in his group.

My deepest thanks also to François Hayata for providing ideas and technical assistance.

I would like to send my gratitude to the people involved in the establishment of the cooperation between the universities of Ngaoundéré and Bremen through the DAAD Programme on Subject-Related Partnerships with Universities in Developing Countries. This project and the BMBF project "Qualifying for an occupational perspective as software engineer - A contribution to the improvement of the teaching situation in Computer Science in Cameroon" helped me to travel abroad to work with my supervisor.

Last not least, I would like to express my deep-felt gratitude to my parents and my family. Thank you for giving me so much support.

Abstract

Android security is built upon a permission-based mechanism, which restricts access of third-party Android applications to critical resources on an Android device. The user must accept the set of permissions that an application requires, before the installation proceeds. This is to inform the user about the risks of installing and using an application. It has two problems. The first one is that users are not aware enough of existing threats and trust either the application store or the popularity of the application and accept the installation without analysing the intentions of the developer. The second one is that Android does not display the specific resource needed by the application and the corresponding permissions during its installation. It rather presents different categories representing the set of resources with a description. The categories include implicitly permissions necessary to access some resources. The user grants more authorisations than necessary probably confused by the management of permissions, increasing the difficulty of detecting malicious applications and constituting the basis for many attacks.

The thesis defines a system for detecting Android malware based only on requested permissions. It focuses on 222 permissions including some exclusively for third-party applications. It is a static analysis technique, which combines two reliable strategies. The the first one focuses on the discriminating metric based on the frequency of permissions and the proportion of requests by malicious applications within the whole sample. The second one relies on security risks related to granting permissions. A comparison has shown that the four protection levels of permissions defined by Google are coarse-grained, hiding the real sense of permissions. The first strategy is fine-grained and more precise in terms of permission semantics. We collected a dataset with 6783 malicious and 1993 normal applications, which have been tested and validated. Profiles for each sample have been generated, depending on both strategies and used as input for training and learning processes. Seven classifiers have been applied to the models to output performance results. We select the good ones to define our classifier, which provides outstanding performance in detection and prediction. A dataset of associations of permissions to weights that can be reused in a different research has been released from our work.

Evaluations indicate that our model is one of the best tools with only requested permissions as a feature. It is able to detect around 99.20% of 1260 cases of malware released by the Genome project, which represents behaviour of nowadays malware. This work provides a scheme for weighting permissions possibly applicable to an unknown samples dataset, while keeping a good performance in classification. The model is good in detecting Android malware with around 97% of the True Positive Rate and predicting Android malware with around 95% of the True Positive Rate. This means that it is capable to discriminate almost all cases of malware in detection and prediction. The Area Under Curve (AUC) metric is between 97% and 99%, which confers the outsatnding property of the outstanding detection system for the detection of malware.

We propose additionally a system that can be embedded into an Android hand-held device for real-time detection. The results of the comparison to three renowned antiviruses reveal that our framework clearly outperforms two of them.

Zusammenfassung

Die Sicherheit von Android baut auf einen berechtigungsbasierten Mechanismus, der Dritt-Anwendungen den Zugang zu kritischen Ressourcen auf einem mit diesem Betriebssystem ausgestatteten Gerät beschränkt. Benutzer müssen etliche Berechtigungen akzeptieren, die eine Anwendung benötigt, bevor die Installation fortgesetzt wird. Damit sollen Benutzer auf das Risiko der Installation und Verwendung einer Applikation hingewiesen werden. Dies wirft zwei Probleme auf: Erstens sind sich Benutzer nicht ausreichend der Gefährlichkeit bewusst und vertrauen entweder dem App-Markt oder der Popularität der Anwendung; sie akzeptieren die Installation, ohne die Intentionen des Entwicklers zu analysieren. Zweitens zeigt Android nicht die bestimmte Ressource mit den entsprechenden Berechtigungen an, die bei der Installation benötigt werden. Stattdessen präsentiert Android dem Benutzer verschiedene Kategorien von Ressourcen mit Beschreibungen. Implizit enthalten sie auch diejenigen Berechtigungen, die für den Zugang zu einigen Ressourcen notwendig sind. Benutzer, verwirrt durch das Berechtigungs-Management, vergeben dann mehr Berechtigungen als nötig. Damit vergrößern sie die Schwierigkeit, schädliche Anwendungen zu erkennen, und schaffen somit die Grundlage für viele Angriffe.

In dieser Arbeit wird ein Modell zur Erkennung von Malware unter Android entwickelt, das nur auf erforderlichen Berechtigungen basiert. Es konzentriert sich auf 222 Berechtigungen darunter auch einige, die ausschließlich für Dritt-Anwendungen bestimmt sind. Es basiert auf einer statische Analysetechnik, die zwei zuverlässige Strategien kombiniert. Die erste fokussiert auf eine diskriminierende Metrik, die auf der Häufigkeit von Berechtigungen und der Anzahl der Anfragen durch schädliche Anwendungen innerhalb des gesamten Spektrums fußt. Die zweite Strategie setzt auf Sicherheitsrisiken, hervorgerufen durch die Gewährung von Berechtigungen. Die erste Strategie ist feinkörnig und bezogen auf die Semantik von Berechtigungen präziser. Ein Vergleich zeigt, dass die vier von Google definierten Berechtigungs-Schutzniveaus grobkörnig sind und den eigentlichen Sinn von Berechtigungen verbergen. Wir haben einen Satz von 6783 Schadprogrammen und 1993 normalen Anwendungen gesammelt, getestet und validiert. Für jede Probe werden in Abhängigkeit von beiden Modellen Profile generiert, die als Eingabe für Trainings- und Lernprozesse eingesetzt werden. Um Performanz-Ergebnisse zu erzielen, werden sieben Klassifikatoren auf die Modelle angewendet. Wir nehmen die besten, um damit unseren Klassifikator zu bestimmen, der herausragende Ergebnisse bez. Erkennung und Prognose von Malware erreicht. Ergebnis unserer Arbeit ist ein Satz Assoziationen von Berechtigungen mit Gewichten, die in weiteren Forschungsvorhaben genutzt werden können.

Evaluationen ergeben, dass unser Modell eines der besten Werkzeuge für Anwendungen mit angeforderten Berechtigungen ist. Es ist in der Lage, 99,2% von den 1260 vom Genome-Projekt zur Verfügung gestellten Malware-Fällen zu identifizieren, die das aktuelle Verhalten von Schadsoftware repräsentieren. Diese Arbeit stellt ein Schema bereit, mit dem Berechtigungen für noch unbekannte Beispiele gewichtet werden können. Das Modell ist mit rund 97% der True Positive Rate gut in der Erkennung von Android-Malware und mit rund 95% der True Positive Rate bei Prognosen. Das heißt, es ist in der Lage, beinahe alle Fälle von Malware zu erkennen und zu prognostizieren. Die Fläche unter der Kurve (AUC) liegt zwischen 97% und 99%; das Modell hat damit hervorragende Eigenschaften für die Erkennung von Malware.

Wir schlagen zusätzlich ein System vor, das zur Echtzeit-Erkennung auf ein mobiles Android-Gerät geladen werden kann. Ein Vergleich mit drei renommierten Anti-Virus-Systemen zeigt, dass unser Rahmenwerk zwei von ihnen übertrifft.

Contents

Contents iv

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Relevance for Africa/Cameroon . . . 3
1.2 Target and Motivation . . . 5
1.3 Overview of Malware Detection Approaches . . . 6
1.4 Problem Statement . . . 7
1.5 Main Requirements . . . 10
1.6 Research Hypotheses . . . 10
1.7 Objectives . . . 11
1.8 Thesis Structure . . . 12

2 Android Ecosystem 14

2.1 Architecture of Android . . . 14
2.1.1 Linux Kernel . . . 15
2.1.2 Libraries . . . 15
2.1.3 Android Runtime . . . 15
2.1.4 Application Framework . . . 15
2.1.5 Applications . . . 15
2.2 Android Application . . . 17
2.2.1 Application Structure and Reversing Techniques . . . 17
2.2.2 Composition . . . 20
2.2.3 Signature . . . 24
2.2.4 Development . . . 25
2.3 Android and Third Party Markets . . . 25
2.4 Android Security Landscape . . . 26
2.4.1 Security on Android . . . 26
2.4.2 Mapping Android Security Mechanisms to Android Layers . . . 29
2.4.3 Security from Google . . . 29
2.4.4 Limitations . . . 30

3 Android Malware Landscape 32
3.1 Malware Life Cycle . . . 32
3.2 Android Malware Threats and their Evolution . . . 33
3.2.1 Malware . . . 33
3.2.2 Spyware . . . 33
3.2.3 Grayware . . . 34
3.2.4 Fraudware . . . 34
3.2.5 Trojan . . . 34
3.2.6 Root exploit . . . 34
3.2.7 Bot . . . 35
3.2.8 Privilege Escalation Exploits . . . 35
3.3 Malware Techniques on Android . . . 35
3.3.1 Repackaging . . . 35
3.3.2 Update Attack . . . 36
3.3.3 Drive by Downloads . . . 36
3.3.4 Remote Control . . . 37
3.3.5 Premium Rate SMS . . . 37
3.3.6 Phishing Scams . . . 37
3.3.7 Browser Exploits . . . 38
3.3.8 Capability Leaking . . . 38
3.3.9 Information Leaking . . . 38
3.3.10 Threats Based on Vulnerabilities . . . 38
3.4 Actual Attacks on Android . . . 39
3.4.1 Privilege Escalation Attacks . . . 39
3.4.2 TOCTOU attack . . . 41
3.4.3 Attacks on Misconfigured Applications . . . 41
3.4.4 Attack on Applications Sharing the User ID . . . 42
3.4.5 UI Takeover Attack . . . 43
3.4.6 Zero Permission Attacks . . . 43
3.4.7 Reverse Shell Execution . . . 43
3.4.8 Unprivileged Attacks . . . 44
3.4.9 Physical Access with ADB Enabled . . . 44
3.4.10 Physical Access without ADB Enabled . . . 45
3.4.11 Physical Access to Unobstructed Device . . . 45
3.5 Families of Malware . . . 46
3.6 Types of Malware Analysis . . . 46
3.6.1 Static Analysis . . . 46
3.6.2 Manifest Analysis . . . 47
3.6.3 Dynamic Analysis . . . 47
3.6.4 Comparison between Static, Manifest, and Dynamic Analysis . . . 47
3.7 Tools for Malware Detection . . . 48
3.7.1 Firewall . . . 48

3.7.2 Intrusion Detection Systems 49
3.7.3 Antiviruses 49

4 Concepts on Machine Learning for Malware Detection 51
4.1 Machine Learning 51
4.1.1 Definition of Concepts 52
4.1.2 Evaluation Metrics for Performance 63
4.1.3 Performance Evaluation of a Classifier 66
4.2 Related Work: Machine Learning and Permissions 68
4.2.1 Permission Analysis 68
4.2.2 Characterisation and Classification of Applications 69
4.2.3 Discussion 75

5 A Three-Layered Malware Detection and Alerting System for Android (TLMDASA) 79
5.1 The Architecture of TLMDASA 80
5.2 Definitions 82
5.3 Static Analysis 83
5.3.1 Malware for the Learning Phase 83
5.3.2 Normal Applications for the Learning Phase 83
5.3.3 Applications for System Validation 84
5.3.4 Readjustment of the Normal Sample 84
5.3.5 Tools for the Extraction 85
5.4 Layer 1: Relying on Discriminating Metrics 86
5.4.1 First Strategy 86
5.4.2 Second Strategy 87
5.4.3 Translation in Vector Space 88
5.4.4 Distribution of DMs and Discussion 89
5.4.5 Comparison with the Google Protection Levels 90
5.5 Layer 2: Relying on the Risks for Sensitive Resources 92
5.5.1 Translation in Vector Space 96
5.5.2 Results and Discussion 97
5.6 Layer 3: The Combination of DM and Resource Risks 103
5.7 Learning Detection 104
5.7.1 Environment for Learning 104
5.7.2 Classifiers 104
5.7.3 Our Classifier Algorithm 107
5.8 Alerting 109
5.9 Displaying 110
5.10 Install-Time Scanning (ITS) 110

6 Implementation and Evaluation 112
6.1 Architecture . . . 112
6.2 General Architecture of DASAMoid . . . 113
6.2.1 Module Retrieving Applications . . . 113
6.2.2 The Analysis Module . . . 115
6.3 Implementation Environment . . . 129
6.3.1 JAVA and XML . . . 129
6.3.2 Android Studio . . . 129
6.3.3 Android SDK . . . 130
6.4 Deployment of the Environment . . . 131
6.5 Evaluation and Discussion . . . 132
6.5.1 Generalisation for the Learning Dataset . . . 132
6.5.2 Performance in Detection and Prediction of TLMDASA . . . 137
6.5.3 Comparison between Models . . . 139
6.5.4 Detection of Malware Families . . . 140
6.5.5 Runtime Performance . . . 142
6.5.6 Comparison with Antivirus Scanners . . . 143
6.5.7 Comparing Different Methods . . . 145
6.5.8 Comparison with Related Approaches . . . 146
6.5.9 Limitations . . . 147

7 Conclusions and Perspectives 149
7.1 Thesis Contributions . . . 149
7.2 Thesis Limitations . . . 152
7.3 Future Opportunities for Research . . . 152

References 154

Terms and Acronyms 181

I Appendices 184

A Permissions Weights 185

B F-SECURE Classification 192

C Resources Associated to Risks 194

D Scripts for Analysis 200

List of Figures

1.1 Cumulative Android malware samples from November 2010 to January 2014 [Svajcer 2015]. . . . 2
1.2 The user has a choice to install . . . 9

2.1 Architecture of Android [Android Developers 2013b] . . . 14
2.2 Build process of Android applications . . . 18
2.3 Application's source code and different disassemblies . . . 19
2.4 Intent and intent filters . . . 21
2.5 Components private or public . . . 22
2.6 Environment of the application store . . . 25

3.1 Repackaging process . . . 36
3.2 A version of GGTracker . . . 37
3.3 Categorisation of the permission escalation attack. . . . 39
3.4 Permission escalation attack (modified from [Bugiel et al. 2011b], p. 3). . 40
3.5 Permission collusion by malevolent applications . . . 40
3.6 Internal activity invocation or confused deputy. . . . 42
3.7 Application sharing the user ID with a compromised application. . . . 42

4.1 Classification as the task of mapping . . . 54
4.2 General approach for building a classification model . . . 56
4.3 Machine learning flow [Sabnani 2008] . . . 57
4.4 A decision tree for the mammal classification problem . . . 60
4.5 Classifying an unlabelled vertebrate . . . 61
4.6 Examples of contingency table . . . 64
4.7 ROC curve . . . 66

5.1 The TLMDASA architecture . . . 81
5.2 Overview of the Santoku environment . . . 85
5.3 Distribution of DM for the whole sample . . . 89
5.4 Correlation GPLs to DMs . . . 91
5.5 Distribution of resources . . . 98
5.6 Repartition of resources of telephony in samples . . . 98
5.7 Distribution for the weight 2 . . . 100
5.8 Distribution for the weight 3 . . . 100

5.9 Overview of WEKA 104

6.1 DASAMoid Architecture 113
6.2 Home screen 115
6.3 List of user applications 115
6.4 Complete analysis 120
6.5 Selective analysis 120
6.6 Confirmation for the analysis 121
6.7 Possible operations on an application 121
6.8 Scanning results 126
6.9 Settings and preferences for DASAMoid 127
6.10 Intelligent code editor 130
6.11 Android builds evolved, with Gradle 130
6.12 Multi-screen application development 130
6.13 Deployment environment 131
6.14 Steps to discuss the generalisation of the learning dataset 132
6.15 Convergence of perm11 and perm22 134
6.16 Comparison between experiment 1 and 3 134
6.17 Comparison between experiment 2 and 3 135
6.18 Comparison between experiment 2 and 4 135
6.19 Comparison experiment 4 and 1 136
6.20 Comparison experiment 2 and 3 on the whole system 136
6.21 Evolution of AUC in the partitioning dataset 138
6.22 Graphical representation of Table 6.7 138
6.23 Model 3 outperforms models 1 and 2 with RandomForest 139
6.24 Model 1 is more precise than models 2 and 3 in PART 140
6.25 Detection per malware family 142

List of Tables

2.1 History of Android's main releases [SocialCompare 2015] 16
2.2 Example of explicit intent 21
2.3 Example of implicit intent 21
2.4 Some examples of permission descriptions 23
2.5 Correspondence of security mechanisms to architecture layers 29

3.1 Different configurations lead to different levels of vulnerability. 41
3.2 Comparison of type of analysis 48

4.1 Data for classifying vertebrates into one of the categories 54
4.2 Determination of the class corresponding to an instance 55
4.3 An example of a consistent dataset 58
4.4 An example of an inconsistent dataset 59
4.5 General form of the contingency table for a binary classification 64

5.1 Determination of predicted requests of permissions 84
5.2 Tools for static analysis 85
5.3 Examples of determination of DM1 87
5.4 List of permissions found only in malware samples 88
5.5 Vector representation 88
5.6 Vector representing the malware package 030b481.apk 89
5.7 Vector representing the normal package com.whatsapp.apk 89
5.8 DM statistics 89
5.9 Permissions in DM1 and DM8 90
5.10 Example to illustrate the second model 96
5.11 The vector for the application in the previous example 97
5.12 Resource statistics 97
5.13 Distribution by resources and weights 99
5.14 List of applications with no requests of permissions 102
5.15 Representation of the application vector in layer 3 103
5.16 Results of classification 106

6.1 Normal application 117
6.2 Unclassified application 117
6.3 Malicious application 117

6.4 Experiments for the generalisation of the learning dataset 133
6.5 Detection results obtained with the known dataset 137
6.6 Prediction results simulating the unknown dataset 137
6.7 Values of AUC for every partition 138
6.8 Malware families 142
6.9 Comparison of the runtime 143
6.10 Comparison of the package size 143
6.11 Detection results for unknown malware. 144
6.12 Detection results 145
6.13 Classification performances 145

A.1 Permissions with GPL and DM 190
A.2 Permissions of GPL 0 191

B.1 Testing dataset 193

C.1 Resources and risks generated 199

List of Algorithms

5.1 Construction of the vector for the second model 96
5.2 Selection of the association 108

Listings

2.1 Intent filter 22
2.2 Specification of a permission 23
2.3 Portion of a requested permission in the Manifest file 23
2.4 Custom permission 24
6.1 Retrieving and displaying applications 114
6.2 Filtering out user applications 114
6.3 Listing of required permissions 115
6.4 Formatting permissions 116
6.5 Portion of code to analyse an application 117
6.6 Determining the attribute values for model 1 and 2 118
6.7 Portion of code to analyse an application 119
6.8 Portion of code to analyse an application 120
6.9 Notifying a new installation 122
6.10 Automatic analysis of a new installation 123
6.11 Portion of the Manifest, which declares a receiver 124
6.12 Filtering criteria 125
6.13 Fragment portion of code from the complete analysis 128
D.1 Decompiling packages 200
D.2 Removing duplicates 200
D.3 Extracting formatted permissions 201
D.4 Script to determine vectors of normal applications for the model 1 202
D.5 Determining frequency counts 203
D.6 Determining risks for the telephony resources 204
D.7 Determining risks for the network category 205

Chapter 1

Introduction

Android has become the most popular open source operating system for smartphones and tablets with an estimated market share of 70% to 80%, [Canalys 2013], [Llamas/Reith/Shirer 2013]. A shipment of one billion Android devices has been forecast in 2017; over 50 billion applications have been downloaded since the first Android phone was released in 2008. The average number of applications per device increased from 32 to 41 and the proportion of time spent by users on smartphone applications almost equals the time spent on the Web (73% vs. 81%) [Nielsen 2012].

The rapid growth of smartphone technologies and their widespread user acceptance came simultaneously with an increase in the number and sophistication of malicious software targeting popular platforms. Malware (short for **mal**icious soft**ware**) developed for early mobile devices such as Palm platforms and featured mobile phones was identified prior to 2004. The presence of malware specifically developed for them (mostly Symbian OS) grew exponentially with the proliferation of mobile devices with more than 400 cases between 2004 and 2007 [Dunham 2008], [Shih et al. 2008]. The iPhone and Android OS were released later on and became shortly the predominant platforms. This gave rise to an alarming escalation in the number and sophistication of malicious software targeting these platforms, particularly Android OS. Over 250.000 Android users have been victim of an unprecedented mobile attack when they downloaded malicious software disguised as legitimate applications from the Android Market in 2011 ([Lookout 2011], p. 4). According to the mobile threat report published by Juniper Networks in 2012, the number of unique malware variants for Android OS has increased by 3325.5% during 2011 [Juniper 2012a]. A similar report by F-Secure reveals that the number of malicious Android OS applications received during the first quarter of 2012 increased from 139 to 3063 compared to the first quarter of 2011 [F-Secure 2012] and it already represents 97% of the total mobile malware by the end of 2012 according to McAfee [McAfee 2013].

The Sophos Mobile Security Threat Report [Svajcer 2015] demonstrates more recently that the cumulative Android OS malware samples reach almost 700 thousands reported units by 2014 as depicted in Figure 1.1.

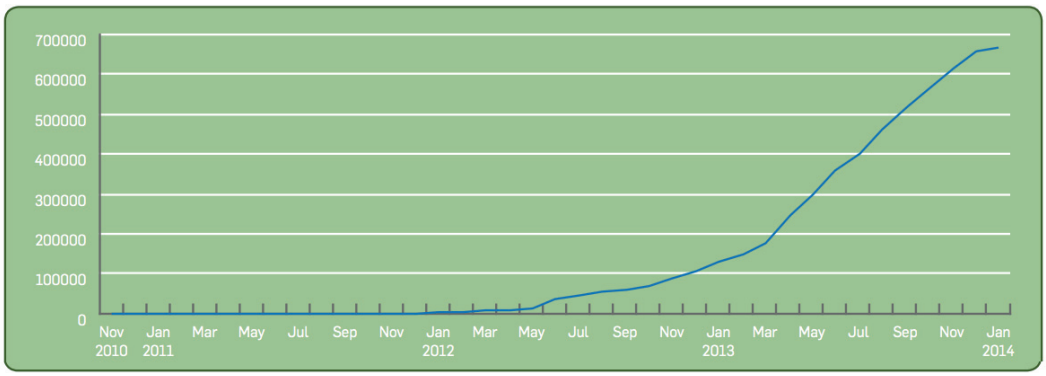


Figure 1.1: Cumulative Android malware samples from November 2010 to January 2014 [Svajcer 2015].

Google's mobile platform is really the target against which cyber criminals expanded their malicious activities. Mobile threat researchers indeed recognise an alarming increase of Android malware from 2012 to 2013 and estimate that the number of detected malicious applications is in the range of 120.000 to 718.000 [Kindsight 2013], [Fortinet 2013], [Juniper 2012b], [Trend Micro 2014]. Google reacted to the growing interest of attackers in Android by revealing Bouncer in February 2012, a service that checks applications submitted to the Google Play Store for malware [Lockheimer 2013]. However, research has shown that Bouncer's detection rate is still fairly low and that it can easily be bypassed [Jiang 2013], [Oberheide/Miller 2012]. The sophisticated Eurograbber attack showed in 2012 that mobile malware might be a very lucrative business by stealing an estimated 23.580.000.000 CFA from bank customers [Kalige/Burkey 2012].

A large body of research against Android malware has been proposed. Research towards fighting malware and securing Android has focused on three main directions. The first one consists of static [Enck/Octeau/McDaniel/Chaudhuri 2011] and dynamic ([Bläsing et al. 2010], [Honeynet Project 2014]) analysis of the application code to detect malicious activities before the application is loaded onto the device. The second consists of modifying the Android system to insert monitoring modules to allow the interception of malicious activities occurring on the device ([Enck et al. 2010], [Nauman/Khan/Zhang 2010], [Xu/Saïdi/Anderson 2012], [Backes et al. 2012]). The third approach consists of using virtualisation to implement the separation of domains ranging from lightweight isolation of an application on the device [Sven et al. 2011] to running multiple instances of Android on the same device ([Andrus et al. 2011], [Gudeh/Pirretti/Hoeper/Buskey 2011], [Lange et al. 2011]). Attacker techniques evolved rapidly with sophistication, changing dynamically the behaviour of applications during their runtime. This poses significant challenges for their detection [Enck 2011], [Zhou/Jiang 2012].

Thwarting malware attacks in Android smartphones is a thriving research area with a considerable amount of still unsolved problems despite such approaches. They come mostly

from limitations of Android design and the design of security mechanisms. One major source of security and privacy problems in Android is the fact that users are able to install third party applications from alternative sources, which do not carry out a revision process over submitted applications for security testing to detect whether the application includes malicious code, as it done in the official Android store. A significant fraction of users rely on alternative markets to get access for free of charge applications in Google Play. Such unofficial markets have repeatedly proven to be a fertile ground for malware, particularly in the form of popular applications modified (repackaged) to include malicious code [Lookout 2011]. One primary line of defence is given by the security mechanisms designed for the device, one of these foremost features is the permission system that restricts application privileges. This has been proven with limitations so far. Application request permissions in a non-negotiable fashion for instance in such a way that users are left with the choice of either granting everything the application asks for at installation time or it will not be possible to use it. Most users simply do not pay attention to such requests; or do not fully understand the meaning of each permission; or, even if they do, it is hard to figure out all possible consequences of granting a given set of privileges. For example, applications requesting a permission to access the accelerometer of a smartphone or a tablet are rather common. However, it has been demonstrated that it is possible to infer the keys pressed by the user on a touch-screen from just vibrations and motion data [Cai/Chen 2011]. Therefore, using such a permission combined with Internet access, which is another common permission, can lead to a serious risk of user sensitive information. Additionally, Android malware is often not limited to an application's sandbox, but to the collaboration of many applications. The permission escalation attack defined by [Fang/Weili/Yingjiu 2014] belongs to this objective. It allows a malicious application to collaborate with other applications in order to access critical resources without requesting for corresponding permissions explicitly [Davi/Dmitrienko/Sadeghi/Winandy 2011], [Bugiel et al. 2011b], [Felt et al. 2011c], [Marforio/Ritzdorf/Francillon/Capkun 2012]. It is based on the fact that a malicious (unprivileged) application can take advantage of benign (privileged) applications to perform malicious actions.

1.1 Relevance for Africa/Cameroon

The growing penetration of Android in Africa demonstrates its adoption by the population [IDC 2013], [VisionMobile 2013]. Android devices are increasingly used in Cameroon to perform sensitive operations such as electronic payment, management of banking accounts and transactions. The availability of such services together with the open nature of this platform attracts attackers.

The Growth and Employment Strategy Paper in Cameroon ([GESP 2010], p.53) states that

"

Following a participatory approach bringing together all development actors of the Nation and founded on the "Greater Achievements" policy of the Head of State, the structural

studies of the system, aspirations of the Cameroonian population and international commitments made by the Government, a shared Development vision for Cameroon by 2035 has ensued. It reads: "CAMEROON: AN EMERGING AND DEMOCRATIC COUNTRY UNITED IN DIVERSITY"

"

In order to achieve the goals set by the Government in the ICT field, a certain number of plans have been launched. They involve organising the use of ICT in order to have reliable and sufficient infrastructure, to facilitate the development of ICT in order to popularise it and to make it possible for all citizens to use it. The Government also has an objective to fight for the security of information manipulated by citizens. The Government created in this regard the National Agency for Information and Communication Technologies (ANTIC) that has the mandate to secure the Cameroonian cyberspace. The first step consisted of fixing laws for security of information. Among them, law No. 2010/012 (21 December 2010) related to cyber security and cyber criminality in Cameroon. ANTIC is also responsible for the regulation, control, and monitoring of activities related to the security of electronic communication networks, information systems, and electronic certification on behalf of the Government of Cameroon. Our research project sustains the vision and provides a strategy to assist users in detecting malware for Android systems that are penetrating increasingly in the continent in general, in Cameroon in particular.

The summit's assembly, comprising Heads of State and Government of the African Union, was held from June 26 to 27, 2014. The African Union established a document called "African Union Convention On Cyber Security and Personal Data Protection" at the end ([African Union 2014], p.1 ff.). The Preamble of this document states that

"

The Member States of the African Union:

[...]

Stressing that at another level, the protection of personal data and private life constitutes a major challenge to the Information Society for governments as well as other stakeholders; and that such protection requires a balance between the use of information and communication technologies and the protection of the privacy of citizens in their daily or professional lives, while guaranteeing the free flow of information;

[. . .]

Concerned by the urgent need to establish a mechanism to address the dangers and risks deriving from the use of electronic data and individual records, with a view to respecting privacy and freedoms while enhancing the promotion and development of ICTs in Member States of the African Union;

[...]

Aware of the need, given the current state of cybercrime which constitutes a real threat to the security of computer networks and the development of the Information Society in Africa, to define broad guidelines of the strategy for the repression of cybercrime in Member States of the African Union, taking into account their existing commitments at sub-regional, regional and international levels ".

The African Union States have agreed on the recommendations of the convention document. This document is to encourage African governments to protect personal data of citizens following the guidelines defined. Our research project is relevant because it is straightforward following the guidelines consisting of protecting sensitive information specifically for Android users in Africa.

1.2 Target and Motivation

Our target is directed towards Android smartphone users. Some elements motivate us to work on this research topic.

Android is the smartphone operating system that largely dominates the African handset market. Research companies [IDC 2013], [VisionMobile 2013] forecast on market share of smartphone operating in Africa. They indicate about 68.85% of Android smartphone sold from the middle of 2013 to nearby 75% in 2017 . This is an appeal to consider preventing security attacks in Cameroon.

New Android smart devices are appearing at a steady pace, including TVs [Samsung 2015], watches [Sony 2015], glasses [Google 2015], clothes [CuteCircuit 2015] and cars [Newcomb 2015]. Paradigms such as wearable computing or the Internet of Things (IoT) are to become reality. Critical domains such as healthcare, which takes advantages of these devices needs security mechanisms.

Google applies the permission system as a measure to restrict access to privileged resources of the system. Android adopts an all-or-nothing permission granting policy. This means: When a user is installing an application, he must either grant all permissions to the application that it requests or refrain from installing the application. The problem is the lack of comprehension of permissions by users [Felt et al. 2012], who authorise unconsciously malware to infiltrate. This situation concerns also Cameroonian users. Researchers propose tools such as permissionWatcher [Struse et al. 2012], but the problem remains. Users should learn how to use these tools before applying them despite the insufficient documentation of permissions [Vidas/Christin/Cranor 2011]. Moreover, they are applicable after the installation of an application during which malware can install its payloads. They do not take into account implicit permissions given to the application. The permissions *INTERNET*, *READ_PHONE_STATE*, *WRITE_SETTINGS* defined by Google are coarse-grained as they give arbitrary access to certain resources of an application. Developers also lack professional knowledge in the process of permission. They might choose to simply over-claim permissions in order to make sure that their applications work anyway [Barrera/Kayacik/van Oorschot/Somayaji 2010].

A variety of permission attacks exploit the previous flaws of the permission system: The permission escalation attack [Fang/Weili/Yingjiu 2014], which allows a malicious application to collaborate with other applications in order to access critical resources without requesting for corresponding permissions explicitly; the collusion attack, in which applications combine their permissions allowing them to perform actions beyond their individual privileges [Schlegel et al. 2011].

During a survey has been realised that apart from using the Internet, banking and payment services are growing in the use of smartphones in Cameroon. Consumers download banking applications to perform banking and payment transactions introduced by mobile operators and established banks. About 3% of Cameroonians use services like Orange Money [Orange 2015] and Abyster [Abyster 2014] for electronic transactions on bank accounts to pay bills, buy and sell products. The manipulation of such information is an attacker target. It is urgent to investigate their protection.

Google develops several programs to encourage the development of Android applications to solve local problems. Competitions such as Africa Android Challenge [AAC 2015] stimulate this spirit of booming the Africa market of Android applications related to healthcare, agriculture, banking, e-commerce, and education. A survey in Cameroon examined the awareness of users about security [Tchakounté/Dayang/Nlong/Check 2014]. It reveals that users are not aware of the security risks. They do not take necessary measures to protect their sensitive data. This thesis is essential to help these users to protect their information.

There are no active research groups working on Android security in Cameroon at the time of writing this thesis. It is a challenge to carry this axe of security into this country.

1.3 Overview of Malware Detection Approaches

Research on the detection of Android malware has been enriched with several concepts and techniques against the spreading of malware. The first approach for detecting Android malware used static program analysis. Several methods have been proposed statically inspecting applications and disassembling their code. Kirin [Enck/Ongtang/McDaniel 2012] checks the permission of applications for indications of malicious activity. Stowaway [Felt/Chin/Hanna/ Song/Wagner 2011] similarly analyses API calls to detect over-privileged applications and RiskRanker [Grace et al. 2012] statically identifies applications with different security risks. Drebin [Arp et al. 2014] is related to this approach and employs permissions, network addresses, and API calls. Open-source tools for static analysis such as Smali [Freke 2013] and Androguard [Desnos/Gueguen 2011] enable dissecting the content of applications efficiently.

Detection of Android malware at runtime has been the second approach. TaintDroid [Enck et al. 2010] and DroidScope [Yang/Yin 2012] enable monitoring applications dynamically during their execution. The first one focuses on taint analysis and the later enables introspection at different layers of the Android platform. Dynamic analysis is mainly applied for offline detection of malware, such as scanning and analysing large collections of Android applications. DroidRanger [Zhou/Wang/Zhou/Jiang 2012], AppsPlayground [Rastogi/Chen/Jiang 2013], and CopperDroid [Reina/Fattori/Cavallaro 2013] have been successfully applied to study applications with malicious behaviour in different Android markets in that direction.

Detection through system calls is a type of dynamic analysis. Although some literature uses system calls to detect malicious repackaged applications, they suffer from few drawbacks ([Isohara/Takemori/Kubota 2012], [Blasing et al. 2010], [Burguera/Zurutuza/Nadjm-

Tehrani 2011], [Shabtai et al. 2011]). They first use the number of system calls, but the metric is too coarse ([Blasing et al. 2010], [Burguera/Zurutuza/Nadjm-Tehrani 2011]), resulting in low detection accuracy. Second, they need the original benign application so that they can distinguish the number of system calls between the original benign application and the malicious one [Burguera/Zurutuza/Nadjm-Tehrani 2011]. [Lin et al. 2013] use the thread-grained system call sequences, because these sequences can be regarded as the actual behaviour of the application. These approaches need generally more refinements in the choice of features to take into account the sequence of calls. Many works evaluate the detection of malware with permissions using machine learning on Android [Sanz et al. 2013a], [Aung/Zaw 2013], [Huang/Tsai/Hsu 2013], [Sanz et al. 2012], [Xing Liu/Jiqiang Liu 2014]. They all realise that a permission-based mechanism can be used as a quick filter to identify malicious applications, and that it must be associated with a second element (such as dynamic analysis) to conduct a complete analysis for a malicious application. However, some others analyse dangerous combinations of permissions to deny the installation of the application [Enck/Ongtang/McDaniel 2012], [Tang/Jin/He/Jiang 2011], [Tchakounté/Dayang 2014].

A third approach is directed to security-code repackaging. This approach consists of rewriting a not trusted application in such a way that the code monitoring the application is directly embedded into its code or its package instead of relying on a system centric solution. Appguard [Backes et al. 2012] for instance is a reference monitor system, which supports permission revocation; Aurasium [Xu/Saïdi/Anderson 2012] is a solution that repackages arbitrary applications to attach user level sandboxing and policy enforcement code. However, this approach can be subject to a high overhead.

1.4 Problem Statement

Existing approaches evolved to address issues of the permission system are fundamentally limited by the categorisation of permissions done by Android. Permissions are grouped into four categories according to Google: normal, dangerous, signature, and signatureOrSystem. The first category cannot impart real harm to the user; it is automatically granted without the user while the second one can have a negative impact if used incorrectly. In the second one (dangerous) the user probably grants the requested permissions. Most users do not understand what each permission means and blindly grant them, allowing the application to access the user's sensitive information, according to [Felt et al. 2012] and [Kelley et al. 2012] . This conceptual categorisation fixed by Google however becomes false when the association of many permissions is considered. For instance, the permission *READ_PHONE_STATE* (which is normal according to Google) allows applications to get some information (identifiers, phone numbers), but the permission itself taken individually is not risky concerning personal data. When, however, the application requests also a number of other permissions such as *INTERNET* (which allows accessing the Internet) personal data can be transmitted to a remote server for bad usage.

Permissions are directed to protect the access of specific resources. During the installa-

tion of an application, Android does not display the specific resource needed by the application and the corresponding permissions. It rather presents different categories representing set of resources with a description to the user. The categories include implicitly permissions necessary to access some resources.

This mechanism gets the user confused and it probably will give more authorisations than it should be allowed (Figure 1.2).

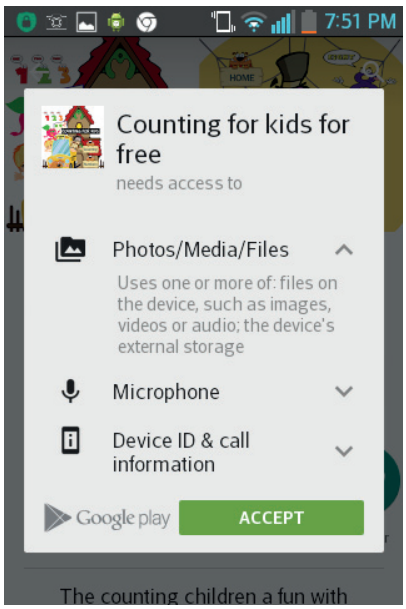


Figure 1.2: The user has a choice to install

Authors relying on permission detection ([Su/Chang 2014], [Sato/Chiba/Goto 2013], [Huang/Tsai/Hsu 2013], [Sanz et al. 2013a], [Sanz et al. 2013b]) restrict their study only to the most requested permissions (or a precise set of permissions) without considering the problem related to resource risk-categorisation.

The research problem provides detection models of Android malware, which consider combinations of permissions and security risks generated by permission resources, while considering any permission as risky. This study is specifically concerned with the following research questions:

1. How to define meaningful risk levels for permissions while considering frequency of occurrence in malicious and normal applications? There are currently several Android permissions officially provided by Google. It is important to define a metric called risk level that relies on the occurrence of permissions beyond the malware and benign software sets. The evaluation of a risk level should consider also low proportions as helpful to detect malicious actions.

2. How to provide a fine-grained reclassification which evaluates individual Android permissions: The permission classification based on protection levels as defined by Google is intuitive and provides false interpretation when permissions are associated. A new classification relying on some deterministic metric is more concise.

3. How to profile Android applications based on fine-grained reclassification? The problem of application characterisation remains uneasy. One should choose the characteristic that represents a signature of the application clearly. According to the new

categorisation, one must be able to represent an application associated to risk level and other minimal features.

4. How to profile Android applications using risks provided by permissions on resources? Android includes the possibility to group permissions according to resources to access and to certain objectives. For instance, the MESSAGES group contains permissions that allow an application to send messages on behalf of the user or to intercept messages being received by the user. This group contains also permissions, which can be used to make users spending money without their direct involvement. However, during the installation, it is only the group, which is displayed to the user, which means that the user grants permissions such as READ_SMS and others that are unknown. Users are confused and not aware of security risks related to granting permissions with this approach. Our research should redefine resource-based categories of permissions based on security risks.

5. How to associate machine classifiers and Android profiles for the detection of malware? Machine learning techniques help training Android samples with profiles to determine rules classifying them either as normal or malicious.

1.5 Main Requirements

There are several requirements that need to be addressed while resolving previous research questions:

- No device rooting: The system does not need to modify the Android architecture as well as the user does not require getting root rights to deploy our solution;
- Usability: The security solution should be well understandable by non-expert users. It reduces the participation of users to take security decisions. Configuring must be easy;
- Performance: The Performance overhead must be smaller compared to similar works.

1.6 Research Hypotheses

The present research assumes that:

1. The device is non-rooted and free of vulnerabilities: To assure that system files are not tampered with.

2. A malicious application with no permission cannot be detected: Since the thesis aims at detecting malware based on permissions, an application without any permission can not be detected.

3. The revaluation of the risk level of permissions can improve the characterisation based on permissions: Only permissions dangerous are displayed to the user for granting during the installation of an application. According to Google, the more an application asks for permissions dangerous, the more the application tends to be malware. However, it is practically false. Vennon and Stroop [Vennon/Stroop 2010] analysed 68% of Google applications for requested permissions and found that more than a half (20.786) is considered mistakenly as malicious if the criteria dangerous is requested [Vennon/Stroop 2010].

4. There exists a combination of permissions proper to malicious applications and to benign ones: We are guided by observations done in the literature and in official documentations on permissions. *INTERNET* permission needs other associated permissions to damage the user, even if it is the most requested permission by malicious applications. The association *INTERNET* and *READ_PHONE_STATE* can allow the application to retrieve device information and send them to a remote server. This association seems to be more dangerous than taking permissions individually. Similar results have been published [Aung/Zaw 2013].

1.7 Objectives

The main goal of this thesis is to automate efficiently the profiling and detection of Android malware. We will focus in the following general objectives to achieve this goal:

- Study of security methods designed for Android as well as their limitations and vulnerabilities;
- Study of the malware landscape on Android as well as the recent progress made to analyse and detect it;
- Study of tools and techniques to assist security analysts and to users in the analysis of untrusted applications for Android smartphones;
- Study of machine learning concepts to apply for the detection of malware as well as research works done using this approach on Android;
- Constitute a dataset of malware based on different techniques and a dataset of benign applications, which supports the study;
- Develop models aiming at better profiling and detecting Android malware with particular emphasis on requested permissions;
- Produce a new classification of Android permissions reusable in unknown samples;
- Produce a lightweight tool that can be installed on smartphones assisting users in the identification of malware.

1.8 Thesis Structure

This document is organised into seven chapters:

Chapter 2 presents the Android ecosystem for a better understanding of the Android security limitations. It dissects the application environment in terms of its architecture, its structure, its composition in terms of components and its version evolution from 2008 to 2015. In order to understand some attacker techniques, which modify an application, chapter 2 elucidates reversing techniques that can be used to retrieve the original source code of the developer from the package. It outlines also techniques to rewrite the code and to recreate the initial package. Since requested permissions are the key feature in our scheme and since these features support the Android security model the chapter tries to describe principles around permissions with related protection levels. The chapter explains the process from distributing to installing Android applications from Google Play or third party sources to the user device. After describing how security is implemented on Android and on Google Play, we discuss open issues that arise from the design of the permission model.

To propose a concrete solution against malware, a deeper understanding of its infiltration method, its payload, and its techniques is obligatory. Malware has to be infiltrated on the user device to perform maliciously. Chapter 3 helps to describe and to understand several important aspects on the malware landscape. The first one is to decrypt each phase of the life cycle of malware from its creation to its destruction by its activation and reproduction. Then, it presents types of malware related to particular threats. It shows frequent techniques employed by attackers such as social engineering and privilege escalation, to steal user information. It reports a variety of attacks in relation to the exploitation of vulnerabilities and the limitations imposed by the Android system presented in chapter 2. The comprehensive analysis on the evolution of malware in smartphones motivates the need for intelligent instruments to automate the analysis. Thus, chapter 3 provides an overview of analysis models used by security researchers in current platforms for smartphones, to understand the behaviour of applications and to search for suspicious components. Particularly, chapter 3 describes the static analysis, the dynamic analysis, and the permission analysis while outlining advantages and disadvantages of such mechanisms.

Machine learning is a set of techniques used to learn and to train application profiles, then to detect and predict their status: malicious or normal. They are characterised by some performance criteria. The understanding of the concepts and practical things around this discipline refers to chapter 4. It illustrates particularly the application of classification learning in this work with examples. It summarises how to determine essential criteria and elements used in the evaluation of the classification and the detection done on dataset samples. It additionally proposes a study of research works aiming to detect Android malware using permissions and machine learning techniques. Finally, chapter 4 extracts advantages and limitations of these approaches.

Chapter 5 defines the TLMDASA system to detect Android malware based on 222 permissions and structured in three layers. The first layer is supported by a model, which

focuses on the discriminating metric based on the frequency of permissions and the proportion of requests by malicious applications within the whole sample. The second layer uses a model, which relies on security risks related to granting permissions. The last layer uses a model, which characterises an application based on an association of vectors derived from the two first layers. Chapter 5 compares the protection levels of permissions defined by Google with the permission classification determined in the first model and justifies, which one is more concise. This chapter defines, tests, and validates a dataset, which includes a collection of 6783 malware and 1993 normal applications, which is selected for the learning phase. Chapter 5 generates complete profiles for each sample depending on layers and transfers them as input for learning and training processes. This chapter applies eight classifiers to models, outlines performance results, and selects the best ones to construct the classifier, which can provide outstanding performance in detection and prediction.

The first part of chapter 6 provides details about the implementation of the system. It specifies its architecture, its modules, and its interfaces. This part describes also tools and technologies used for the development as well as the environment for the deployment of the final product. The second part concerns the evaluation of the system of detection and the implementation. The particular questions are to determine whether our learning dataset can be generalised to an unknown dataset, to know if the model is globally significant, to evaluate the performance of the implementation after deployments, to compare our model with similar ones, to compare our model with renowned antiviruses, and to discuss the obtained results. Chapter 6 ends by mentioning some limitations of the proposed solution.

Chapter 7 draws the main conclusion, analyses the contribution, and presents the limitations of this work. Finally, it presents possible research questions for future work.

Chapter 2

Android Ecosystem

We should explain the Android ecosystem for a better understanding of the problems, before going deeper into the problems of Android malware.

2.1 Architecture of Android

The Android operating system is designed for mobile platforms, especially for smartphones and tablets. The system is based on a Linux kernel and designed for Advanced RISC Machine (ARM) architectures. Android includes similarly to other platforms various layers running on top of each other, the lower ones providing services to the upper level layers. We just give an overview of the architecture of Android presented in Figure 2.1; existing studies [Brähler 2010], [Ehringer 2010] give more details.

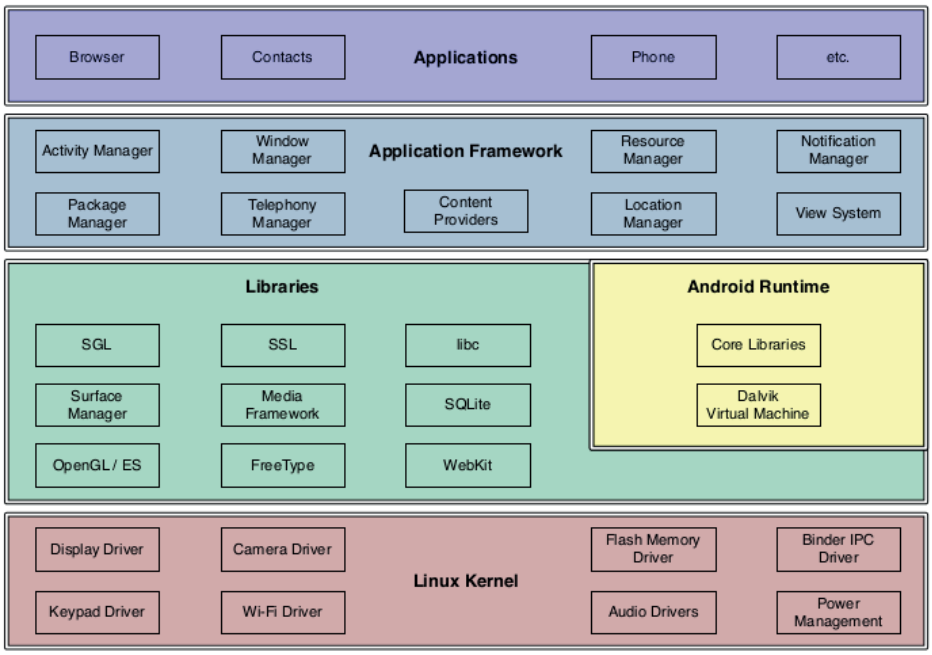


Figure 2.1: Architecture of Android [Android Developers 2013b]

2.1.1 Linux Kernel

Android uses a specific version of the Linux Kernel with some additions. The Linux Kernel is responsible for most of the things that are usually delegated to the operating system kernel, in this case mostly hardware abstraction.

2.1.2 Libraries

A set of native C/C++ libraries is exposed to the application framework and to the Android runtime via the libraries component. It includes the Surface Manager (responsible for graphics on the device screen), 2D and 3D graphics libraries, WebKit, the Web rendering engine that powers the default browser), and SQLite, the basic data store technology for the Android platform.

2.1.3 Android Runtime

Each application runs in its own instance of the Android runtime, and the core of each instance is a Dalvik Virtual Machine (DVM). The DVM is a mobile-optimised virtual machine, specifically designed to run fast on the devices that Android targets. Present in this layer and in each application runtime are also the Android core libraries, such as the Android class libraries (I/O).

2.1.4 Application Framework

The Application Framework provides high level building blocks for applications in form of various android.* packages. Most components in this layer are implemented as applications and run as background processes on the device. Some components are responsible for managing basic phone functions like receiving phone calls or text messages or monitoring power usage. Some examples are: The package manager is responsible for managing applications on the phone and the activity manager is responsible for loading activities and managing the activity stack.

2.1.5 Applications

This includes applications that developers write and those one that Google and other Android developers do as well. Applications running on this layer include typically one or more of four different types of components: activities, broadcast receivers, services, and content providers. Different versions of the Android operating system follow this architecture.

Table 2.1 gives the whole time line of Android major releases specifying the release dates, Application Programming Interface (API) levels and usage. The API level is an integer value that uniquely identifies the framework that API revision offered by a version of the Android platform. The Android platform provides a framework API that can use applications to interact with the underlying Android system. The framework API consists of:

- A core set of packages and classes;
- A set of XML elements and attributes for declaring a Manifest file;
- A set of XML elements and attributes for declaring and accessing resources;
- A set of intents;
- A set of permissions that applications can request, as well as permission enforcements included in the System.

Each successive version of the Android platform can include updates for the Application Framework API that it delivers.

Version	Codename	Release Date	API Level	Usage	Market share
1.0	Apple pie	09/2008	1	Smartphone	0,00%
1.1	Banana bread	02/2009	2	Smartphone	0,00%
1.5	Cupcake	04/2009	3	Smartphone	0,00%
1.6	Donut	09/2009	4	Smartphone	0,00%
2.0	Eclair	10/2009	5	Smartphone	0,00%
2.1	Eclair	01/2010	6,7	Smartphone	0%, 0%
2.2	Froyo	05/2010	8	Smartphone	0.3 %
2.3	Gingerbread	12/2010	9,1	Smartphone	0%, 5.6 %
3.0	Honeycomb	02/2011	11	Smartphone, tablet	0,00%
3.1	Honeycomb	04/2011	12	Smartphone, tablet	0,00%
3.2	Honeycomb	07/2011	13	Smartphone, tablet	0,00%
4.0	Ice Cream Sandwich	10/2011	14,15	Smartphone, tablet	0, 5.1 %
4.1	Jelly Bean	07/2012	16	Smartphone, tablet	14.7 %
4.2	Jelly Bean	11/2012	17	Smartphone, tablet	17.5 %
4.3	Jelly Bean	07/2013	18	Smartphone, tablet	5.2 %
4.4	KitKat	10/2013	19	Smartphone, tablet	39.2 %
5.0	Lollipop	10/2014	21	Smartphone, tablet	11.6 %
5.1	Lollipop	03/2015	22	Smart watches, TV player, car media center	0.8 %

Table 2.1: History of Android's main releases [SocialCompare 2015]

2.2 Android Application

This section concerns the description of the structure of an Android application, the description of components included in an Android application, and tools used to dissemble and to implement Android applications.

2.2.1 Application Structure and Reversing Techniques

We give an overview of the application structure and the contents of the corresponding package as well as a short introduction into the tools used for unpacking and decompiling of Android applications. Android Package (APK) files are unencrypted archives, which contain all the necessary files needed by the application to function. An APK contains the following files and folders:

- META-INF: It is the directory which contains the following files:
  - MANIFEST.MF: The Manifest file,
  - CERT.RSA: The application certificate,
  - CERT.SF: The list of resources and SHA-1 digest.
- Lib: This is the directory, which includes the compiled code specific to a software layer of a processor:
  - armeabi: The compiled code for arm based processors,
  - armeabi-v7a: The compiled code for armv7 and above based processors,
  - x86: The compiled code for x86 processors,
  - mips: The compiled code for MIPS processors.
- Resources.arsc: This file contains pre-compiled resources.
- Res: This directory contains resources not compiled into resources.arsc.
- AndroidManifest.xml: This is the Manifest file. It includes meta information of the application and describes the application capabilities.
- Classes.dex: The compiled classes in the Dalvik Executable (DEX) file format.

Figure 2.2 shows the simplified process of how Java source code projects are translated to APK files.

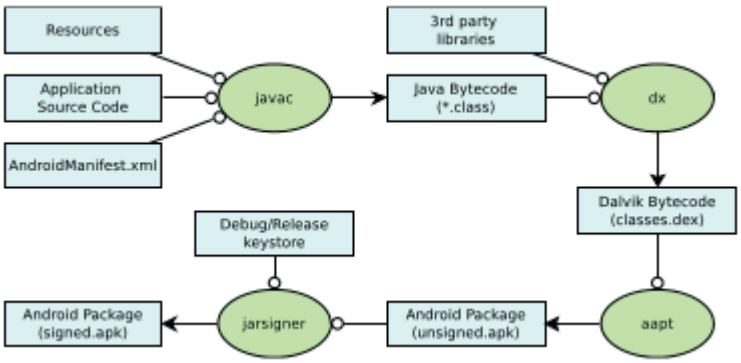


Figure 2.2: Build process of Android applications

Android applications themselves are mainly written in Java with support for their own native libraries written in C. The Java source code gets compiled to a DVM executable byte code, which is stored in a DEX file.

The Manifest, which is very important to execute the application, is called AndroidManifest.xml. It contains all permissions, listeners, receivers and the meta information of the application.

The DEX file, the Manifest, all resources, the certificates and own libraries for the application are packaged to a ZIP archive file with the APK suffix. This APK file is provided via the Google Play to the users. The source code of an Android application is not available in clear text when unpacking an APK file.

Tools are often needed for an analysis to get the source code, which is as close as possible to the original code. The tool dex2jar [Dex2jar 2015] decompiles the Dalvik Virtual Machine (DVM) byte code to the Java byte code, which is easily readable with a graphical Java decompiler like JD-GUI [JD-GUI 2015]. Unfortunately, dex2jar has some limitations and is sometimes not able to retrieve the corresponding Java byte code. A second tool for disassembly is used for this reason: baksmali. This tool disassembles DVM byte code to a new language called Smali [Smali 2015], which is easily readable and which can be reassembled to an Android application.

We illustrate the source code of a simple “hello Android” application and the different disassembles in Figure 2.3a. The Smali output of the main Java class is shown in Figure 2.3b. The disassembly of IDA Pro [IDA 2015] is illustrated in Figure 2.3c for a more comprehensive overview.

```java
public class MainActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        tv.setText("Hello, Android");
        setContentView(tv);
    }
}
```

(a) Original source code

```
.class public Lcom/example/helloandroid/MainActivity;
.super Landroid/app/Activity;
.source "MainActivity.java"

# direct methods
.method public constructor <init>()V
    .registers 1
    .prologue
    .line 7
    invoke-direct {p0}, Landroid/app/Activity;-><init>()V
    return-void
.end method

# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
    .registers 4
    .parameter "savedInstanceState"
    .prologue
    .line 11
    invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
    .line 12
    new-instance v0, Landroid/widget/TextView;
    invoke-direct {v0, p0}, Landroid/widget/TextView;-><init>(Landroid/content/Context;)V
    .line 13
    .local v0, tv:Landroid/widget/TextView;
    const-string v1, "Hello, Android"
    invoke-virtual {v0, v1}, Landroid/widget/TextView;->setText(Ljava/lang/CharSequence;)V
    .line 14
    invoke-virtual {p0, v0}, Lcom/example/helloandroid/MainActivity;->setContentView(Landroid/view/View;)V
    .line 15
    return-void
.end method
```

(b) SMALI disassembly

```
CODE:0002C990                 Method 2958 (0xb8e):
CODE:0002C990                 public void
CODE:0002C990                 com.example.helloandroid.MainActivity.onCreate(
CODE:0002C990                   android.os.Bundle p0)
CODE:0002C990 this = v2
CODE:0002C990 p0 = v3
CODE:0002C990                 invoke-super                {this, p0}, <void Activity.onCreate(ref) imp. @ _def_Activity_onCreate@UL>
CODE:0002C996                 new-instance                v0, <t: TextView>
CODE:0002C99A                 invoke-direct               {v0, this}, <void TextView.<init>(ref) imp. @ _def_TextView__init_@UL>
CODE:0002C9A0                 const-string                v1, aHelloAndroid # "Hello, Android"
CODE:0002C9A4                 invoke-virtual              {v0, v1}, <void TextView.setText(ref) imp. @ _def_TextView_setText@UL>
CODE:0002C9AA                 invoke-virtual              {this, v0}, <void MainActivity.setContentView(ref) imp. @ _def_MainActivity_setContentView@
CODE:0002C9B0
CODE:0002C9B0 locret:
CODE:0002C9B0                 return-void
CODE:0002C9B0                 Method End
```

(c) IDA Pro disassembly

Figure 2.3: Application’s source code and different disassemblies

2.2.2 Composition

Android applications include different core components. We refer to the official documentation[1] to describe them.

2.2.2.1 Activity

An activity presents the user interface (UI) of the application (buttons, checkboxes, list, text, etc.). It is similar to a single screen displayed on the device to a user that is composed of what the user currently sees. We also can compare it to a page of a Website. Each activity in an application has different UIs just like each page in a Website has a different look. Android takes the user to the appropriate activity like clicking links on a Website to go from one page to another one depending on the user's actions. Each activity contains one or more views, which are objects that a users can interact with such as reading text, clicking a button, etc.

2.2.2.2 Service

A service is designed for a process, which runs in the background. These types of components generally run when your application is not visible; that means none of its activities are displayed to the user. To illustrate, consider a music application, which should keep playing the music even after the user has left the application. The activity (UI) will display the play/pause buttons. The button actions will trigger a service ON or OFF. The service will start or pause the playback of a music file. Even if the user leaves the UI of the application (activity), the service will continue to perform its action.

2.2.2.3 Broadcast Receiver

A broadcast receiver is a type of component without any user interface that listens for requests related to system actions. Applications can send either direct messages to a specific component or broadcast these messages system-wide to all applications that are running. This component can receive these system-wide broadcasts and acts upon them. There are occasionally phone-wide broadcast announcements made by the Android system (for example, battery low, download completed, headset connected). The broadcast receiver can start an activity, when receiving a broadcast announcement. It can use the phone vibration, flashlight or sounds to notify the user of any broadcast that it has received.

2.2.2.4 Content Providers

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

[1] http://developer.android.com/guide/components/fundamentals.html

2.2.2.5 Intents

Activities, services, and broadcast receivers are activated by an asynchronous message called intent. Intents define an action that should be performed for activities and services (e.g., view or send). They may include additional data that specify what to act on. A music player application, for example, may send view intent to a browser component to open a Web page with information on the currently selected artist. The intent simply defines the current announcement that is being broadcast for broadcast receivers. The additional data field will contain the content of the message and the sender's phone number for an incoming SMS text message. An intent is explicit when the target component to start is specified as parameter in the intent structure. The intent is otherwise implicit.

Explicit intent
Intent actIntent= new Intent(myContext,com.example.testapplications.test1.mainActivity.class); myContext.startActivty(actIntent);

Table 2.2: Example of explicit intent

Implicit intent
Intent actIntent= new Intent(Intent.ACTION_VIEW, actURI); mContext.startActivty(actIntent);

Table 2.3: Example of implicit intent

2.2.2.6 Intent Filters

An intent is sent to the whole Android system if it does not specify a target component. The Android system will then have to determine suitable applications for this intent. Android offers the choice to open one of them when several components have been registered for this type of intent. This selection is based on intent filters. An intent filter specifies the types of an intent to which an activity, a service or broadcast receivers can respond. It declares the capabilities of a component and specifies what an activity or service does and what types of broadcast receivers can handle. It allows the corresponding component to receive intents of the declared type.

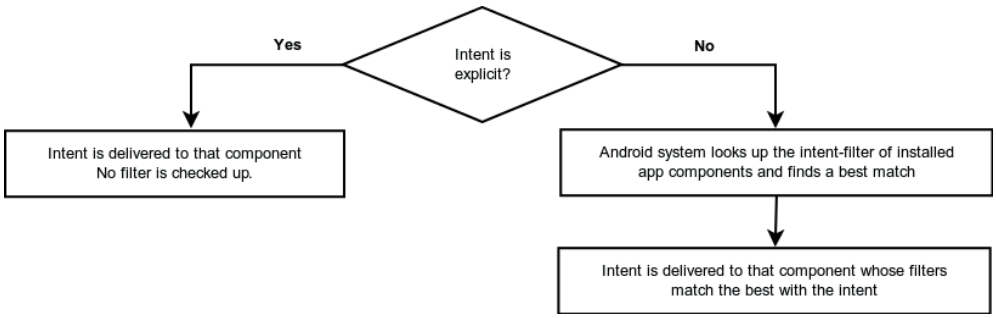


Figure 2.4: Intent and intent filters

2.2.2.7 Component: Public or Private?

A component can be public or private. It is represented in the Manifest file by the attribute exported.

- Public: A component is public when components of the other applications can interact with it. The attribute exported is set to true in this case that means that the component is exported to other applications.
- Private: A component is private when it can just interact with components within the same application or another application that runs with the same UID. The attribute exported is set to false in this case; that means that the component is not exported to other applications.

This mechanism for components is obviously linked to how intents are implemented. A component can specify an intent filter that allows it to receive intents from other applications to perform tasks. We distinguish two cases:

- If a component specifies an intent filter, then the programmer wants the component be accessed by other applications; i.e. exported is true.
- If no intent filter is specified for a component, the interaction with it is only possible if the received intent is explicit. It means that the programmer's idea is not to make this component public; exported is therefore false. It is possible in this case to override the default value of an exported attribute by fixing it as true as shown in the following example:

```
1 <service android:name=".MailListenerService"...
2 android:exported="true"
3 <intent-filter></intent-filter
4   </service>
```

Listing 2.1: Intent filter

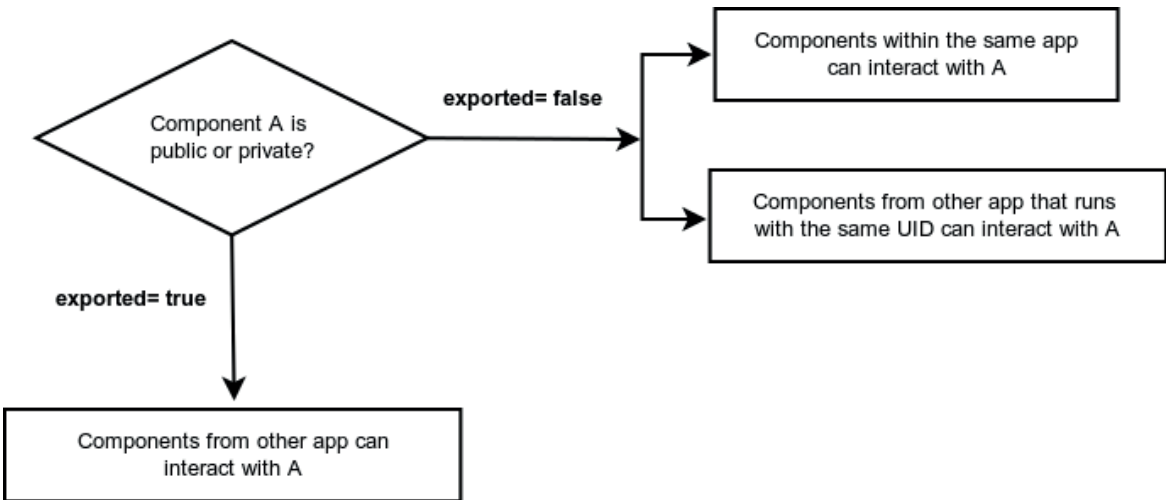


Figure 2.5: Components private or public

2.2.2.8 Application Permissions and the Manifest File

The security model in Android is mainly based on permissions. A permission is a restriction limiting the access to a part of the code or to data on the device. The limitation is imposed to protect critical data and code that can be misused to distort or to damage the user experience [Felt et al. 2012]. Permissions are used to allow or to restrict an application access to restricted APIs and resources. The Android *INTERNET* permission, for example, is required by applications to perform network communication; opening a network connection is restricted by the *INTERNET* permission. An application must have the *READ_CONTACTS* permission in order to read entries in a user's phonebook as well. The developer declares a <uses-permission> attribute to require a permission. The android:name field specifies the name of the permission.

```
1 <uses-permission android:name="string"/>
```

Listing 2.2: Specification of a permission

If an application wants to communicate over the network, for example, it needs an entry like it is shown in Listing 2.3.

```
1 <manifest package="com.sled.hf.mads">
2 ...
3 <uses-permission android:name="android.permission.INTERNET"/>
4 ...
5 </manifest>
```

Listing 2.3: Portion of a requested permission in the Manifest file

The Android operating system defines natively several permissions, which are specified as static string members in the android.Manifest.permission class. Table 2.4 gives some examples of permissions[2].

Manifest Permission Strings	Description
CALL_PHONE	It allows an application to initiate a phone call without going through the dialer user interface to confirm the call being placed.
MODIFY_PHONE_STATE	It allows modifications of the telephony state such as power on.
WRITE_SMS	It allows an application to write SMS messages.
READ_CONTACTS	Allows an application to read the user's contact data.

Table 2.4: Some examples of permission descriptions

[2]https://github.com/android/platform_frameworks_base

A permission can be associated with one of the following four protection levels [Android 2015]:

- Normal: A low-risk permission, which allows applications to access API calls (e.g., *SET_WALLPAPER*) causing no harm to users.
- Dangerous: A high-risk permission, which allows applications to access potential harmful API calls (e.g., *READ_CONTACTS*), such as leaking private user data or control over the smartphone device. Dangerous permissions are explicitly shown to the user before an application is installed. The user must choose whether he grants the permission, determining that the installation continues or fails.
- Signature: A permission, which is granted if the requesting application is signed with the same certificate as the application, which defines the permission.
- Signature-or-System: A permission, which is granted only if the requesting application is in the same Android system image or is signed with the same certificate as the application, which defines the permission.

An application can additively define its own custom permissions to protect its resources by declaring a <permissions /> attribute. It can also specify permissions to restrict access to the system components by defining the *android: permission*. Listing 2.4 illustrates a custom permission.

```
1 <manifest xmlns:android="http://schemas.android.com/apk/res/
      android" package="com.example.testapplications.test1">
2 ...
3 <permission android:name="com.example.testapplications.test1.perm
      .READ_INCOMING_EMAIL" android:label="Read␣incoming␣email"
      android:description="description␣of␣the␣permission␣" android:
      protectionLevel="dangerous" android:permission="android.
      permission-group.PERSONAL_INFO"
4 />
5 ...
6 </manifest>
```

Listing 2.4: Custom permission

This XML block defines a new permission, *READ_INCOMING_EMAIL* belonging to the com.example.testapplications.test1 package. The protectionLevel attribute is set to dangerous; this permission will consequently be displayed to the user. The new permission is of type *PERSONAL_INFO* defined in the android.permission-group package in this case; it will therefore be presented to the user right next to other permissions that allow access to personal information when the list is presented to the user.

2.2.3 Signature

Android applications are digitally signed. The digital certificate associated with a private key identifies each developer. Self-signed certificates are the standard way of signing An-

droid applications and are generated by the developers themselves. Android applications must be signed before they can be installed on a device. The Android system does not require a certificate issued by Certificate Authorities (CA); it accepts self-signed certificates. The Android tool set and development environment supports both debug keys/certificates and release keys/certificates. When developers implement and test an application, whenever compiling, the tool set will sign the resulting APK file with an automatically generated key. jarsigner and keytool are specific tools that can be used.

2.2.4 Development

The Android ecosystem is based on two pillars:

- The Java programming language,
- An Software Development Kit (SDK), a tool box for the developer.

The development kit gives access to examples, documentation, an API, and an emulator to test applications. The plug-in Android Development Tool (ADT) integrates functionalities of the *SDK*. The developer installs it as a classical plug-in by specifying its URL. The next step consists of defining precisely the SDK location (already uploaded and unzipped) in the ADT preferences.

2.3 Android and Third Party Markets

Android users are free to install any (third party) application via the Google Play Store (previously known as the Android Market). Google Play is an online application distribution platform where users can download and install free or paid applications from various developers including Google itself (Figure 2.6).

Users have the possibility to install applications from other sources than Google Play. A user must enable for this the unknown sources option in the device's settings overview and explicitly accept the risks of doing so. Users can install APK files downloaded from the Web directly by using external memory.

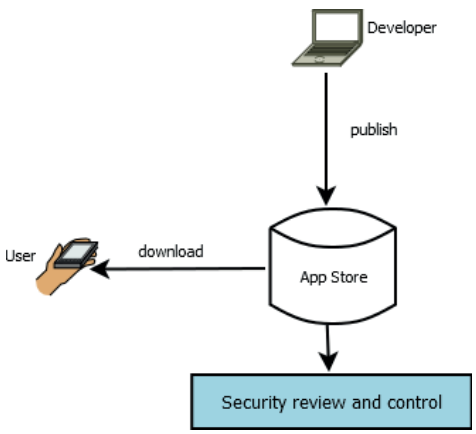


Figure 2.6: Environment of the application store

There are often no entry limitations for mobile application developers, which result in poor and unreliable applications being pushed to these stores for Android devices. Juniper Networks finds that malicious applications often originate from these marketplaces, with China (173 stores hosting some malware) and Russia (132 infected stores) being the world's leading suppliers [Juniper 2012a].

2.4 Android Security Landscape

This section focuses on security aspects around the Android operating system, which are also of special interest for the understanding of this thesis. This section is compiled from the official documentation on security[3], [Fedler/Banse/Krauß/Fusenig 2012], [Six 2011], [Shabtai et al. 2009], and [Spreitzenbarth 2013].

2.4.1 Security on Android

Android benefits from two more Linux based mechanisms: User management via the Portable Operating System Interface (POSIX) and file access via the Discretionary Access Control (DAC). Both mechanisms are based on the fact that every Process Identifier (PID) is associated with one User Identifier (UID). Another typical security feature in Android is the permission of an application that regulates access rights to system services or other installed applications. Application signing is an additional key feature of the Android system and is also used for security purposes.

Android follows the Principle of Least Privilege (PLP) that stipulates entities having just enough privileges to do their job and no more as a prerequisite for security. For instance, if an application does not need Internet access, it should not request the Internet permission.

2.4.1.1 Application Sandbox

The Android platform takes advantage of the Linux user based protection as means of identifying and isolating application resources. The Android system assigns a unique UID to each Android application and runs it in a separate process. This approach is different from other operating systems (including the traditional Linux configuration), where multiple applications run with the same user permission.

2.4.1.2 File Access

The kernel enforces security between applications and the system at the process level through standard Linux facilities, such as user and group IDs that are assigned to applications. All processes spawned from that application belong to the same UID and get separated address spaces, which are only accessible from other processes with the same UID. Applications cannot interact with each other by default; they also have limited access to the operating system. If application A tries to do something malicious like read application B's data or

[3] http://source.android.com/tech/security/index.html

dial the phone without permission (which is a separate application), then the operating system protects against this because application A does not have the appropriate user privileges. The sandbox is simple, auditable, and based on UNIX style of user separation of processes and file permissions. If different processes need to communicate with each other, this can be done by requesting the sharedUserId feature. The application receives the same UID during installation by requesting this feature; it is able to access the address space of the running process from other applications with the same UID and the enabled sharedUserId feature without the need of special permissions.

2.4.1.3 Application Signing

The Android system requires all applications to be digitally signed with a certificate whose private key is held by the developer of the application, otherwise it will not be accepted by the system (unless the system settings are set on the development environment)[Android Developers 2014]. An application is valid as long as the certificate is valid and the public key validates the signature of the file. One does not need a certificate authority unlike other certification or signing operations: the typical Android application is self-signed. Signing applications in Android is used to verify that two or more applications are from the same owner. This feature is used by the sharedUserId mechanism and by the permission mechanism to verify Signature and SignatureOrSystem protection level permissions; only applications signed by the same developer and the same key can be coupled [Shabtai et al. 2009].

2.4.1.4 Dalvik Virtual Machine and Memory Management Unit

Each Android application is executed in the Dalvik VM. This VM is adapted for low memory devices and for spawning multiple instances of it. Each process receives its own instance of the VM, which creates a sandboxed environment. This protects the running application and the allocated memory against other applications. Android employs the Memory Management Unit (MMU) in such a way that one process is unable to read the memory pages of another one, or to corrupt its memory. The likelihood of privilege escalation is reduced since a process is unable to make its own code running in a privileged mode by means of overwriting the private OS memory.

2.4.1.5 Permission System

The core of the application level security in Android is the permission system, which enforces restrictions on specific operations that an application can perform. The package manager is in charge of granting permissions to applications at installation; the application framework is in charge of enforcing system permissions on runtime. Due to the fact that applications need to interact with other applications, system services or even special hardware, applications sometimes need to escape the sandbox environment. This is done through the permission model in Android enforcing restrictions on specific operations an

application can perform. We consider the following example to illustrate the procedure of security through permissions ([Spreitzenbarth 2013], p. 19):

"*If an application A wants to access the stored calendar on the device, it sends a request to the reference monitor. This request is called InterComponent Communication (ICC). The reference monitor reads the permission label of the calling component (in this case application A) and checks whether it has the necessary permissions to access the calendar. Depending on the result of this check, application A may access the calendar or access is denied.*"

2.4.1.6 Component Security

The permission mechanism should be used to secure the various components in addition to protect framework APIs. This capability is achieved primarily by associating permissions with the relevant component in its declaration in the Manifest having Android automatically enforced for the existence of the permissions in the relevant scenarios. An activity can specify a set of permissions that will be required for any application wishing to launch it. A service can control similarly, which applications will be allowed to start/stop it or, which ones to bind to this service. A more fine-grained control of any API exposed through binding can be obtained through runtime checks of the permissions granted to the bound application, whereas such permissions provide a crude restriction. A content provider may define permissions to regulate who is allowed to read or write the information associated with itself. The content provider has fine-grained control over the access since the read and write permissions are defined separately and are not interconnected. Intent broadcasts and broadcast can both be associated with a set of permissions. This feature makes it possible for the broadcast receiver to control, which components are allowed to broadcast the intents for which it is configured to receive. This associative feature provides the ability to control, which broadcast receivers will receive the broadcasted intent. The implication is that any attempt to spy on the intents being broadcasted is captured and efforts to spoof broadcasts by unauthorised components will fail.

2.4.1.7 Component Encapsulation

Android disallows by default any access to components from other applications by defining the exported feature of the component. Whether a component is encapsulated, has no effect on its ability to access other components, which is limited only by their encapsulation and access permissions.

2.4.2 Mapping Android Security Mechanisms to Android Layers

Table 2.5 corresponds Android security mechanisms to the different layers of the architecture [Zhauniarovich 2014].

Mechanisms	Layers
The application sandbox	Kernel
File access	Kernel
Application signing	—-
Dalvik virtual machine and memory management unit	Kernel
Application permissions	Application framework
Component security	Application framework
Component encapsulation	Application framework
Firmware for security	—-

Table 2.5: Correspondence of security mechanisms to architecture layers

2.4.3 Security from Google

Google disposes security techniques to detect if the application includes malicious code as well as some actions to help users inside their phone.

2.4.3.1 Bouncer

Google tests applications for possibly malicious behaviour through a service called Bouncer in 2012 [Lockheimer 2012]. Bouncer examines applications submitted to Google Play with the following process [Google Mobile Blog 2015]: Once an application is uploaded, the service Bouncer immediately starts analysing it for known malware, spyware and trojan. It also looks for behaviour that indicates an application might be misbehaving, and compares it against previously analysed applications to detect possible red flags. Every application is executed on Google's cloud infrastructure, to simulate how it will run on an Android device to look for hidden, malicious behaviour. Bouncer analyses also new developer accounts to help to evacuate malicious developers.

2.4.3.2 Remote Installation and Uninstallation

Google possesses the ability to remotely remove or install any application from/to all Android devices. It is used for the removal of malicious applications and the addition of new services to phones, which may require the installation of new applications. This is executed through the Android Market protocol's *REMOVE_ASSET* and *INSTALL_ASSET* commands. The latter is in fact used for normal installation of an application through Google Play as well.

2.4.3.3 Patch Process

The Android OS patch process is a complex and time consuming combined effort with many parties involved. Google has to develop and push out a patch, when a vulnerability has been discovered. This patch is merged with the Android OS source code, which is then provided to device manufacturers. They adjust the new code base to their devices, affecting not only original equipment, but also handsets specifically produced for carriers or geographical regions. This may include branding, software enforced usage limitations (e.g., prohibition of tethered network access) or even special hardware features. Finally, a system image (ROM) has to be generated and distributed over the air to the customer devices for installation.

2.4.4 Limitations

Android disposes several limitations and flaws, which can expose users to malicious actions, although it is the most used mobile OS. This section presents only those related to the permission system, which is the principal concern of this thesis.

2.4.4.1 Sandboxing

The limitations imposed by the sandboxing make attacks on the underlying operating system from Android applications nearly impossible. However, native Advanced RISC Machine (ARM) architecture code can be compiled for Android devices; both the Linux kernel and essential libraries are implemented in C. Native code may even be called from within Android applications, for the desire of the user or malicious usage. As native code is executed outside the Dalvik VM, it is not limited to API methods, thus receiving more direct and uncontrolled access to the operating system core.

According to [Fang/Weili/Yingjiu 2014], there are in general four issues on the permission model.

1. Coarse-granularity of permissions: Most of Android permissions are coarse-grained. For instance, *INTERNET* permission [Barrera/Kayacik/van Oorschot/ Somayaji 2010], *READ_PHONE_STATE* permission [Pearce/Felt/Nunez/Wagner 2012], and *WRITE_SETTINGS* permission [Jeon et al. 2012] give arbitrary accesses to certain resources: *INTERNET* permission allows an application to send HTTP(S) requests to all domains and connect to arbitrary destinations and ports [Felt/Greenwood/Wagner 2010]. The *INTERNET* permission therefore provides insufficient expressiveness to enforce control over the Internet accesses of the application [Barrera/Kayacik/van Oorschot/-Somayaji 2010].

2. Over-claim of permissions: Over-claim of permissions is probably the most severe threat to Android security. It directly breaks the Principle of Least Privilege (PLP) [Saltzer 1974]. This violation of PLP exposes users to potential privacy leakage and financial losses. For example, if a standalone game application requests the

SEND_SMS permission, which is unnecessary, the permission can be exploited to send premium rate messages without user's acknowledgment. Developers may make wrong decisions because of several reasons, concluded by Felt et al. [Felt/Greenwood/Wagner 2010]: At first, developers tend to request for permissions with names that look relevant to the functionalities they design, even if the permissions are not actually required; second, developers may request for permissions, which should be requested by deputy applications instead of their own application; finally, developers may make mistakes due to using copy and paste, deprecated permissions, and testing artefacts. Other issues, including coarse-granularity of permissions, incompetent permission administrators, and insufficient permission documentation are drivers of over-claim of permissions.

3. Incompetent permission administrators: Both developers and users lack professional knowledge in the process of permission. They have sometimes conflicting interests [Han et al. 2013]. A developer may not know precisely user risks, once permissions declared in the Manifest are granted. Developers might choose to simply over-claim permissions to make sure that their applications work anyway [Barrera/Kayacik/van Oorschot/Somayaji 2010], while others might take time to learn individual permissions to request them appropriately. A survey done by Felt et al. [Felt et al. 2012] shows that only 3% of respondents (users) answered correctly having understood permissions and 24% of the laboratory study participants demonstrated competent but imperfect comprehension.

4. Insufficient permission documentation: Google provides a great deal of documentation for Android application developers, but the content on how to use permissions on the Android platform is limited [Vidas/Christin/Cranor 2011]. The insufficient and imprecise permission information confuses Android application developers, who may write applications with guesses, assumptions, and repeated tries. This leads to defective applications, which become threats with respect to security and privacy of users [Felt et al. 2011a]. The content of permissions is usually too technical for users to understand. Google describes the *INTERNET* permission as follows: "*allows an application to create network sockets*" [Android 2011]. This description seems to be too complex and abstruse for the user. The user might not know exactly risks related to this permission once granted.

Chapter 3

Android Malware Landscape

A malicious application or malware refers to an application that can be used to compromise an operation of a device, steal data, bypass access controls or otherwise cause damage on the host terminal. Normal or benign applications or good software are in contrast those that do not perform any dangerous action on the system. Android malware is malicious software on the Android platform. This chapter is dedicated to the presentation of existing malware on the Android platform. The first section of the chapter presents their life cycle, their typology and evolution. The second section describes the main techniques of attack used to infect and cause damage to the victims. A review of methodologies and tools used for studying and detecting these malware is proposed finally.

3.1 Malware Life Cycle

Malware for mobile platforms in general and Android in particular reproduce the behaviour of viruses encountered on desktop [Bloch/Wolfhugel 2006]. Their life cycle is structured around seven main phases.

- **Creation**: Step in which the programmer designs and implements all malicious code that will be included in the malware.
- **Gestation**: Stage during which the malicious application infiltrates and settles in the system that it wants to infect. It remains inactive throughout this stage. This is why its presence remains totally unknown for the user.
- **Reproduction or infection**: The malware reproduces a significant number of times before manifesting in this phase. The author of the malware seeks to remotely control devices and access private data. Malware spreads via file sharing or social engineering techniques on Android. It uses SMS, Bluetooth, Wifi as communication means and often disguise themselves as a normal application [Castillo 2011].
- **Activation**: Some malware activates their destruction routine when certain conditions are satisfied (internal countdown reaches for example). The activation can also

be done remotely. The purpose of this phase is to appropriate gradually all device resources.

- **Discovery**: The user notices strange behaviour and suspects the presence of a malicious application. This strange behaviour may include performance losses, current changes in the Web browser home page or the unavailability of certain system functions. Anti-viruses often assist the user in detecting malicious actions in sending alerts to the device owner. However, the furtive character of certain malware may extend, even complicate this phase.
- **Assimilation**: Antiviruses update their virus database after the discovery of new malware. If possible, a fix or antidote is also proposed to eliminate this threat.
- **Elimination**: This is the phase when the antivirus discovering the malware prompts the user to remove it. It marks the *death* of the malware.

3.2 Android Malware Threats and their Evolution

Felt et al. [Felt et al. 2011d] and Spreitzenbarth [Spreitzenbarth 2013] elucidate types of threats found on Android mobile applications, which are summarised in this section.

3.2.1 Malware

Attackers want to gain access to a device by installing malware on it. The purpose is to steal data or damage the device. Malware is installed by tricking the user to install a legitimately looking application or in most cases to exploit vulnerability on the device, e.g., a security flaw in the Web browser.

3.2.2 Spyware

One of the most common types of malicious applications for the Android platform is spyware. It is designed to get sensitive information from a victim's system and transfer this information to the attacker. Spyware can be commercial and malicious. Commercial spyware are applications installed on the user's handset manually by another person specifically to spy on the user, while malicious spyware covertly steal data and transmit it to a third party. An example of commercial spyware is CarrierIQ, used extensively by various mobile device manufacturers and vendors [Kravets 2013]. CarrierIQ had the capability to log everything that was done on a device, including Web search using the secure HTTPS protocol, and was allegedly used to increase customer satisfaction by logging dropped calls and similar information [CarrierIQ 2013].

3.2.3 Grayware

The main purpose of grayware is to spy on users who installed the software on their own because they thought that it is legitimate software. This is partly correct because the authors include real functionality as advertised. Nevertheless, they also collect information from the system such as the user's address book or his browsing history. The main goal is to collect information for marketing purposes, etc.

3.2.4 Fraudware

Corresponding applications are installed by tricking the user to install a legitimately looking application, which will gain full functionality after sending several premium-rated SMS messages. In contrast to malware, fraudware informs the user about the upcoming charges, but this information is often hidden and not minded by the user.

3.2.5 Trojan

Trojans are applications that pretend to be useful, but perform malicious actions in the background such as downloading additional malware, modifying system settings, or infecting other files on the system. Android malware is mostly Trojans. The attack vectors used by viruses and worms are largely unavailable to malware developers because of the sandboxing model described in Section 2.4.1. The malicious code is usually included into legitimate applications, which are then redistributed [Cluley 2012] as the original application. Applications misused for this purpose are often paid applications redistributed as free applications on third party markets.

3.2.6 Root exploit

Root exploits are possible on Android in order to gain control of the device, but are considered a double-edged sword among the security community. Rooting can give the user control over a device and also gives the same amount of control to any application, which gains access to the root rights. Root privileges given to a malicious application can completely compromise the device, as the application can theoretically remove the root privileges from the user. This is a security flaw in this case. The malicious application pretends to be normal until it is installed on the user's device, like most Trojans. It attempts, when installed, to use one or more root exploits to gain root access to the device. An application with root access can replace, modify and install applications as it wishes. The DroidKungFu Trojan [F-Secure 2013] is an example. It installs a backdoor on the phone once it has gained root access. It then disguises this backdoor from the user both by using a name, which looks innocent and hiding the application icon from the user. This backdoor can then be used for installing other malicious applications on the device or simply for stealing private information.

3.2.7 Bot

Bots are an emerging trend in mobile malware that communicate with and receive instructions from one or more Command and Control (C&C) servers, giving the malware writer remote control over all infected devices. Malware developers obfuscate their code and use encryption to hide critical information susceptible to detect them. Some examples of malware in this case are DroidDream and jSMSHider [Bradley 2015]. The recent version of the DroidKungFu Trojan [F-Secure 2013], mentioned earlier, was used to create a group of bots consisting of compromised Android devices. Moreover, a bot can install additional applications without any user knowledge or intervention.

3.2.8 Privilege Escalation Exploits

They are part of applications that use vulnerabilities to gain full access to a device. Every Android application runs in a security sandbox; however, if malware is successful in escalating its privileges, it is able to perform actions normally not allowed to applications. DroidDream contains two exploits, Exploid and RageAgainstTheCage, that it used to gain root access and install a secondary application that allows the malware to install additional applications without user knowledge. Another piece of malware, jSMSHider, was signed with a compromised key that also allowed installing applications without user intervention on any mobile device firmware builds that were also signed with that key.

We will understand malware as applications belonging to threats presented previously throughout this work. We give a short overview of the main techniques used in the gestation phase in the next section.

3.3 Malware Techniques on Android

Zhou and Jiang [Zhou/Jiang 2012] categorise existing ways used by Android malware to install on user phones and generalise them into three main social engineering-based techniques: repackaging, update attack, and drive-by download. They are used during the gestation phase.

3.3.1 Repackaging

It is one of the most common techniques that malware authors use to piggyback malicious payloads into popular applications. In essence, malware authors may locate and download popular applications, disassemble them, enclose malicious payloads, and then reassemble and submit the new applications to Google Play and alternative markets. Users can be vulnerable by being enticed to download and install these infected applications.

Figure 3.1 illustrates this technique.

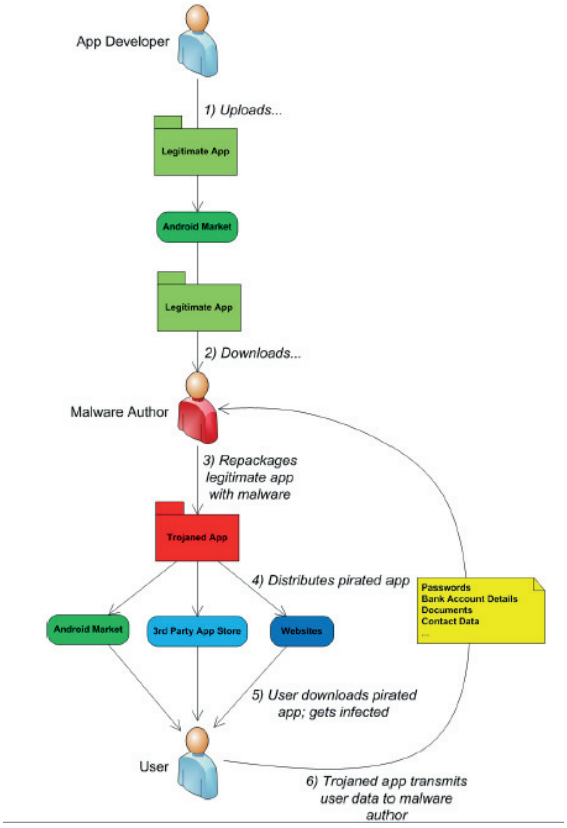


Figure 3.1: Repackaging process

3.3.2 Update Attack

Malware developers insert a special upgrade component into a legitimate application allowing it to be updated to a new malicious version, unlike the first technique that typically piggybacks the entire malicious payloads into applications.

3.3.3 Drive by Downloads

The ability to install and to download applications outside the official marketplaces allows malware developers to mislead users to download and easily install malicious applications. It is a class of techniques where a Web page automatically starts downloading an application when a user visits it. Drive by downloads can be combined with social engineering tactics to appear as if they are legitimate. Because the browser does not automatically install downloaded applications on Android, a malicious Website needs to encourage users to open the downloaded file for actually infecting the device with malware.

Android malware is also characterised by its payloads and mechanisms to launch malicious actions, during the infection phase. GGTracker is an example.

3.3.4 Remote Control

Malware authors aim to access the device during the infection phase remotely. Zhou and Jiang noted that 1.172 samples (93.0%) turn the infected phones into bots for remote control during their analysis.

3.3.5 Premium Rate SMS

Premium rate SMS is an important way for people to charge purchases to their phone bills. SMS billing is used by many legitimate services due to its ease-of-use as a phone payment mechanism; however, malware can also use premium rate SMS messages to steal money. Premium rate SMS is a technique in which malware requires SMS permissions to exploit the capability of sending messages to premium rate numbers (as shown in Figure 3.2). Malware intercepts any SMS messages from the SMS service to prevent a user from becoming aware of the charge. The Rufraud Trojan [Symantec 2011] pretended to be free versions of popular applications, and once installed on the user's device it would send SMS messages to a premium rate number determined by the country where the phone is located. GGTracker is one example of this threat.

Figure 3.2: A version of GGTracker

3.3.6 Phishing Scams

Malicious applications use Web pages or other user interfaces designed to lure a user to give unconsciously his sensitive information such as account login information to a malicious party posing as a legitimate service. As noted by Felt and Wagner [Felt/Wagner 2011] the small screen on mobile devices makes it in some cases harder for a user to identify whether it is spoofed. Fake applications pretending to be banking applications, which would steal the user's login information when used to access the bank, are reported [Travis Credit Union

2009].

3.3.7 Browser Exploits

These exploit vulnerabilities in Web browsers or applications that can be launched via a Web browser such as a Flash player, PDF reader, or image viewer in order to install malware.

3.3.8 Capability Leaking

According to the Woodpecker project [Grace/Zhou/Wang 2012] applications are leaking access to privileged device features, providing other applications with access to features they should not have access to. These applications expose restricted features through less restricted interfaces. As an example, a flaw was discovered in the Power Control widget [Ccjernigan 2013], which is standard on Android devices. This flaw leaked access to interfaces on the widget, allowing applications that did not have access to these features to toggle features like the GPS on and off.

3.3.9 Information Leaking

Information leaks expose sensitive data to other applications on the device, like capability leaks. This can be due to storing sensitive data in unprotected areas, as demonstrated by Brodeur [Brodeur 2013], or the application gives the information to anyone who knows how to ask. The logging tool HTC installed on the handsets was an example of this [Russakovskii 2011]. This logging tool exposed large amounts of private data to anyone requesting it using a simple HTTP request, without any validation on whether they should have access to this information. Another source of information leaking is the *READ_LOGS* permission [Lineberry/Richardson/Wyatt 2010]. This permission allows the application to access the system logs, depending on the applications that run on the device. It can provide the application with access to information equivalent to the *GET_TASKS*, *DUMP* and *READ_HISTORY_BOOKMARKS*. Additionally, third party applications were seen writing information usually restricted to *ACCESS_COARSE_LOCATION*, *ACCESS_FINE_LOCATION*, *READ_SMS* and *READ_ CONTACTS* to the system logs, providing equivalent access to these resources as well.

3.3.10 Threats Based on Vulnerabilities

Section 2.4.4 describes some vulnerabilities and flaws that exist on the Android platform. Malicious developers can exploit them causing various damages such as using root rights to access sensitive information. It is important to note that applications in this class of threats can be developed for no malicious actions but apparently access more sensitive data than required by it to perform tasks, not respecting the PLP.

3.4 Actual Attacks on Android

A variety of attacks have been reported on Android since its introduction showing the deficiencies of its security framework.

3.4.1 Privilege Escalation Attacks

The privilege escalation attacks refined by [Fang/Weili/Yingjiu 2014] as permission escalation attacks allow a malicious application to collaborate with other applications to access critical resources without requesting for corresponding permissions explicitly [Davi/Dmitrienko/Sadeghi /Winandy 2011], [Bugiel et al. 2011b], [Felt et al. 2011c], [Marforio/Ritzdorf/Francillon/Capkun 2012]. It is based on the fact that a malicious (unprivileged) application can take advantage of benign (privileged) applications to perform malicious actions. Figure 3.3 shows the categorisation of the permission escalation attack. It can be classified into two prominent categories: A confused deputy attack and a collusion attack.

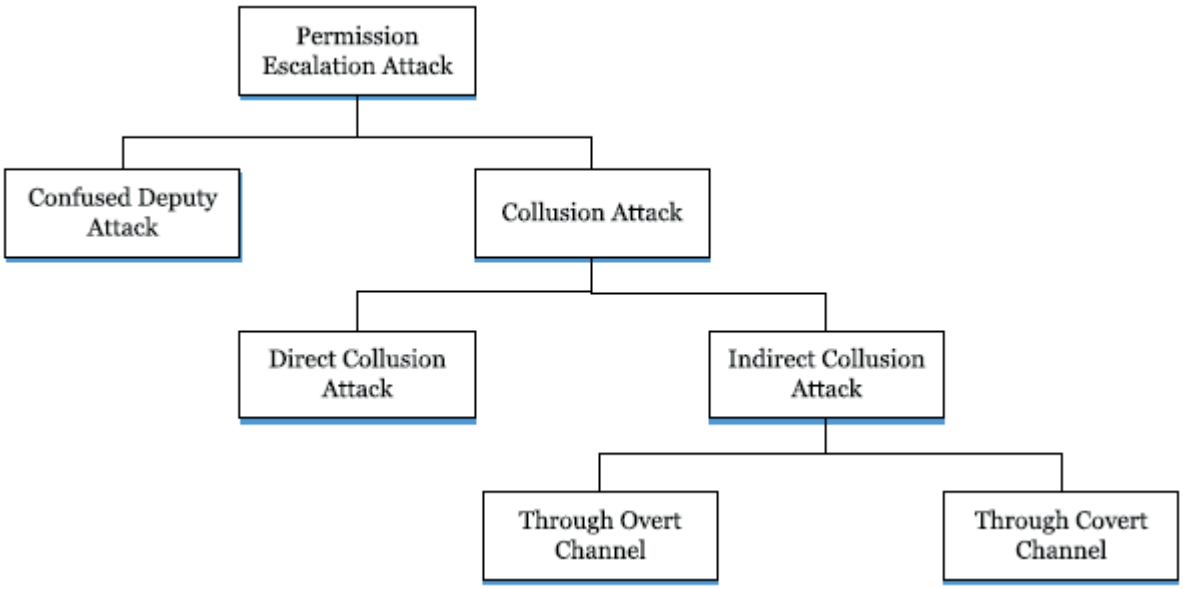


Figure 3.3: Categorisation of the permission escalation attack.

The confused deputy attack concerns scenarios where a malicious application exploits the vulnerable interfaces of another privileged but confused application [Dietz et al. 2011]. The application $A1$ is not granted with the permission $P1$ as shown in Figure 3.4; $C1$, a component of $A1$, cannot directly access the system resource $R1$ protected by permission $P1$.

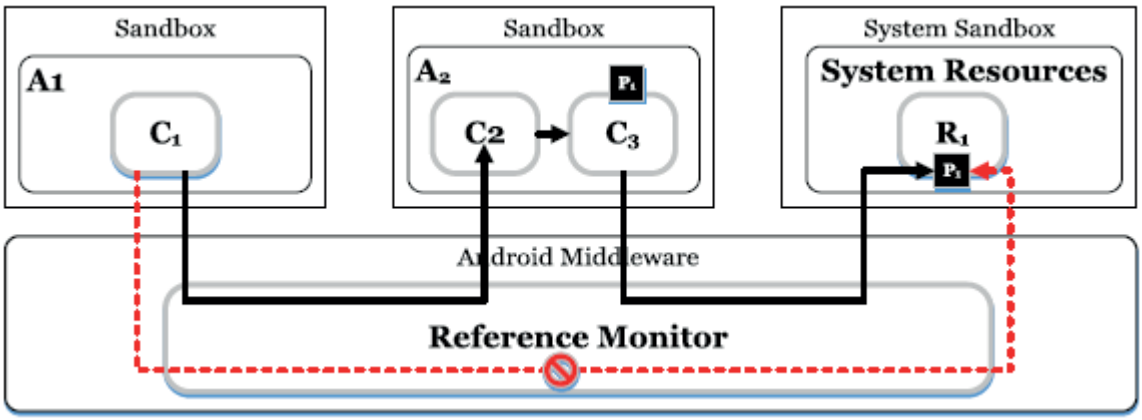


Figure 3.4: Permission escalation attack (modified from [Bugiel et al. 2011b], p. 3).

However, $C1$ can access $R1$ transitively if application $A2$ is granted with the permission $P1$ and one of $A2$'s component. $C2$ does not require any permission to be accessed. As a result, $C1$ can access $R1$ through $C2$ and $C3$ [Bugiel et al. 2011b].

Collusion attacks rather concern malicious applications that collude to combine their permissions, allowing them to perform actions beyond their individual privileges [Schlegel et al. 2011]. Figure 3.5 illustrates a scenario in which a user is tricked by a malevolent developer (MD) into installing two separate applications that seem to have little risk associated with them. For example, a camera application not requiring the Internet permission seems to be safe, as it cannot upload the pictures to the Internet.

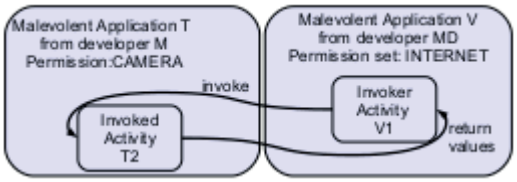


Figure 3.5: Permission collusion by malevolent applications

Similarly, a music streaming application not requesting the Camera permission would be acceptable. However, if the two applications are malicious, the music streaming application can invoke the camera application and send the pictures obtained from it remotely. The Android security system does not inform the user about this application collusion risk. Note that the camera application can include checks on the identity of the caller, such that it returns the pictures only to its music streaming collaborator, which allows colluding applications to pass a dynamic security analysis that invokes all possible activities and checks the returned information.

The collusion attack can be further classified by the way applications communicate with each other, into direct collusion attack, where applications communicate directly [Marforio/Francillon/Capkun 2011], and indirect where applications communicate via a third application or component in between [Bugiel et al. 2011b].

3.4.2 TOCTOU attack

The vulnerability of Time of Check to Time of Use (TOCTOU) exists in Android mainly due to naming collusion [Fang/Weili/Yingjiu 2014]. No naming rule or constraint is applied to a new permission declaration [Shin et al. 2010]. Any two permissions with the same name string are treated as equivalent even if they belong to unrelated applications [Fragkaki/Bauer/Jia/Swasey 2012]. Malicious application developers may exploit this flaw. Suppose a malicious developer manages luring a user to install a malicious application A which declares permission P', and another malicious application B, which requests permission P'. The name of permission P' is the same as permission P, which protects access to a critical resource. The user uninstalls application A afterwards and installs the benign application C, which declares permission P. Now the malicious application B would be able to use permission P' to access the critical resource. This TOCTOU flaw exists in Android 1.5, 1.6, 2.0, and 2.1, on both emulators, and on actual devices according to Shin et al. Fragkaki et al. claimed to have reproduced it in Android 2.3.7 [Fragkaki/Bauer/Jia/Swasey 2012]. This vulnerability cannot be thwarted using protection levels in the current Android permission system, unfortunately.

3.4.3 Attacks on Misconfigured Applications

These attacks are possible when an attacker application installed on the device can exploit the flows of a misconfigured application on encapsulation of components. If an application has any one of the configuration combinations listed in Table 3.1 as high risk, then any application on the device can spawn it.

Activity configuration		Application configuration	Consequence	
Exported	Intent-filter	SharedUid	Callers accepted	Risk level
exported=true	present	any	any	HIGH
exported=false	present	any	any	HIGH
default	present	set	from the same developer	LOW
default	absent	set	from the same developer	LOW
exported=true	present	any	any	HIGH
exported=false	present	set	from same developer	LOW

Table 3.1: Different configurations lead to different levels of vulnerability.

If an activity is listed in the application Manifest with an intent-filter and is not accompanied by a exported= false attribute, any other application on the system can invoke it. In absence of declarative or dynamic permission checking by the developer, information

returned to the caller through the intent result may then compromise permission-protected information, as no permission is required from the caller.

The user installs the malevolent application *B* (a music streaming application) with the permission to access the Internet in Figure 3.6.

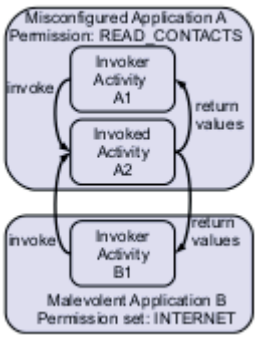


Figure 3.6: Internal activity invocation or confused deputy.

B can exploit the honest but misconfigured contact manager application A by invoking activity *A*2 that returns the contacts; *B* can then send the contacts to a remote server. If *A*2 is built to reply to external requests and it just fails to check that B has the proper permission, then the attack is a classic confused deputy attack described previously. However, because intents are also used as an internal (intra-application) communication mechanism in Android, it is possible that *A*2 is not built for communicating with another application and is just misconfigured. Table 3.1 shows the combination of configuration parameters that may lead to information leaks.

3.4.4 Attack on Applications Sharing the User ID

The attack aims to compromise a benign application through some vulnerability by exploiting a compromised application of the same developer. As shown in Figure 3.7, the vulnerability allows an attacker to access the information protected by permissions of the application *M*, which is configured to share the UID with the application *N* of the same developer. Some prerequisites are to be respected to set up this attack: an attacker would need to control the application *M* of the developer *D*, *M* should have any exfiltration permission (*SMS* or *INTERNET*) and *M* should share the *UID* with an application belonging to *D*.

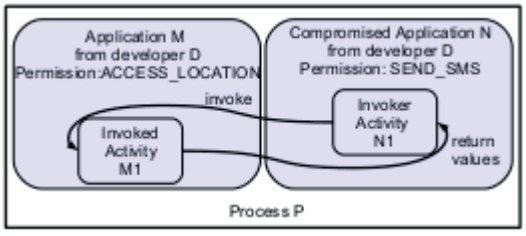


Figure 3.7: Application sharing the user ID with a compromised application.

3.4.5 UI Takeover Attack

The idea of the UI takeover is to intercept all key presses and to keep the current application in the foreground. Thus, the device is effectively blocked as there can only be one activity running at a time. Tchakounté and Dayang [Tchakounté/Dayang 2013b] shows with proof of concept that Task Killer Pro and Super History Eraser use this technique. The user has just to click somewhere on the screen to launch malicious payloads.

3.4.6 Zero Permission Attacks

[Lineberry/Richardson/Wyatt 2010] and [Lineberry/Richardson/Wyatt 2010a] discovered vulnerabilities on zero permissions in 2010. Every application normally requests its permissions during installation and the user has to accept them. Applications that request no permission are instantly downloaded and installed from the market without any warning. This kind of application seems harmless because the application cannot access any sensitive data on the phone, as no permission is required. [Lineberry/Richardson/Wyatt 2010] showed in their work that this assumption is false and that such an application is able to act maliciously and harmfully. We illustrate how zero permission application can access the Internet and reboot the smartphone with the following two scenarios.

Gain Internet access: An application normally needs the *INTERNET* permission to establish a bidirectional communication between the smartphone and a server on the Internet or, it uses another application which has the permission to access the Internet and is accepting intents. In this approach, the zero-permission application is sending a Uniform Resource Locator (URL) to the Android browser application via an intent. The browser then transfers the received URL and all appended parameters with a HTTP GET to the remote server. For the other sense from the server, we define a custom receiver for Uniform Resource Identifiers (URI) in the manifest.xml with the aim to start our zero permission application every time the browser connects to a server with our predefined scheme in the URL and transfers all parameters in the URL to our application. With this two-step approach our application is able to send and to receive data from a remote server without the necessary permission.

Reboot the device: The application has to request the REBOOT permission during installation for rebooting an Android smartphone. This permission is normally not granted to an application because the protection level is SystemOrSignature, even if the user would accept this permission. A zero-permission application is therefore not able to reboot the device. There are however several approaches to reboot a device without this permission. One approach shown by [Lineberry/Richardson/Wyatt 2010a] uses Toast notifications, a message that pops up on the surface of the window [Trend Micro 2015], crashing the system server which results in a reboot of the smartphone.

3.4.7 Reverse Shell Execution

This vulnerability is closely linked to the WebKit. The WebKit is an open source browser engine that is used by Android's browser application. The weak point of the WebKit is an

input validation issue in its JavaScript core. The *parseFloat()* operation, which parses a string to a floating point number, triggers a buffer overflow. The attacker can trigger this vulnerability when a smartphone user is visiting a manipulated Website. The attacker can open a remote shell with the permissions of the browser application, if the correct shell code is available on the malicious Website. This means that the attacker is able to browse and upload any data located on the Secure Digital (SD) card.

[Vidas/Votipka/Christin 2011] specify attacks to the device based on full privileged device access and on those without privileged access.

3.4.8 Unprivileged Attacks

Misleading the user may be required using social engineering techniques or permission escalation attacks to infiltrate the device in this category. An attacker may rely on convincing the user to install a malicious application when turning to privileged access. Such an application may present an enticing feature to the user but contains software that executes a privilege escalation attack. Oberheide [Oberheide 2010] demonstrated such an attack. Oberheide's seemingly benign application received more than 200 downloads within 24 hours. This application would routinely make remote requests for new payloads executing in the background. A remote attack may not even require the installation of a new application. Android's use of commodity software components, such as the Web browser and Linux base can be leveraged for an attack that requires no physical access [Shabtai et al. 2010c]. A concrete example of such an attack was deployed for the penetration-testing tool CANVAS 6.65 by Immunity Incorporation [Canvas 2015]. The user visits a malicious Website using the device's built-in browser in this attack. The attacker then uses this request taking advantage of the vulnerability in the WebKit browser [Woods 2010] to obtain remote shell access to the device with the level of privileges given to the browser. The attacker can then copy a Linux privilege escalation exploit to an executable mount point on the device, run the secondary exploit, and gain privileged access to the whole device.

3.4.9 Physical Access with ADB Enabled

An attacker may obtain privileged access to a device that has the Android Debug Bridge (ADB) enabled through physical access [Android Forums 2010]. Given physical access, an attacker can easily determine if ADB is enabled, by connecting the device via USB and by executing *adb get-serialNo* on the attached computer. The ADB is enabled, if the serial number of the device is returned. A privilege escalation requires the attacker to use ADB's push feature to place an exploit on the device and to use ADB's shell feature to execute the exploit and to gain privileged access, once the attacker knows that ADB is enabled. However, if the device is not password-protected, the attacker can simply interact with the common device interface and enable ADB. An example usage of this method is the Super One-Click desktop application [Waqas 2010]. Super One-Click requires users to enable first ADB debugging on the device. The main advantage of ADB-based attacks is the minimal observable footprint left on the device: No new application needs to be installed and a

reboot is also not necessary. The lack of device modification in this method makes it much harder to trace than other attacks, and is unlikely to be detected by security applications on unrooted devices.

3.4.10 Physical Access without ADB Enabled

An attacker may still take advantage of the device's recovery mode on an obstructed device, which does not have ADB enabled [Vidas/Zhang/Christin 2011]. Since the attack does not rely on software vulnerability to a particular version of Android, the attack has more longevity than other exploits such as the WebKit and Linux exploits mentioned above. The deployment method is specific to a device leading to extensive fragmentation based on the device model and/or the manufacturer. The attacker must create first a customised recovery image to use the recovery mode. The main modification necessary for this image is to the *init.rc (*it is the primary configuration file for the *init* process*)* and to the *default.prop (*it is a file to store persistent properties of the *init* process*)* files in the *initrd* section of the image. To give the attacker the necessary privileges, the init.rc file must list the executables that they wish to add with the rights necessary to be executed. Any executable necessary for the attack must also be added to the initrd section of the image.

Once the image is built and the attacker is able to gain physical access to the device, the attacker must then attach the device to a computer through a USB connection and run a manufacturer specific tool to flash the image to the recovery partition of the phone.

After the device has been flashed, the attacker then can access the recovery image using a device specific key combination (e.g. power button while holding X for Motorola Droid). When the device boots into the recovery partition, the *init.rc* file is executed. The *init.rc* file can be modified to run any malicious code that the attacker has added to the recovery image, such as auto-installing a rootkit without attacker interaction. The attacker can alternatively update the *default.prop* file to enable ADB, crafting *init.rc* to give executable rights to a Set User (Su) executable (added previously to the custom recovery image initrd). When the recovery image loads, the attacker opens an interactive shell on the device using ADB. He can now simply execute the Su executable to gain root access.

3.4.11 Physical Access to Unobstructed Device

In some cases the attacker may actually have access to a device without a password protected screen lock. Such a situation allows the attacker to actually leverage any other attack method since he can choose to install applications, to visit malicious Websites, to enable ADB on the device, etc.

3.5 Families of Malware

Public access to known Android malware samples is mainly provided via the Android Malware Genome Project[1] and Contagio Mobile[2]. The Malgenome Project was a result of the work done by Zhou and Jiang [Zhou/Jiang 2013]; it contains over 1200 Android malware samples, classified in 49 malware families. They were collected in the period of August 2010 to October 2011. Contagio is a Web platform offering an upload dropbox to share mobile malware samples among security researchers; it currently hosts several items. Families of malware that have been detected until the end of October 2012 are listed with their functionality and their vector threat in [Spreitzenbarth 2013].

3.6 Types of Malware Analysis

Given the complexity of an Android application package and the enormous growth of Android malware, security researchers and vendors must analyse a raising number of applications in a given period to understand the purpose of the software and to develop countermeasures. The analysis was done manually by using tools like decompilers and debuggers until recently. This process is very time consuming and error-prone depending on the skill set of the analyst. Therefore, tools for automatic analysis of applications have been developed to be used in the discovery and assimilation phases. According to Bergeron et al. [Bergeron et al. 2001] one can classify two types of automated analysis: Static and dynamic analysis, which we describe in the following sections. Additionally, we describe the Manifest analysis, an approach that we use in our work.

3.6.1 Static Analysis

The classical approach for automated analysis is static analysis. Static analysis considers software properties that can be investigated by inspecting the downloaded application and its source code only. Within this type of analysis tools often decompile or disassemble application packages and search for specified patterns like permissions, function calls and hard-coded variables throughout the source-code. All these techniques have in common that the suspicious application is not being executed. Signature-based detection of applications, the common approach by antivirus technologies, is an example of static analysis. Static analysis is carried out using reverse engineering techniques presented in section 2.2.1.

According to Moser et al. [Moser/Kruegel/Kirda 2007] the big advantage of static analysis is that this technique is relatively simple and fast. One of its disadvantages is that all patterns have to be known in advance, making it hard to detect new malware or polymorphic code without the use of a human expert or machine-learning techniques. Malware uses obfuscation techniques in practice to make static analysis even harder. A particular form of obfuscation used by Android applications is to hide system activities by calling functions

[1] http://www.malgenomeproject.org
[2] http://contagiominidump.blogspot.nl

outside the Dalvik runtime environment, i.e., in native libraries written in C or in other programming languages.

3.6.2 Manifest Analysis

This method focuses on the analysis of information of the Manifest file [Enck/Ongtang/McDaniel 2009], [Barrera/Kayacik/Oorschot/Somayaji 2010]. The purpose is to check permissions required by the application, or components encapsulation. Policy rules are applied after every control. It is a case of static analysis.

3.6.3 Dynamic Analysis

Dynamic analysis does not inspect the source code as in static analysis, but rather inspects the behaviour of the application during its execution life. It executes the suspicious application for that within a controlled environment, often called sandbox. Bishop ([Bishop 2002], p. 444]) defines such a sandbox as "*an environment in which the actions of a process are restricted according to a security policy*". A report is automatically generated for the analysis by monitoring and logging every relevant operation of the execution, like code blocks and function calls, as well as system calls. An analyst can identify outgoing data and modifications on the hard-drive or network traffic made by the application out of this report.

3.6.4 Comparison between Static, Manifest, and Dynamic Analysis

Table 3.2 gives an overview of the motivations using a specific method of analysis, based on [Moser/Kruegel/Kirda 2007]. It is more strengthen combining permission and dynamic analysis to find and prevent detection mechanisms and simultaneously track function calls and variables even if they are obfuscated. The analysis of the Manifest file is the approach used in our system.

Analysis type	Advantages	Disadvantages
Static	This technique is relatively simple and fast.	If a malware author is hiding the malicious code-patterns inside obfuscated blocks, a static analysis will fail classifying the application as malware.
Dynamic	A dynamic analysis can succeed to classify the application as malware if a malware author is hiding the malicious code-patterns inside obfuscated blocks.	There are several detection methods that a malware author can implement to notify if the application is running inside such a sandboxed environment. Malicious applications often try to act like benign ones or use timers to delay malicious behaviour and so hoping to run this code after the analysis period. This type of analysis consumes a lot of resources, often unavailable in smartphone.
Manifest	The Android system relies only on information, which is included in the Manifest file, such as permissions. Therefore, the process to analyse applications in this case is easier and faster.	This detection scheme focuses on fixed items and will not be dynamically applicable.

Table 3.2: Comparison of type of analysis

3.7 Tools for Malware Detection

While malware tries to conceal its presence and its actions, users try to find it and protect themselves. To help users in the task, free and paid tools are available to them. Three tools are commonly used for this purpose in discovery, assimilation and destruction stages: Firewalls, Intrusion Detection Systems (IDS) and antivirus software. Their common mission is to track down and to eliminate potential malicious applications.

3.7.1 Firewall

A firewall is a barrier that protects information from a device or network when establishing communication with other networks, e.g. the Internet. Its purpose is to protect the purity of the devices on which they are installed by blocking intrusions orchestrated from the Internet. Several benefits are associated with their use. First, they are well known solutions. They are also extensively used on other platforms (PC and server). And finally, they are very effective

because they take advantage of the maturity gained by firewalls on PCs. A disadvantage is, that they are ineffective against attacks on the browser, bluetooth, e-mail, SMS, and MMS. They are used as modules in antiviruses on Android.

3.7.2 Intrusion Detection Systems

An Intrusion Detection System (IDS) represents a set of software and hardware components whose main function is to detect abnormal or suspicious activities on the analysed target: A network or a host. This is a family of tools of many types: IDS, Host Intrusion Detection System (H-IDS), Network Intrusion Detection System (NIDS), IDS hybrid, Intrusion Prevention System (IPS), and Kernel IDS / IPS Kernel (K-IDS / IPS-K). IDS has two major advantages. First, it is able to detect new attacks, even those that seem isolated. Second, it can be easily adapted to any task. It generates unfortunately a high consumption of resources and a high false alarm rate. Andromaly [Shabtai et al. 2012] and Crowdroid [Burguera/ Zurutuza/Nadjm-Tehrani 2011] are examples of an IDS dedicated to detecting malware on the Android platform. Crowdroid is specifically designed to recognise Trojans.

3.7.3 Antiviruses

Antiviruses are security software mostly used on smartphones. The popularity gained by their counterparts on desktop has greatly contributed to increase the level of confidence acquired by mobile users. Avast, AVG and F-Secure are examples of renowned antiviruses. They are facing new constraints brought by the rapid evolution of malicious applications. Like desktop platforms, their efficiency is closely related to their detection methods. Filiol [Filiol 2005] classifies these methods in three families: Form analysis, integrity checking, and dynamic behaviour analysis.

1. Form analysis is detecting the presence of a threat in an application by static characters. It can be based on research of signatures, heuristics or spectral analysis.

    - Research of signatures: Searches for patterns or bits, which are characteristics of a known threat. Its main disadvantage is that it is not able to detect unknown threats and known threats, which are modified. It requires a permanent update of the signature database. It is simple to implement and the most used in antivirus companies [Zhou/Jiang 2012].
    - Spectral analysis: Scrutinises statements commonly used by malware samples but rare in normal applications. It analyses the frequency of such statements statistically to detect unknown threats. This approach is subject to false positive, i.e. normal applications, which are incorrectly classified as malware.
    - Heuristic analysis: Its approach is to establish and maintain rules, which are used as pattern to recognise malicious applications. It is also subject to false alerts, as the previous approach.

2. Integrity checking is based on the evidence that abnormal modifications of a file can reveal contamination by dangerous code. Dynamic behaviour analysis is used to scrutinise the actions of an application when it is running.

3. The third method detects suspicious actions such as attempting to modify data of another application or to modify libraries and memory space reserved for the system.

The subject of this thesis is to build a system, which uses the form analysis.

Chapter 4

Concepts on Machine Learning for Malware Detection

The rapid growth of the Android platform involves a pressing need to develop effective solutions. However, our defence capability is largely constrained by the limited understanding of the emerging malware and the lack of timely access to related samples. Moreover, Zhou and Jiang [Zhou/Jiang 2012] showed that malware is rapidly evolving and existing anti-malware solutions are seriously becoming ineffective. For instance, it is not uncommon for Android malware to have encrypted root exploits or obfuscated C&C servers. The adoption of various sophisticated techniques greatly raises the bar for their detection. Conventional security measures relying on the analysis of security incidents and attack development inherently fail to provide a timely protection. As a consequence, users often remain unprotected over longer periods of time.

The field of machine learning has been considered as an ideal match for these problems, as learning methods are able to automatically analyse data, provide timely decisions and support early detection of threats. Much work on mobile security based on this approach has produced interesting results.

The concepts around machine learning are described in this chapter at first using a specific type in the thesis: the classification learning. We secondly illustrate metrics to evaluate the classification and lastly we investigate works concerning Android security in the literature, specifically on detecting malware based on permissions with classification learning techniques. This investigation is ended with a discussion on limitations and enhancements for this type of detection.

4.1 Machine Learning

The concept of learning can be described in many ways including acquisition of new know ledge, enhancement of existing knowledge, representation of knowledge, organisation of knowledge and discovery of facts through experiments [Michalski/Carbonell/Mitchell 1983]. This approach can be used to acquire knowledge from malware and good software in our case. A learning task may be considered as the estimation of a function with sets of inputs

and outputs.

When such learning is performed with the help of computer programs, it is referred to as machine learning. Depending on the way in which the knowledge may be represented, machine learning may be divided into decision trees, neural networks, probability measures or other representations. However, as identified in [Dietterich/Langley 2003], a more fundamental way to distinguish machine learning is on the basis of the input type and the way in which the knowledge is used. This division consists of:

- Learning for classification and regression: This is the most widely used method of learning. Classification consists of assigning a new instance into one of the fixed classes from a finite set of classes. The learning scheme is presented with a set of classified examples for that from which expected learning of classifying unknown instances. Regression involves the prediction of the new value on the basis of some continuous variable or attribute.

- Learning for acting and planning: In this case, the knowledge is used for selecting an action for an agent. The action may be chosen in a purely reactive way, ignoring any past values. Alternatively, the agent may use the output of classification or regression to select an action based on the description of the current world state. These approaches are useful for problem solving, planning, and scheduling.

- Learning for interpretation and understanding: This type focuses on the interpretation and understanding of situations or events rather than just the accurate prediction of new instances. Many separate knowledge elements are used to derive this understanding, which is known as abduction.

4.1.1 Definition of Concepts

Machine learning requires introducing a number of definitions, which will be used later.

4.1.1.1 Datasets

A set of data items, the dataset, is a very basic concept of machine learning. A dataset is roughly equivalent to a two-dimensional spreadsheet or database table. A dataset is a collection of examples, each instance consisting of a number of attributes.

- Training dataset (or set): This is the sample of items or records (training items) used to determine rules to acquire knowledge for its items after the learning process.

- Testing dataset (or set): This is a set of items or records (testing items) disjoint from the learning dataset. It is used to evaluate the capacity of the knowledge to classify unknown instances.

4.1.1.2 Attributes and Classes

Each instance that provides the input to machine learning is characterised by its values on a fixed, predefined set of features or attributes. The instances are the rows of the table and the attributes are the columns. It is presented like this for space reasons. They are generally in numeric (both discrete and real-value) or nominal form. Numeric attributes may have continuous numeric values whereas nominal values may have values from a pre-defined set. For instance, an attribute like temperature, if used as a numeric attribute, may have values like 25 degrees celsius, 28 degrees celsius, etc. It may take values from a fixed set (such as high, medium, low), on the other hand, if it is used as a nominal attribute.

The input data for a classification task are formally a collection of records. Each record, also known as an instance or example, is characterised by a tuple (x, y), where x is the attribute set and y is a special attribute, designated as the class label (also known as category, target attribute or output). Table 4.1 shows a sample dataset used for classifying vertebrates into one of the following categories: mammal, bird, fish, reptile, or amphibian. The attribute set includes properties of a vertebrate such as its body temperature, skin cover, method of reproduction, ability to fly, and ability to live in water. Although the attributes presented in Table 4.1 are mostly discrete, the attribute set can also contain continuous features.

The class label, on the other hand, must be a discrete attribute. This is a key characteristic that distinguishes classification from regression, a predictive modelling task in which y is a continuous attribute.

Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hibernates	Class Label
human	warm-blooded	hair	yes	no	no	yes	no	mammal
python	cold-blooded	scales	no	no	no	no	yes	reptile
salmon	cold-blooded	scales	no	yes	no	no	no	fish
whale	warm-blooded	hair	yes	yes	no	no	no	mammal
frog	cold-blooded	none	no	semi	no	yes	yes	amphibian
komodo	cold-blooded	scales	no	no	no	yes	no	reptile
dragon								
bat	warm-blooded	hair	yes	no	yes	yes	yes	mammal
pigeon	warm-blooded	feathers	no	no	yes	yes	yes	mammal
cat	warm-blooded	fur	yes	no	no	yes	no	mammal
leopard	cold-blooded	scales	yes	yes	no	no	no	fish
turtle	cold-blooded	scales	no	semi	no	yes	no	reptile
penguin	warm-blooded	feathers	no	semi	no	yes	no	reptile
porcupine	warm-blooded	quills	yes	no	no	yes	yes	mammal
eel	cold-blooded	scales	no	yes	no	no	no	fish
salamander	cold-blooded	none	no	semi	no	yes	yes	amphibian

Table 4.1: Data for classifying vertebrates into one of the categories

4.1.1.3 The Classification Model

Classification is the task of learning a target function f that maps each attribute set *x* to one of the predefined class labels *y* (Figure 4.1).

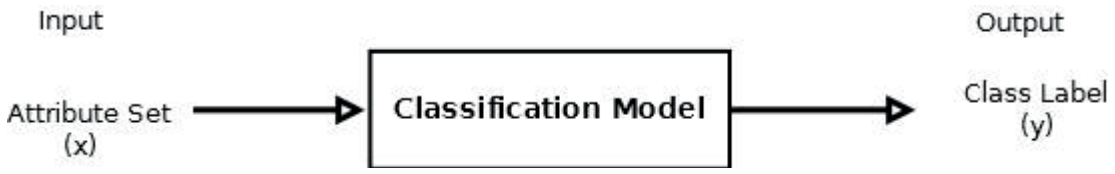


Figure 4.1: Classification as the task of mapping

The target function is also informally known as a classification model. A classification model is useful for the following purposes: a descriptive model and a prescriptive model.

1. Descriptive modelling: A classification model can serve as an explanatory tool to distinguish between objects of different classes. For example, it would be useful for

biologists to have a descriptive model that summarises the data shown in Table 4.1 and explains what features define a vertebrate as a mammal, reptile, bird, fish, or amphibian.

2. Predictive modelling: A classification model can also be used to predict the class label of unknown records. A classification model can be treated as a black box that automatically assigns a class label when presented with the attribute set of an unknown record. Supposed we have the following characteristics of a creature known as a Gila monster in Table 4.2.

Name	Body Temperature	Skin Cover	Gives Birth	Aquatic Creature	Aerial Creature	Has Legs	Hibernates	Class Label
Gila monster	cold-blooded	scales	no	no	no	yes	yes	**?**

Table 4.2: Determination of the class corresponding to an instance

One can use a classification model built from the dataset shown in Table 4.1 to determine the class to which the creature belongs.

Classification techniques are most suited for predicting or describing datasets with binary or nominal categories. They are less effective for ordinal categories (e.g., to classify a person as a member of high-, medium-, or low-income group) because they do not consider the implicit order among the categories. The remainder of this chapter focuses on binary classification based only on binary or nominal class labels.

4.1.1.4 Production of Knowledge

The way in which knowledge is learned is another important issue for machine learning. The learning element may be trained in different ways [Dietterich/Langley 2003]. For classification and regression, knowledge may be learned in a supervised, unsupervised or semi-supervised manner. Concerning supervised learning, the learner is provided with training examples with associated classes or values for the attribute to be predicted. Decision-tree and rule induction methods, neural network methods, the nearest neighbour approaches, and probabilistic methods are types of supervised learning. These methods differ in the way they represent the learned knowledge and also in the algorithms that are used for learning.

Unsupervised learning is concerned with the provision of training examples without any class association or any value for an attribute used for prediction. Clustering and density estimation are examples of unsupervised learning approaches. In case of clustering, the goal of learning is to assign training instances to classes of its own invention, which can then be used to classify new instances. Density estimation is used to build a model that predicts the probability of occurrence for specific instances.

A third approach, which is essentially between the two described above, is that of semi-supervised learning. In this type of learning, the set of training examples is mixed, i.e., for

some instances the associated classes are present, whereas for others, they are absent. The goal in this case is to model a classifier or regression coefficient that accurately predicts and improves its behaviour by using the unlabelled instances.

4.1.1.5 General Approach for the Classification Problem

A classification technique (or classifier) is a systematic approach to build classification models from an input dataset. Each technique employs a learning algorithm to identify a model that fits the relationship best between the attribute set and the class label of the input data. The model generated by a learning algorithm should both fit the input data well and correctly predict the class labels of records it has never seen before. Therefore, a key objective of the learning algorithm is to build models with good generalisation capability; i.e., models that accurately predict the class labels of previously unknown records.

Figure 4.2 illustrates a general approach for solving classification problems. First, a training dataset consisting of records whose class labels are known must be provided ([Tan/Stein- bach/Kumar 2005], p.148). The training dataset is used to build a classification model, which is subsequently applied to the test dataset, which consists of records with unknown class labels.

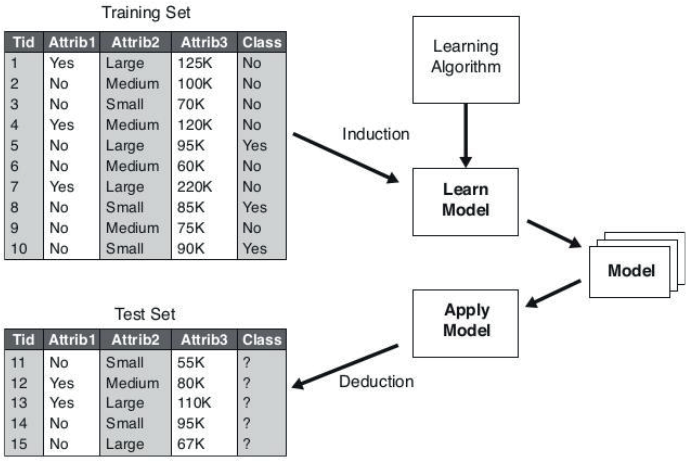


Figure 4.2: General approach for building a classification model

The life cycle of a machine learning task generally follows the process as depicted in Figure 4.3.

1. Choosing a learning algorithm;
2. Training the algorithm using a set of instances (referred to as the training dataset);
3. Evaluating the performance by running the algorithm on another set of instances (referred to as the testing dataset).

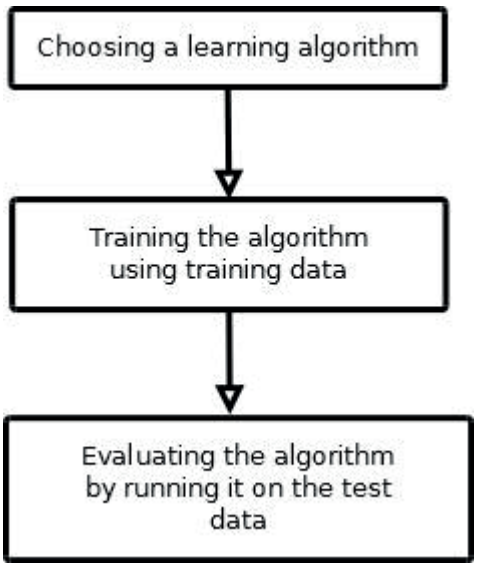


Figure 4.3: Machine learning flow [Sabnani 2008]

Different types of algorithms may be chosen at different times depending on the nature of the knowledge to be learned. Choosing an algorithm depends also on input and output type. Machine learning algorithms are described in section 4.1.1.8.

The next step is to train the algorithm by providing it with a set of training instances, once the algorithm is selected. The training instances are used to build a model that represents the target concept to be learned (i.e. the hypothesis). The main goal of this hypothesis formulation is to generalise the data to the maximum possible extent. This model is then evaluated using the set of test instances. It is extremely important to take care that the data instances used in the training process must not be used during the testing phase because it may lead to an overly optimistic model.

In order to enable the classification of previously unseen data in a test dataset on the basis of its attribute values, knowledge representation from the data in the training dataset is to be determined. Two major kinds of knowledge representation are used in learning: the decision tree and the classification rule.

4.1.1.6 Classification Rules

Rules are usually expressions of the following form [Grzymala-Busse 2010]:

if (attribute1, value1) and (attribute2, value2) and . . . and (attributen, valuen) then (decision, value).

or

(attribute1, value1) & (attribute2, value2) &..&(attributen, valuen)→ (decision, value).

Attributes are independent variables, decision is a dependent variable. A very simple example of such a table is presented as Table 4.3, in which attributes are: Temperature, Headache, Weakness, Nausea, and the decision is Flu. The set of all cases labelled by the same decision value is called a *concept*. For Table 4.3, case set $1,2,4,5$ is a concept of all cases affected by flu; (the corresponding value of Flu is yes for each case from this set).

Instance	Attributes				Decision
	Temperature	Headache	Weakness	Nausea	Flu
1	very high	yes	yes	no	yes
2	high	yes	no	yes	yes
3	normal	no	no	no	no
4	normal	yes	yes	yes	yes
5	high	no	yes	no	yes
6	high	no	no	no	no
7	normal	no	yes	no	no

Table 4.3: An example of a consistent dataset

Types of rules A case x is covered by a rule r if and only if every condition (attribute-value pair) of r is satisfied by the corresponding attribute value for x. The concept C defined by the right hand side of rule r is indicated by r. A concept C is completely covered by a rule set R if and only if for every case x from C there exists a rule r from R such that r covers x. A rule set R is complete if and only if every concept from the dataset is completely covered by R.

A rule r is consistent (with the dataset) if and only if for every case x covered by r, x is a member of the concept C indicated by r. A rule set R is consistent if and only if every rule from R is consistent with the dataset. For example, *case* 1 from Table 4.3 is covered by the following rule r:

(Headache, yes) $\rightarrow$ (Flu, yes).

The rule r indicates the concept $1,2,4,5$. Additionally, the concept $1,2,4,5$ is not completely covered by a rule set consisting of r, since r covers only *cases* $1,2,$ *and* 4, but the rule r is consistent with the dataset from Table 4.3.

On the other hand, the single rule (Headache, no) $\rightarrow$ (Flu, no) completely covers the concept $3,6,7$ in Table 4.3, though this rule is not consistent. The above rule covers cases 3, 5, 6, and 7.

Any of the set of the two rules:

(Headache, yes) & (Weakness, yes) $\rightarrow$ (Flu, yes)

(Temperature, high) & (Headache, yes) $\rightarrow$ (Flu, yes)

is consistent with the dataset from Table 4.3, but the concept $1,2,4,5$ is not completely covered by the rule set consisting of the above two rules since case 5 is not covered by any rule. The first rule covers *cases* 1 and 4, the second rule covers *case* 2.

They are meant to be interpreted sequentially when there are more than one rule: The first one; then if it does not apply, the second; and so on. A set of rules that are intended to be interpreted in sequence is called a decision list.

Classification rules are simple, straightforward, and easy to understand. The limitation of classification rules is that they cannot resolve conflicting information as shown in Table 4.4 for instances 7 and 8. If more than one rule has the same condition but specifies different classes for the instance, then additional attributes are required to resolve the contradiction

and to form a new knowledge rule.

Instance	Attributes				Decision
	Temperature	Headache	Weakness	Nausea	Flu
1	very high	yes	yes	no	yes
2	high	yes	no	yes	yes
3	normal	no	no	no	no
4	normal	yes	yes	yes	yes
5	high	no	yes	no	yes
6	high	no	no	no	no
7	normal	no	yes	no	no
8	normal	no	yes	no	yes

Table 4.4: An example of an inconsistent dataset

4.1.1.7 Decision Trees

A classification problem can be solved by asking a series of carefully crafted questions about the attributes of the testing record [Tan/Steinbach/Kumar 2005]. Supposed there exist only two categories: mammals and non-mammals. When scientists discover a new species, how can they determine whether it is a mammal or a non-mammal? The first question to be asked is whether the species is cold- or warm-blooded. If it is cold-blooded, then it is definitely not a mammal. Otherwise, it is either a bird or a mammal. In the latter case, we need to ask a follow-up question: Do the females of the species give birth? Those that give birth are definitely mammals, while those that do not are likely to be non-mammals. Each time one receives an answer, a follow-up question is asked until a conclusion is reached about the class label of the record. The series of questions and their possible answers can be organised in the form of a decision tree, which is a hierarchical structure consisting of nodes and directed edges. Figure 4.4 shows the decision tree for the mammal classification problem.

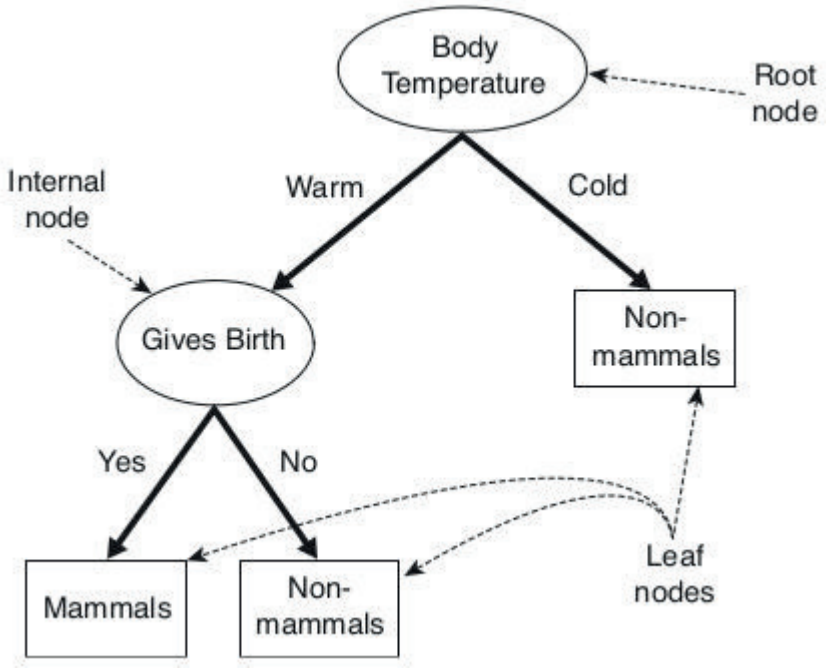


Figure 4.4: A decision tree for the mammal classification problem

The tree has three types of nodes:

- A root node that has no incoming edges and zero or more outgoing ones;
- Internal nodes, each of which has exactly one incoming edge and two or more outgoing edges;
- Leaf or terminal nodes, each of which has exactly one incoming edge and no outgoing one.

Each leaf node in a decision tree is assigned to a class label. The non-terminal nodes, which include the root and other internal nodes, contain attribute test conditions to separate records that have different characteristics. For example, the root node uses the attribute Body Temperature to separate warm-blooded from cold-blooded vertebrates. Since all cold-blooded vertebrates are non-mammals, a leaf node labelled non-mammals is created as the right child of the root node. If the vertebrate is warm-blooded, a subsequent attribute, Gives Birth, is used to distinguish mammals from other warm-blooded creatures, which are mostly birds.

Classifying a testing record is straightforward once a decision tree has been constructed. We apply the test condition to the record starting from the root node, and follow the appropriate branch based on the outcome of the test. This will lead either to another internal node, for which a new test condition is applied, or to a leaf node. The class label associated with the leaf node is then assigned to the record.

Figure 4.5 illustrates traces the path in the decision tree that is used to predict the class label of a flamingo. The path terminates at a leaf node labelled Non-mammals.

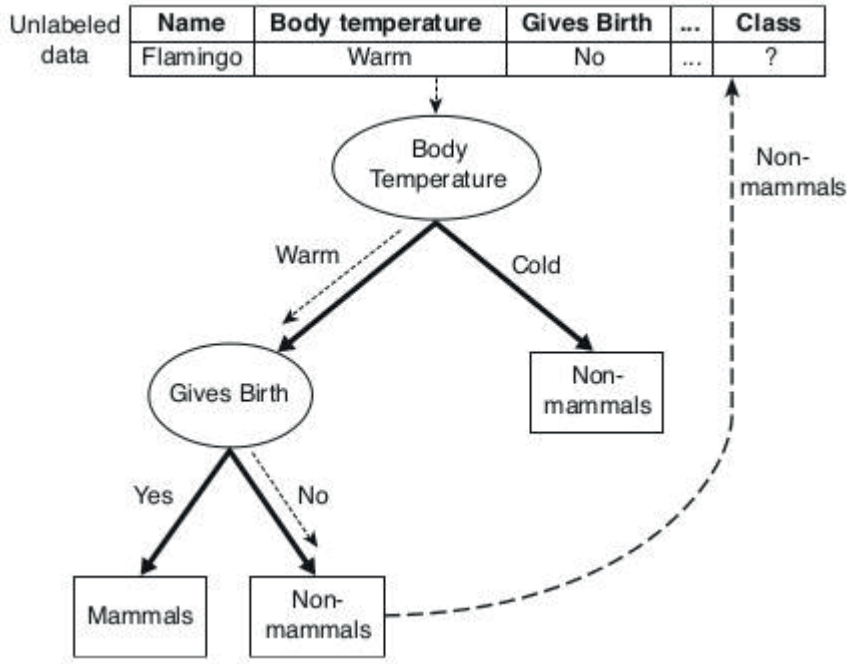


Figure 4.5: Classifying an unlabelled vertebrate

The dashed lines represent the outcomes when applying various attribute test conditions on the unlabelled vertebrate. The vertebrate is eventually assigned to the Non-mammal class.

4.1.1.8 Machine Learning Algorithms

This section presents many of the popular machine leaning algorithms grouped according to some classification techniques.

- Divide and Conquer: This approach defines classification rules uses a top-down, divide and conquer technique where the classification rules can be directly generated from the tree by traversing through the paths from the root to each leaf [Suh 2011]. This is an approach in classification learning based on using decision trees to induce classification information. The decision tree recursively selects attributes to test and to split the dataset into subsets according to the outcome of the test until a subset is obtained that contains instances of only one class. Different training methods are typically used for learning the graph structure of these models from a labeled dataset. As examples, RandomForest, an ensemble (i.e. a combination of classifiers) of different randomly-built decision trees [Breiman 2001], and J48, which implements the C4.5 algorithm [Quinlan 1993].

- Separate and Conquer: This approach generating classification rules uses a top-down separate and conquer or covering technique that takes each class in turn and directly induces a set of rules, each covering as many instances of the class as possible [Suh 2012]. This approach examines only one class at a time. It builds a rule for each class by selecting ones and adding tests to the rule until the complete subset of instances covered by the rule is pure (e.g., all members belong to only one class). The covered

subset of instances is then excluded from further processing. The rule-generation process continues until no more unclassified instances are left in the dataset. The advantage of this approach is time efficiency as a result of the following two characteristics: first, it creates knowledge rules directly without inducing an intermediate decision tree; secondly, it immediately excludes instances covered by a newly created rule from further induction. A variant of this approach is PART [Frank/Witten 1998], which is a class for generating a PART decision list. It uses the separate and conquer approach, builds a partial decision tree in each iteration and makes the best leaf into a rule. Another rule classifier is the Decision Table, classifier which builds and uses a simple decision table in which the output will show a decision on a number of attributes for each instance. The number and the specific types of attributes can vary to suit the needs of the task [Sheng/Ling 2005].

- Bayesian Networks: Bayesian Networks [Pearl 1982] are based on the Bayes Theorem and defined as graphical probabilistic models for multivariate analysis. They are specifically directed acyclic graphs that have an associated probability distribution function [Castillo/Gutiérrez/Hadi 1997]. Nodes within the directed graph represent problem variables (they can be either a premise or a conclusion) and the edges represent conditional dependencies between such variables. The probability function illustrates the strength of these relationships in the graph. The most important capability of Bayesian Networks is their ability to determine the probability of a certain hypothesis being true, e.g., the probability of an application of being malware. Naïve Bayes [Kohavi 1996] is an example of this approach and assumes the features being independent random variables and calculates their probabilities to draw a conclusion. It is a relatively fast algorithm, but the initial assumption that features are strongly independent is not always realistic in real world. A Naive Bayes respects three properties:

  - A Naive Bayes classifier assumes that the presence or absence of a particular feature of a class is unrelated to the presence or absence of any other feature, given the class variable. For example, a fruit may be considered to be an apple if it is red, round, and about 4" in diameter.
  - A Naive Bayes classifier considers all properties contributing independently to the probability that the fruit is an apple, even if these features depend on each other or upon the existence of the other features.
  - Naive Bayes classifiers can be trained very efficiently in a supervised learning style, depending on the precise nature of the probability model.

- Support Vector Machine (SVM): SVM algorithms divide the n-dimensional space representation of the data into two regions using a hyperplane. This hyperplane always maximises the margin between those two regions or classes. The margin is defined by the furthest distance between the examples of both classes and is computed based on the distance between the closest instances of both classes, which are called supporting

vectors [Vapnik 2000]. Instead of using linear hyperplanes, it is common to use the so-called Kernel functions. Kernel methods map input data into a higher dimensional vector space where some classification or regression problems are easier to model.

- Ensemble Methods: Ensemble methods are models composed of multiple weaker models that are independently trained and whose predictions are combined in some way making the overall prediction. The difficulty resides in determining the weak learners to combine and how to combine them. AdaBoost and RandomForest are examples in this model. AdaBoost [Freund/Schapire 1996] is an iterative classifier that runs other algorithms multiple times in order to reduce the error. For the first iteration all the algorithms have the same weight. As the iterations continue, the boosting process adds weights to the algorithms that reveal lower errors from the results of the classifier runs.

- K-Nearest Neighbours: The K-Nearest Neighbour (KNN) [Fix/Hodges Jr 1952] classifier is one of the simplest supervised machine learning models. This method classifies an unknown specimen based on the class of the instances closest to it in the training space by measuring the distance between the training instances and the unknown instance. Even though several methods exist in order to choose the class of the unknown sample, the most common technique is simply classifying the unknown instance as the most common class amongst the K-nearest neighbours. An object is classified by a majority vote of its neighbours, with the object being assigned to the most common class among its K-Nearest neighbours. Such methods typically build up a database of example data and compare the new data to the database using a similarity measure in order to find the best match and to make a prediction. This method is an instance of the so-called lazy algorithms [Raviya/Gajjar 2013] such as IBK [Aha/Kibler 1991].

4.1.2 Evaluation Metrics for Performance

The knowledge can be applied to the test dataset to predict the class labels of previously unseen records. It is often useful measuring the performance of the knowledge on the test dataset because such a measure provides an unbiased estimate of its generalisation error. The accuracy or error rate computed from the testing dataset can also be used to compare the relative performance of different classifiers in the same domain. The class labels of the test records are required to do this.

4.1.2.1 Performance of Classification Models

The evaluation of the performance of a classification model is based on the counts of testing records correctly and incorrectly predicted by the model. These counts are represented in a table known as a confusion matrix [Witten/Eibe/Hall 2011]. Table 4.5 depicts the confusion matrix for a binary classification problem. The confusion matrix is more commonly named contingency table. The number of correctly classified instances is the sum of the diagonals

in the matrix; all others are incorrectly classified; class yes gets misclassified as class no, and class no gets misclassified as yes in the example.

		Predicted Class	
		yes	*no*
Actual Class	*yes*	True Positive	False Negative
	no	False Positive	True Negative

Table 4.5: General form of the contingency table for a binary classification

- True Positive, False Positive, True Negative, False Negative Rate: The True Positive Rate (TPR) and the False Positive Rate $(FPR)$are performance metrics that are especially useful for imbalanced class problems. The TPR is the proportion of examples, which were classified as class X among all examples, which truly have class X, i.e. which class was captured. It is equivalent to Recall (REC). This is the diagonal element divided by the sum over the relevant row in the confusion matrix, i.e. $7/2 = 0.778$ for class yes and $2/(3+2) \doteq 0.4$ for class no in Figure 4.6.

  The FPR is the proportion of examples, which were classified as class X, but belong to a different class, among all examples, which are not of class X. This is the column sum of class x minus the diagonal element in the matrix, divided by the row sums of all other classes; i.e. $3/5 = 0.6$ for class yes and $2/9 = 0.222$ for class no. The True Negative (TN) rate is the number of examples, which are correctly predicted as negative when it is actually negative. The False Negative (FN) rate is the number of examples, which are incorrectly predicted as negative when it actually should be positive. The total number of positives is $TP+TN$ and the total number of negatives is $FP+TN$.

$$FPR = \frac{FP}{N} = \frac{FP}{FP+TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN+TP}$$

Where N is the total number of negatives and P the total number of positives.

```
=== Detailed Accuracy By Class ===

TP Rate  FP Rate  Precision  Recall  F-Measure  Class
0.778    0.6      0.7        0.778   0.737      yes
0.4      0.222    0.5        0.4     0.444      no

=== Confusion Matrix ===

a b   <-- classified as
7 2 | a = yes
3 2 | b = no
```

(a) Bad classifier

```
=== Detailed Accuracy By Class ===

TP Rate  FP Rate  Precision  Recall  F-Measure  Class
1        0        1          1       1          yes
1        0        1          1       1          no

=== Confusion Matrix ===

a b   <-- classified as
9 0 | a = yes
0 5 | b = no
```

(b) Perfect Classifier

Figure 4.6: Examples of contingency table

Summarising this information with numbers would make it more convenient to compare the performance of different models, although a confusion matrix provides the information

needed to determine how well a classification model performs. Both the prediction error (ERR) and accuracy (ACC) provide general information about how many samples are misclassified.

- Error, Accuracy*: Error* can be understood as the sum of all false predictions divided by the number of total predictions; *Accuracy* is calculated as the sum of correct predictions divided by the total number of predictions respectively.

$$Accuracy(ACC) = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions} = \frac{TP+TN}{TP+TN+FP+FN}$$

The performance of a model can be expressed in terms of its error rate. *ERR* is given by the following equation:

$$Error\ Rate(ERR) = \frac{Number\ of\ wrong\ predictions}{Total\ number\ of\ predictions} = \frac{FP+FN}{TP+TN+FP+FN} = 1-ACC$$

- Precision, F-measure: The *Precision* (PRE) is the proportion of the examples, which truly have class X among all those, which were classified as class X. It represents the diagonal element divided by the sum over the relevant column in the matrix, i.e. $7/(7+3) = 0.7$ for the class *yes* and $2/(2+2) = 0.5$ for the class *no*. The *F-measure* (often called *F1-score*) is a combined measure for *Precision* and *Recall*.

$$PRE = \frac{TP}{TP+FP}$$

$$REC = TPR$$

$$F1 = 2 \times \frac{PRE \times REC}{PRE+REC}$$

- Sensitivity, Specificity*: Sensitivity* (SEN) is synonymous to Recall and the True Positive Rate whereas *Specificity* (SPC) is synonymous to the *True Negative Rate*. *Sensitivity* measures the recovery rate of the Positives and complimentary, the *Specificity* measures the recovery rate of the Negatives.

- Receiver Operator Characteristic: Receiver Operator Characteristics (ROC) graphs are useful tools to select classification models based on their performance with respect to the False Positive and True Positive Rates [Raschka 2014]. The diagonal of a ROC graph can be interpreted as random guessing (line of no-discrimination); classification models that fall below the diagonal are considered as worse as random guessing. A perfect classifier would fall into the top left corner of the graph with a True Positive Rate of 1 and a False Positive Rate of 0. The ROC curve can be computed by shifting the decision threshold of a classifier. Based on the ROC curve, the so-called Area Under the Curve (AUC) can be calculated to characterise the performance of a classification model. AUC represents the probability that a randomly chosen malicious sample will be correctly classified [Hanley/McNeil 1982]. Hosmer et al. [Hosner/Lemeshow/Sturdivant 2013] propose the following guidelines for assessing the classification quality by the AUC value.

- $0.7 \leq AUC < 0.8$: is an acceptable discrimination;
- $0.8 \leq AUC < 0.9$: is an excellent discrimination;
- $AUC \geq 0.9$: is an outstanding discrimination.

Figure 6.7 represents an example of ROC with an excellent AUC.

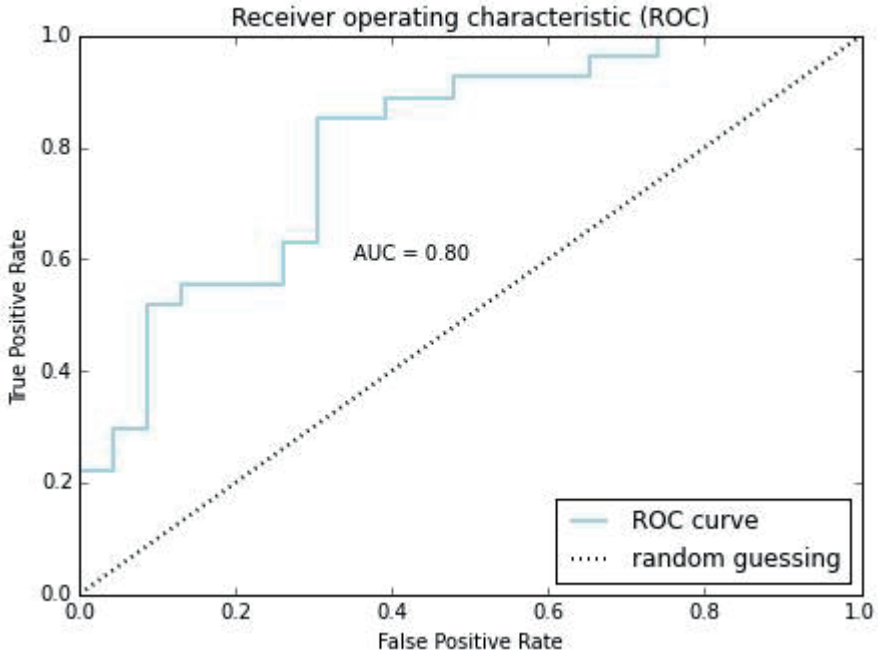


Figure 4.7: ROC curve

Most classification algorithms seek models that attain the highest accuracy, or equivalently, the lowest error rate when applied to the testing dataset.

4.1.3 Performance Evaluation of a Classifier

Some methods commonly used to evaluate the performance of a classifier are: holdout, random sub sampling, cross validation and bootstrap. They are related to each other; The following sections present each of them [Tan/Steinbach/Kumar 2005].

4.1.3.1 Holdout Method

The original data with labelled examples is partitioned into two disjoint sets, called the training and the testing set, respectively. A classification model is then induced from the training set; its performance is evaluated on the testing set. The proportion of data reserved for training and for testing is typically the discretion of the analysts e.g., $50-50$ or two-thirds for training and one-third for testing. The accuracy of the classifier can be estimated based on the accuracy of the induced model on the testing set. The model may be highly dependent on the composition of the training and the testing datasets as a limitation. The smaller the training set size, the larger is the variance of the model. On the other hand, if the training dataset is too large, then the estimated accuracy computed from the smaller testing

dataset is less reliable. Such an estimate is said to have a wide confidence interval. Finally, the training and testing sets are no longer independent of each other. Because the training and testing sets are subsets of the original data, a class that is over-represented in one subset will be under-represented in the other, vice versa.

4.1.3.2 Random Sub-Sampling

The holdout method can be repeated several times to improve the estimation of a classifier's performance. This approach is known as random sub-sampling. It has also no control over the number of times each record is used for testing and training. Some records might be consequently more often used for training than others.

4.1.3.3 Cross Validation

An alternative to random sub-sampling is cross validation. Each record is used the same number of times for training and exactly once for testing in this approach. We partition the data into two equal-sized subsets as an example. We first choose one of the subsets for training and the other for testing. We then swap the roles of the subsets so that the previous training set becomes the testing set and vice versa. This approach is called a two-fold cross validation. The total error is obtained by summing up the errors for both runs. Each record is used exactly once for training and once for testing in this example. The $k-fold$ cross validation method generalises this approach by segmenting the data into k equal-sized partitions. One of the partitions is chosen for testing, while the rest of them are used for training during each run. This procedure is repeated k times so that each partition is used for testing exactly once. Again, the total error is found by summing up the errors for all k runs. A special case of the k-fold cross validation method sets $k = N$, the size of the dataset. In this so-called leave-one-out approach, each testing set contains only one record. This approach has the advantage of utilising as many data as possible for training. The testing sets are mutually exclusive; they effectively cover the entire dataset. The drawback of this approach is that it is computationally expensive to repeat the procedure N times. The variance of the estimated performance metric tends to be high, since each test set contains only one record. It is the one selected for our work, because it is the mostly used in literature.

4.1.3.4 Bootstrap

The methods presented so far assume that the training records are sampled without replacement, as considered in the scope of the thesis. There are therefore no duplicate records in the training and in the testing datasets. The training records are sampled with replacement in the bootstrap approach; i.e., a record already chosen for training is put back into the original pool of records so that it is equally to be reused.

4.2 Related Work: Machine Learning and Permissions

We combine requested permissions and classification learning to build a model for malware detection. The major concern is therefore to investigate the Android literature that focuses on requested permissions and then that one using classification learning for the malware detection. The strategy starts by describing works that analyse the requested permissions to make decisions on behalf of the user. Then we show works that use permissions individually and associatively to characterise and classify applications. The last point presents literature related to the mechanism of classification learning to model malware detection. We discuss the limitations and extract elements to improve all these works.

4.2.1 Permission Analysis

This section presents works that analyse the requested permissions to make decisions on behalf of the user. [Holavanalli et al. 2013] propose flow permissions, an extension to the Android permission mechanism. They allow users to examine and to grant explicit information flows within an application as well as implicit information flows across multiple applications. VetDroid [Zhang et al. 2013] is a dynamic analysis platform for reconstructing sensitive behaviour in Android applications from the use of permissions i.e. how applications use permissions to access sensitive system resources, and how these permissions are further utilised by the application. [Felt/Greenwood/Wagner 2010] evaluated whether permissions are effective in protecting users after collecting them out of a large set of Google Chrome extensions and Android applications. Their results indicate that permissions can have a positive impact on system security when the developer declares the permission requirements upfront. However, this study shows that users are frequently confronted with requests for dangerous permissions during application installation. As a consequence, installation security warnings may not be an effective malware prevention tool, even for alerting of users. The previous work was extended [Felt et al. 2012] to provide guidelines for platform designers for determining the most appropriate permission granting mechanism for a given permission. Permission-based security rules were used by Kirin [Enck et al. 2012] to design a lightweight certification framework that can mitigate malware at install time. Rosen et al. [Rosen/Qian/Mao 2013] present an approach to provide users with the necessary knowledge to make informed decisions about the applications that they would like to install, using mappings between API calls and fine-grained privacy-related behaviours. Barrera et al. [Barrera/Kayacik/van Oorschot/Somayaji 2010] perform an empirical analysis on the expressiveness of Android's permission sets and discuss some potential improvements for Android's permission model. Their work is based on signature verification, UID assignment, and how they relate to the granting of permissions. Grace et al. [Grace/Zhou/Wang/Jiang 2012] describe mechanisms by which permissions granted to one application can be leaked to another, either inadvertently or deliberately through collusion. They propose Woodpecker a tool that examines how the Android permission based security model is enforced in pre-installed applications and stock smartphones. PermissionWatcher

[Struse et al. 2012] is an Android application that analyses permissions of other applications installed on the phone. Based on a custom set of rules, it classifies applications as suspicious if any rule applies. PermissionWatcher increases user awareness of potentially harmful applications through a home screen widget. Sarma et al. [Sarma et al. 2012] investigate the feasibility of using the requested permissions by an application, the category, such as games, education, social, of the application and what permissions are requested by other applications in the same category to better inform users whether the risks of installing an application is commensurate with its expected benefit.

Dini et al. [Dini et al. 2012] propose a multi-criteria evaluation of Android applications helping the user to easily understand the trustworthiness degree of an application, both from a security and a functional side. They assign a threat score to each permission according to the criticality of both resources and critical operations, which they have controlled by these permissions. They compute a global threat score for each application, which is a function of the threat score of all required permissions, combined with information regarding the developer, the rating, and the number of downloads of the application.

Di Cerbo et al. [Di Cerbo/Girardello/Michahelles/Voronkova 2011] describe a methodology for the detection of malicious applications in a forensic analysis. Malicious applications in this context are those having access capabilities to sensible data and transmission capabilities as well at the same time; they deceive the users by offering services that typically do not require such capabilities, or do not make a legitimate use of them. The methodology relies on the comparison of the Android security permission of each application with a set of reference models for applications that manage sensitive data.

4.2.2 Characterisation and Classification of Applications

This section presents works aiming to characterise and to classify applications given a dataset of applications. Techniques described here are sub-grouped in approaches based on individual permissions, the combination of permissions, and machine learning techniques.

4.2.2.1 Using Permission Individually

Zhou and Jiang [Zhou/Jiang 2012] characterise Android malware from various aspects, including the requested permissions. They identify individually the permissions that are widely requested in both malicious and benign applications. According to this work, malicious applications clearly request SMS-related permissions more frequently, such as *READ_SMS*, *WRITE_SMS*, *RECEIVE_SMS*, and *SEND_SMS*. The result is the same with the *RECEIVE_ BOOT_COMPLETED* and *CHANGE_WIFI_STATE* permissions. They found that malicious applications request more permissions than benign ones. Barrera et al. [Barrera et al. 2010] found no strong correlation between application categories and requested permissions; they introduce a self-organising method to visualise permission usage in different application categories.

Sanz et al. [Sanz et al. 2012] propose a method for categorising Android applications through machine-learning techniques. Their method extracts different feature sets: (i) the

frequency of occurrence of the printable strings, (ii) the different permissions of the application itself and (iii) the permissions of the application extracted from the Android market. The aim of their work is to classify Android applications into several categories such as entertainment, society, tools, productivity, multimedia and video, communication, puzzle and brain games. Orthacker et al. [Orthacker et al. 2012] develop a method that circumvents the permission system by spreading permissions over two or more applications that communicate with each other via arbitrary communication channels. Sato et al. [Sato/Chiba/Goto 2013] defines a method that analyses Manifest files in Android applications by extracting four types of keyword lists: (1) permission, (2) intent filter (action), (3) intent filter (category), and (4) process name. This approach determines the malignancy score by characterising individual permissions as malicious or benign.

4.2.2.2 Using Combinations of Permissions

DroidRanger [Zhou/Wang/Zhou/Jiang 2012] is a system that characterises and detects a large set of Android malware integrated with two schemes: The first one is permission-based behavioural foot printing to detect the infection based on an essential group of requested permissions by families of malware; also to match applications to malware information in the Manifest, in semantics in the byte code and in the structural layout of the application; the second one is a heuristics-based filtering scheme that defines suspicious behaviour from possible malicious applications and then uses them to detect suspected ones. The footprint engine filters applications.

PermissionWatcher [Struse et al. 2012] helps to classify Android applications being a potential threat to the user based on a set of rules and attack scenarios. They correlate resources to access permission combinations and to derive possible rules and attacks. Rassameeroj and Tanahashi [Rassameeroj/Tanahashi 2011] performed a high level contextual analysis and an exploration of Android applications based on their implementation of permission based security models by applying network virtualisation techniques and clustering algorithms. From that, they discovered new potentials in permission based security models that may provide additional security to users. This method axed on network classification helps to define irregular permission combinations requested by abnormal applications. Gomez and Neamtiu [Gomez/Neamtiu 2011] classify malicious applications into four classes of malware: DroidDream, DroidDreamLight, Zsone SMS, Geinimi. This categorisation is based on resources accessed by these four families, the infiltration technique, and the payload used.

Wei et al. [Wei/Gomez/Neamtiu/Faloutsos 2012] present the nature, sources, and implications of sensitive data on Android devices in enterprise settings. They characterise malicious applications and the risks they pose to enterprises. Finally, they propose several approaches for dealing with security risks for enterprises. Permission additions dominate the evolution of third party applications, of which dangerous permissions tend to account for most of the changes. Tang et al. [Tang/Jin/He/Jiang 2011] introduce an extension of the Android security enforcement with a security distance model to mitigate malware. A

security distance pair is the quantitative representation of the security threat that this pair of permissions may cause. A permission pair's security distance consists of a threat point, which represents the danger level and related characteristics. Canfora et al. [Canfora/Mercaldo/Visaggio 2013] propose a method for detecting malware based on three metrics, which evaluate the occurrences of a specific subset of system calls, a weighted sum of a subset of permissions that the application required, and a set of combinations of permissions.

Enck et al. [Enck/Ongtang/McDaniel 2008] introduce a policy-based system called Kirin to detect malware at install time based on an undesirable combination of permissions. Su and Chang [Su/Chang 2013] determine whether an application is malware depending on the set of requested permissions and announced in the Manifest. They compute a score depending on the number of occurrences of each of the permissions in malware and good software. Compared with previous works, researches [Huang/Tsai/Hsu 2013]; [Sanz et al. 2013], and [Liu/Liu 2014] take into account the occurrence frequency of two permissions as a pair. The occurrence of two permissions as a pair in one application can reflect the potential malicious activities in some aspects. Such as an application with RECEIVE_SMS and WRITE_SMS can hide or tamper with incoming SMS messages [Enck/Ongtang/McDaniel 2009]. Zhu et al. [Zhu et al. 2012] propose a permission-based abnormal application detection framework, which identifies potentially dangerous applications by the reliability of their permission lists and the description of the application.

4.2.2.3 Using Machine Learning Techniques

The problem of anomaly detection can be seen as a problem of binary classification, in which each normal behaviour is classified as normal, whereas abnormal ones are classified as suspicious. Machine learning techniques have been widely applied for classifying applications mainly focused on generic malware detection.

Sanz et al. [Sanz et al. 2013] introduce a method to detect malicious Android applications through machine learning techniques by analysing the extracted permissions from the application itself. Classification features are the presence of tags *uses-permission* (every permission that the application needs to work), *uses-feature* (which features related to the device the application uses), and the number of permissions of each application. They employ supervised machine learning methods to classify Android applications into malware and benign software. Initially, they gather a total number of 4.301 samples of malware using VirusTotal [VirusTotal 2015]; after they remove duplicated samples, 249 ones remained. Concerning benign applications, they select the number of applications within each category according to their proportion in Google Play: native applications (developed with the Android SDK), Web applications (deve loped mostly with *HTML*, *JavaScript* and *CSS*), and widgets (simple applications for the Android Desktop, which are developed similarly to Web applications). They work with 357 samples.

MAMA [Sanz et al. 2013] is a method that extracts several features from the Android Manifest of the applications to build machine learning classifiers and to detect malware. These features are the permissions requested individually and the *uses-feature* tag. They

used K-Nearest neighbours, Decision trees, Bayesian Networks, and Support Vector Machine algorithms for classification.

Huang et al. [Huang/Tsai/Hsu 2013] explore the performance for detecting malicious Android applications using classification learning with four commonly used machine learning algorithms including AdaBoost, Naïve Bayes, Decision Tree (C4.5), and Support Vector Machine. They extract 20 features including requested permissions and those really used. The values of selected features are stored as a feature vector, which is represented as a sequence of comma-separated values. Their experiments show that a single classifier is able to detect about 81% of malicious applications. Combining results from various classifiers can be a quick filter to identify more suspicious applications according to them. Datasets of 124.769 benign and 480 malicious applications are collected and used to conduct the experiments. Aung and Zaw [Aung/Zaw 2013] propose a framework that intends to develop a machine learning-based malware detection system on Android to detect malicious applications and to enhance security and privacy of Smartphone users. This system monitors various permission-based features and events obtained from the Android applications; it analyses these features by using machine learning classifiers to classify whether the application is benign or malicious. The features are some requested permissions, such as *INTERNET*, *CHANGE_CONFIGURATION*, *WRITE_SMS, SEND_SMS, CALL_PHONE* and others not described in the paper. They experiment with a collection of 500 Android applications with a total of 160 features. Shabtai et al. [Shabtai/Fledel/Elovici 2010] classify two types of applications: tools and games. A successful differentiation between games and tools is expected to provide a positive indication of the ability of such methods to learn and to model benign applications; and to detect malware pieces using machine learning techniques on feature vectors. Shabtai et al. extract feature vectors including APK features (APK size, number of ZIP entries, number of files for each file type, common folders), XML features (count of XML elements, attributes, namespaces, and distinct strings, used permissions in the Android Manifest), dex features (a boolean for each method in the Framework – the method is used or not, and a boolean for every type in the framework - type is used or not).

Drebin [Arp et al. 2014] extracts requested permissions and those really used, combined with six other features from the Manifest and the dissembled code. Machine learning techniques are then applied for automatically learning a separation between malicious and benign applications. The SVM is trained offline on a dedicated system; the learned model is transferred to the smartphone for detecting malicious applications. They focus on the Manifest and the disassembled dex code of the application, which both can be obtained by a linear sweep over the application's content. Feature datasets from the Manifest are constituted of hardware components, requested permissions, application components, and filtered intents.

Liu and Liu [Liu/Liu 2014] extract requested permissions, directly extracted from the AndroidManifest.xml file, requested permission pairs, the combination of any two requested permissions is recognized as a request permission pair, and used permissions by analysing the dex file to identify, which permissions are actually used by the application, and used permission pairs, the generative process of this feature is same as requested permission pairs.

The machine learning techniques and permissions are used to classify an application as benign or malicious in this scheme. MADAM [Dini/Martinelli/Saracino/Sgandurra 2012], which stands for multi-level anomaly detector for Android malware concurrently monitors Android at the kernel and at the user level to detect malware infections using machine learning techniques to distinguish between standard behaviour and malicious one. It combines features extracted from several levels to (i) provide a wider range of monitored events and (ii) to discover correlations among these events belonging to distinct levels. MADAM considers two levels, the kernel-level and the application-level; it monitors system calls at the first level. It extracts features such as considering whether the user is idle, and the number of sent SMS in the second level. Crowdroid [Burguera/Zurutuza/Nadjm-Tehrani 2011] is a machine learning-based framework that recognises Trojan-like malware on smartphones by analysing the number of times each system call has been issued by an application during the execution of user interaction. Andromaly [Shabta 2011] is an intrusion detection system that relies on machine learning techniques. This lightweight malware detection system in terms of CPU, memory and battery consumption assists users in detecting and optionally reporting suspicious activities on their handsets to the Android community. The basis of the malware detection process consists of real-time monitoring, collecting, preprocessing and analysing of various system metrics, such as CPU consumption, number of sent packets through Wifi, number of running processes, and battery level. Schmidt et al. [Schmidt et al. 2008] monitor smartphones to extract features that can be used in a machine learning algorithm to detect anomalies. The framework includes a monitoring client, a Remote Anomaly Detection System (RADS), and a visualisation component. RADS is a Web service that receives the monitored features from the monitoring client and exploits this information, stored in a database, to implement a machine learning algorithm. Lee and Ju [Lee/Ju 2013] introduce a method to prevent an installation of malicious applications using permissions called Maximum Severity Rating (MSR) classification. It attempts to find malicious applications by examining requested permissions as well as to enhance the ability to perceive warning signs in the procedure of an application installation. Su and Chang [Su/Chang 2014] detect whether an application is malware according to the announced permission combinations of the application. They use two different weighted methods to adjust the weights of the permissions. These methods are essentially based on permission occurrences in both samples and the frequency gap between samples. They determine risk scores based on previous metrics. A higher risk score indicates that the user risk is high, and the application is more likely to be malware.

Ping et al. [Ping et al. 2014] propose a malware detection method based on contrasting permission patterns. They specify four subsets used for the classification:

(1) Unique permission patterns in a malware dataset: frequently required patterns in the malware dataset that never occurs in the clean dataset; namely, the support degree of their item sets in the clean dataset is 0. They are used for the malware profile.

(2) Unique permission patterns in a clean dataset: frequently required patterns in the clean dataset that never occur in the malware dataset. They are used for the normal profile.

(3) Commonly required permission patterns: patterns that are supported by both data-

sets. Basically, most patterns are commonly required with significantly different support degrees in two datasets.

(4) Contrasting permission patterns: patterns used to build a hybrid profile that combines the normal and the malicious profile, as well as the common one.

Given an unknown application, they firstly encode it as a 130-bit vector X according to its required permissions. They traverse the patterns then for each sub-profile to find those contained in X, and evaluate how likely X is clean or malicious. Finally, they aggregate the estimations of sub-profiles and make the final detection. They build their own classifier Enclamald for malware detection composed of all contrasting permission patterns discovered from the training dataset and a set of parameters used for malware detection.

Moonsamy et al. [Moonsamy/Rong/Liu 2014] propose a contrast permission pattern-mining algorithm to identify the interesting permission sets that can be used to distinguish applications from malicious to clean. They focused on required and used permissions for that. They equally used the dataset published by Zhou and Jiang [Zhou/Jiang 2012], which includes 1.260 malicious applications from 49 malware families, and 1.227 clean ones collected from two popular third party Android application markets, SlideME and Pandaapp. All clean applications have passed the virus tests by 43 antivirus engines on VirusTotal.

Protsenko and Müller [Protsenko/Müller 2014] use random metrics related to software code combined with a feature specific application structure, to detect malware with machine learning algorithms. Rovelli and Vigfusson [Rovelli/Vigfusson 2014] design PMDS (Permi ssion-based Malware Detection System): A cloud-based architecture based on requested permissions as the main feature for detection abnormal behaviour. They build a machine learning classifier on the features to automatically identify unseen applications with potentially harmful behaviour based on the combination of required permissions.

Wang et al. [Wang et al. 2014] employ three methods: Mutual Information, Correlation Coefficient (CorrCoef) and T-test to rank individual permission with their risk to the system. Then they use Sequential Forward Selection (SFS) as well as Principal Component Analysis (PCA) to identify risky permission subsets. Secondly, they evaluate the usefulness of risky permissions for malware detection with the SVM, decision trees as well as random forest algorithms. Third, they evaluate their malware detectors built on risky permissions.

AndroTracker [Kang/Jang/Mohaisen/Kim 2014] is a method to improve the performance of malware detection by incorporating the developer's information as a feature and to classify malicious applications into similar groups. Features for classification include system commands often used by malware, requested permissions, API, intents and serial number of certificate. The detection algorithm starts by checking the application's serial number for fast scanning by applying one established blacklist. Secondly, the algorithm checks the usage of the system commands, which are previously defined. These commands, which run on a rooted device or to root a device, are found in malicious code. The next step is finding malware that conceals SMS notification. The step checks whether an application uses the above methods and an intent filter to catch such malware. The algorithm adopts a permission based detection rule for the final step. It calculates the likelihood ratio under a given distribution of permissions. Two likelihood ratios are obtained using critical per-

missions and API-related ones. It checks whether an application sends a SMS or collects sensitive information like a device ID, a phone number, a serial number of the SIM card, and the location of the device.

4.2.3 Discussion

Permissions are one of the keys for security on Android. According to the results obtained from previous works, a permission-based mechanism can be used as a quick filter to identify malicious applications; that must be associated with a second element, such as dynamic analysis, to make a complete analysis for a malicious application. This result is justified with interesting performance indicators such as true positive, false positive, true negative and false negative. For the work on permission analysis, the solutions help the user to make the choice of permissions, extending either the installation system without the participation of the user or by presenting an interface with the necessary information on permissions with the participation of the user.

4.2.3.1 Limitations

In most of the previous works, such as [Sato/Chiba/Goto 2014], the authors restrict the study to the most requested permissions or a precise set of permissions determined by Zhou and Jiang [Zhou/Jiang 2012]. However, other permissions such as *READ_LOGS* can be as malicious as the others, *INTERNET* for example, depending on the attack. Every permission should be carefully considered as potentially risky once combined with another one. Moreover, many permissions are defined coarse-grained: this allows the possibility that a permission hides the semantic of other ones.

Researchers on Android require machine-learning techniques for the classification between benign and malicious applications such as decision tree classifiers. While learning techniques provide a powerful tool for automatically inferring models, they require a representative database for training. The quality of the detection model of such systems critically depends on the availability of representative malicious and benign applications [Arp et al. 2014]. While the collection of benign applications is straightforward, gathering recent malware samples requires some technical effort. Another limitation of machine learning is the possibility of mimicry and poisoning attacks [Perdisci et al. 2006], [Venkataraman/Blum/Song 2008]. Obfuscation strategies, such as repackaging, code reordering or junk code insertion, renaming of activities, and components between the learning and the detection phase may not affect the model [Rastogi/Chen/Enck 2013], [Zheng/Lee/Lui 2012]. An attacker may succeed in lowering the detection score by incorporating benign features or fake invariants into malicious applications [Perdisci et al. 2006].

The selection of the right classifier is somehow difficult depending on the classes of applications to be trained with. The rate of false negatives errors arises in the process of filtering as a consequence. If too many features are used in machine learning, the detection rate may decrease. Authors simply select a set of known malware randomly without considering to train a malware detector as it is done in most of the previous works. This way of

selection yields significantly biased results according to [Allix/Bissyande/Klein/Le Traon 2014]. Machine-learning based detection approaches have two limitations: they have high false alarm rates; the determination of features to learn in the training phase is complex. A key step in these approaches thus resides in selecting datasets for training. The performance degrades over time: for a given month i whose applications have been used for the training datasets, the obtained classifier is less and less able to identify malware samples in the following months $i+k$, $k \geq 1$.

The number of features that must be extracted from the Manifest, such as in [Canfora/Mercaldo/Visaggio 2013] and [Huang/Tsai/Hsu 2013], increases the computing overhead and the inefficiency of the solution. The choice of features to associate with is relevant, because its modification can give false results. For instance, Zhu et al. [Zhu et al. 2012] give acceptable results if the developer has really filled the description. The output could otherwise be false. This is also the case for the technique proposed by [Gomez/Neamtiu 2011], which is inadequate for detecting unknown malware because applications are classified using characteristics of known families of malware. [Shabtai/Fledel/Elovici 2010], [Wei/Gomez/Neamtiu/Faloutsos 2012] classify applications especially of known families of malware, categories of applications and enterprise ecosystems following the same idea.

Experimental results are evaluated with TPR, FPR, TNR and FNR metrics to compute the accuracy of the model. Authors often build a ROC curve to graphically evaluate the performance of the model. They unfortunately compare them with similar works although the datasets are different; We think that it is scientifically not precise. Probability mechanisms with parameters such as dataset sizes of datasets involved in different experiments should be taken into account in this case; otherwise it is not possible to affirm that a model A with parameters P_A has a higher performance than a model B with parameters P_B. One can only compare models implemented under the same experimental conditions.

Most of these works extract a feature set to represent the applications. The information carried by these features is different from work to work. They do not show, which features give the best detection result, but permissions are considered in each study. For instance, Moonsamy et al. [Moonsamy/Jia/Shaowu 2014] and [Rovelli/Vigfusson 2014] take the permissions as the only feature to represent the applications and to find specific permission patterns to show the difference between clean and malicious applications with good results.

The problem of usability of solutions remains urgent for the security of Android [Tcha kounté/Dayang 2013]. Many security solutions (Flowdroid [Fritz et al. 2013], Comdroid [Chin/ Felt/Greenwood/Wagner 2011]) are hard to install even for expert users. The deployment is often not applicable in real devices, requiring installing components by command line. The objective of a security system based on permissions should help users to evacuate malicious applications by clearly understandable interfaces.

Most of the approaches using machine-learning classifiers are just theoretical: there is no application built to validate the results found. This shows eventually the difficulty of practicability of such mechanisms. Some works build the classifier inside a remote server, which receives some information necessary for the classification from the smartphone [Rovelli/Vigfusson 2014]. After completing the task, the server replies with the classification

results to the smartphone. This scheme requires that the user possesses a high Internet bandwidth and a secure communication channel. Apart from the high cost to deploy such a solution, the user and the server can receive information modified by a Man in the Middle.

Different users have different types of privacy and security concerns [Zhou/Jiang 2012]. One can need to protect his SMS while another one needs to protect his contacts. Research on permissions try to identify implicitly concerns related to the user while categorising permissions either in privacy threat, system threat, money threat [Dini et al. 2012], or in privacy, monetary, damage [Sarma et al. 2012]. These views are too coarse and not resource-oriented; moreover the user is not involved in the definition of the important resources of the smartphone.

4.2.3.2 Enhancements

Some efforts should be made to improve the effectiveness of permission-based solutions.

For reason of completeness, unlike previous works, one should consider not only the 130 official permissions in Android, but also additional ones published in the GitHub [Android source 2015] and third party permissions not listed in the previous sources. The reason is to consider every permission as risky when it is combined with others. The research should study all these permissions rather than to focus on the most requested one.

For reasons of flexibility and performance, a security system should learn from application profiles rather than using machine-learning techniques. When machine-learning techniques are used, "applications, including malware, used for training in machine learning-based malware detection must be historically close to the target dataset that is tested. Older training datasets indeed cannot account for all malware lineages, and newer datasets do not contain enough representatives of most malware from the past", as stated by Allix et al. ([Allix/Bissyande/Klein/Le Traon 2014], p.11). Building a reliable training dataset is essential to obtain the best real-world performance. There are methods for offline analysis, such as DroidRanger [Zhou/Wang/Zhou/Jiang 2012], AppsPlayground [Rastogi/Chen/Enck 2013] and RiskRanker [Grace et al. 2012] that might help to automatically acquire malware and to provide the basis for updating and maintaining a representative dataset for such techniques over the time.

Following the previous idea, it can be more efficient to build a classifier by using general principles for classification learning, rather than to use the black boxed ones provided by the machine-learning field, such as AdaBoost, Naïve Bayes, Decision Tree (C4.5), and Support Vector Machine as it is done mostly. The simple reason is that only the authors who built these algorithms know how to adapt their contents with parameters. Ping et al. [Ping et al. 2014] built their own classifier Enclamald for malware detection with this logic and found it better compared to other commonly used classifiers (Logistic, LibSVM, Random Tree, RBFNetwork, SMO, BFTree, AdaBoostM1, Bagging, and J48) after performing ten-fold cross validation.

One should avoid using several features to construct the vector representing an application. This can increase significant overhead. Some approaches such as [Canfora/Mercal-

do/Visaggio 2013] for the categorisation of Android applications start by identifying (often intuitively) the set of permissions as features for detection. To determine different clusters of permissions to categorise applications in normal and malicious applications after careful considerations is more straightforward. The refinement is effective with the use of weights on permissions based on its frequency, as suggested by [Rassameeroj/Tanahashi 2011].

None of the previous works determining occurrences of permissions examine duplicated permissions in the Manifest. The extraction of permissions from applications should consider this possibility for the reason of precision. The percentage of permission occurrence in malware and benign software is one of the features often used by works aiming to characterise malware. If a permission is required ten times more in normal applications than in malware, this permission must not be discriminated, in this context. The best approach should be to find a correlation considering every proportion of the use of permissions in malware and in good software. Even when a permission is present just once in malware, it must be significantly noted.

It is further recommended to implement a usable application related to the experiments that the user can install. This application should have all its components installed inline and it should be lightweight in terms of execution time. A survey on the usage can be performed to evaluate the usability in order to improve the design accordingly. For the detection of implicit vulnerabilities, one should associate a dynamic module that learns the behaviour of an application that cannot be detected based on the permission profile [Enck 2011]. A dynamic analysis of system calls and its parameters can be exploited in this case.

None of the previous approaches on permission analysis involves the user with the decision, which resources shall be protected. This represents the user's concern on security of the smartphone and can be used to evaluate the risks for resources.

The task of predicting the normal and abnormal class labels of testing records should be considered as a binomial experiment. To compare the performance of two classification models requires estimating the confidence interval of a given model accuracy and testing the statistical significance of the observed deviation related to each experiment ([Pang-Ning/Steinbach/Kumar 2005], p. 188). These elements are sufficient for performing the evaluation between many models.

Chapter 5

A Three-Layered Malware Detection and Alerting System for Android (TLMDASA)

An Android application requires several permissions to work. To perform certain tasks on the device, such as sending a *SMS* message, each application has to explicitly request a permission from the user during installation. An essential step to install an Android application into a mobile device is to allow all permissions requested by the application consequently. Before an application is installed, the system prompts a list of permissions requested by the application and asks the user to confirm it. The open design of the Android operating system still allows a user to install any application downloaded from an untrusted source, although Google has announced that a security check mechanism is applied to each application uploaded to their market [Lockheimer 2012]. The permission list, nevertheless, is still the minimal defence for a user to detect whether an application can be harmful. Many users tend to blindly grant permissions to unknown applications and thereby undermine the purpose of the permission system. Malicious applications therefore try to exploit the Android permission system in practice.

Several works [Zhou/Jiang 2012], [Ping et al. 2014], [Liu/Liu 2014], [Arp et al. 2014], [Sarma et al. 2012], and [Enck/Ongtang/McDaniel 2009] have so far attempted to characterise and to detect malware relying on permissions. The publication of [Enck/Ongtang/McDaniel 2009] was one of the first works to detect malware using permissions and their association. They proposed Kirin, a security service, which relies on nine rules to identify malware. However, most of these rules cannot be applied to current permissions. [Zhou/Jiang 2012] systematically characterised 1260 Android malicious applications from various aspects, including their installation methods, activation mechanisms, and the carried malicious. This work is just limited to study the top 20 frequently requested permissions for both benign and malicious applications. [Ping et al. 2014] designed a method using contrasting permissions that rely on their frequency for malware detection. This method is limited by high computing cost and by inefficiency, because the number of patterns is high, reaching often 25595. Drebin [Arp et al. 2014] uses other features additionally for identifying ma-

licious applications with a Support Vector Machine. Liu and Liu [Liu/Liu 2014] consider fourty permissions in different vector features: requested permissions, requested permission pairs, used permissions, and used permission pairs. They apply classifiers to vectors for the detection. This approach is not efficient due to the size of a vector: It reaches for instance 870 for the vector requested permission pairs. [Sarma et al. 2012] use 26 critical permissions to generate the risk signal for an application. A critical permission is rare if it occurs in less than 2% of the normal training dataset. They consider therefore frequency of permissions only in a normal dataset, although malicious and normal applications can have the same frequency for permission. All previous works do not involve the user in defining his sensitive data.

Our work designs a Three-Layered Malware Detection and Alerting System for Android (TLMDASA), a new system, which considers 222 permissions composed of those published in the manifest.xml in Android source [Android Source 2015]. The system is based on three different layers including the frequency of permissions, the risks associated to combination of permissions related to sensitive resources and the association of both. We involve moreover the user to specify resources considered as sensitive. Risk signals are generated to alert, depending on this information and the features in the second layer.

5.1 The Architecture of TLMDASA

TLMDASA is a lightweight and flexible system only focusing on requested permissions in applications to be analysed by the user. We propose a three-layered malware detection system using machine learning technology based on extracted features. It aims to detect potential malicious applications fast and relatively accurate. The detection process of our system is divided into three layers. The first layer depends on two schemes: the occurrences of permissions in malicious and normal applications, and the proportion of solicitation of permissions in malware within the whole dataset. The second layer focuses on risks when granting access to sensitive resources. The third layer combines security requirements and a learning algorithm applied to joint vectors for applications. TLMDASA design relies on five operations.

- Static analysis: TLMDASA statically scrutinises a given Android application and extracts requested permissions from the application's Manifest.
- Translation into the vector space: Feature vectors are determined for each layer after dissembling the application;
- Learning-based classification: Classification learning techniques are applied to application vectors determining detection rules, which enable the identification of malware.
- Alerting: The user specifies a category of sensitive resources. The present module retrieves permissions belonging to this category along with risks, retrieves requested permissions from the Manifest and compares the computed risks with those pre-

defined by the second layer. Then, it forms the alert message to the next module accordingly.

- Displaying: It assures the ease of understanding the results displayed to the user. Results come from the classifier and the alerting modules.

Figure 5.1 shows the architecture.

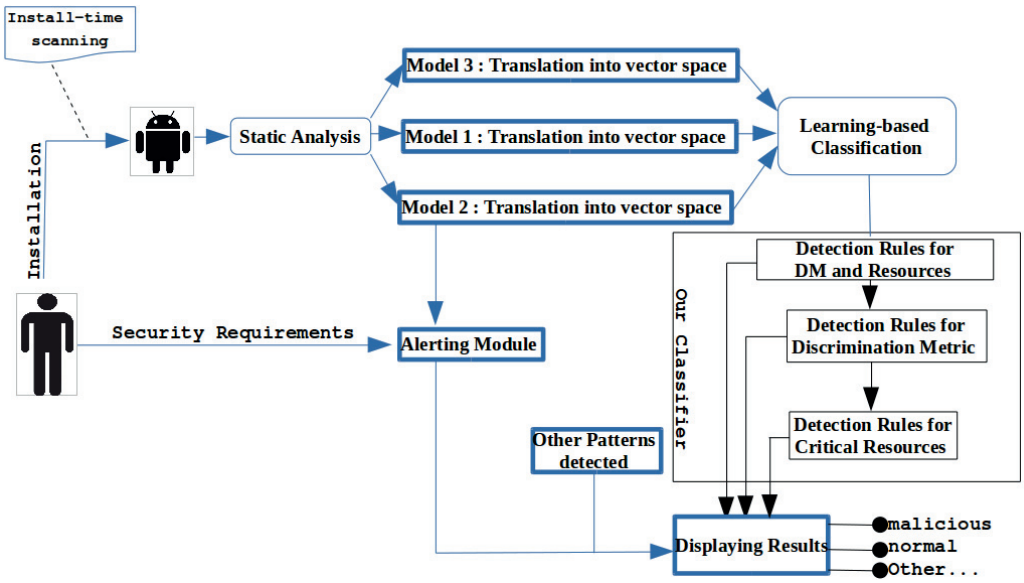


Figure 5.1: The TLMDASA architecture

TLMDASA is built on four hypotheses:

H1: The revaluation of the risk level of permissions can improve the characterisation based on permissions: Only permissions "dangerous" are displayed to the user during the installation of an application. According to Google, the more an application asks for permissions dangerous, the more the application tends to be malware. It is however, practically false. Vennon and Stroop analysed 68% of the Google applications for requested permissions and found that more than half are mistakenly considered as malicious if the criterium dangerous defined by Google is used [Vennon/Stroop 2010]. An occurrences metric can help to identify permissions needed by malware and those needed by normal applications.

H2: There exist combinations of permissions proper to malicious applications and to normal ones, which can be exploited for defining efficient classification rules: We are guided by observations done in literature and in official documentations on permissions. For instance, even if *INTERNET* is the most requested permission by malicious applications, it needs other associated permissions to damage the user. The association *INTERNET* and *READ_ PHONE_STATE* can allow the application to retrieve device information and to send them to a remote server. This association seems to be more dangerous than taking a permission individually. Several works publish results in the same direction [Aung/Zaw 2013]; [Felt et al. 2011]; [Zhu et al. 2012]. On the

contrary to these studies, hypothesis H1 is used as starting point to determine rules to differentiate malicious from normal applications.

H3: The device is non-rooted and free of vulnerabilities: This is to assure that system files are not tampered with.

H4: Malicious applications without any permission can not be detected: Since the thesis aims at detecting malicious relying on permissions, an application without any permission can not be detected.

5.2 Definitions

We are interested in performing each phase of the system, which requires introducing a number of definitions, which will be used later.

Definition 1: We denote by
$A^L = \left\{a_1^L, a_2^L, \ldots\ldots, a_{|A|}^L\right\}$ the learning dataset of malicious applications,
$B^L = \left\{b_1^L, b_2^L, \ldots\ldots, b_{|B|}^L\right\}$ the learning dataset of normal applications,
with $\mid A^L \mid$ and $\mid B^L \mid$ the sizes of A^L and B^L.
$A^T = \left\{a_1^T, a_2^T, \ldots\ldots, a_{|A|}^T\right\}$ the test dataset of malicious applications and,
$B^T = \left\{b_1^T, b_2^T, \ldots\ldots, b_{|B|}^T\right\}$ the test dataset of normal applications,
with with $\mid A^T \mid$ and $\mid B^T \mid$ the sizes of A^T and B^T.

Definition 2: We denote by $Perm = \left\{P_1, P_2, \ldots\ldots\ldots, P_{|Perm|}\right\}$, the set of permissions used in the system with$\mid Perm \mid$ the size of $Perm$, which is constituted by permissions declared in Android GitHub [Android Source 2015]. 206 permissions with complete descriptions are provided. Android GitHub contains 82 permissions more than those published for API 19 [Android 2015] because Kitkat is the most bought Android version in the world with a market share of about 39.2% [SocialCompare 2015]. We consider 16 permissions not listed in the previous sources, but only found in third party applications during an experimental study. Therefore, $\mid Perm \mid = 222$.

We denote $P(a)$ as the set of all different permissions found in application a. $P(a)$ does not contain repeated elements.

Definition 3: The presence, $presence(p,a)$, of permission p in application a is given by:

$$\forall p \in Perm,\ presence(p,a) = \begin{cases} 1 & if\ p \in P(a) \\ 0 & if\ not \end{cases}$$

Definition 4: The occurrences $occurrence(p,E)$ of permission p in the set of applications E are defined by:

$$occurrence(p,E) = \sum_{\forall p \in Perm, \forall a \in E} presence(p,a)$$

Definition 5: The gap_i between the occurrences of permission i in A^L and B^L is given by:

$$\forall i \in \{1, \cdots, |\, Perm \,|\}\,,$$

$$gap_i = \begin{cases} 0 & if\ occurrence(i,A^L) \leq occurrence(i,B^L) \\ occurrence(i,A^L) - occurrence(i,B^L) & if\ not \end{cases}$$

Definition 6: The proportion $proportion(i)$ of requests of permission i by malicious applications is defined by:

$$proportion(i) = \frac{occurrence(i,A^L)}{occurrence(i,A^L)+occurrence(i,B^L)}$$

5.3 Static Analysis

Some elements motivated the choice of static analysis, which is the first step of TLMDASA. The execution time is relatively lower than in dynamic analysis and we do not need to create complex environments of execution specific to each application like in dynamic analysis. The same analysis environment can be used for all applications using scripts to automatise operations. The Manifest is the file scrutinised to extract the only feature: requested permissions. This file is an obligatory component to declare permissions necessary for the application. Because it is easily accessible and includes information on the Android package, it is particularly indicated for static analysis. Additionally, this file gets rarely modified during the existence of the application. Permissions declared during the installation time cannot change (deletion or addition), not even during an update, since version 4.2 of Android.

We represent all extracted features as sets of strings such as permissions to allow a generic and extensible analysis. The permission *android.permission.SEND_SMS* is, for instance, abbreviated to *SEND_SMS*. Applications are required to dissemble in order to gather the requested permissions from the Manifest in a feature set.

5.3.1 Malware for the Learning Phase

Our malware sample includes the malware dataset released by the Drebin authors [Arp et al. 2013]. It is composed of 5560 malicious applications collected from 2010 to 2012. Arp et al. include also malware grouped into 49 families, which have been collected between August 2010 and October 2011 and released by [Zhou/Xiang 2011]. We additionally gather 1223 malicious applications from Contagio [Contagio 2015] and VirusTotal [VirusTotal 2015] collected from 2012 to 2014.

5.3.2 Normal Applications for the Learning Phase

A dataset of 1993 normal applications has been collected from 2012 to 2015 in Google Play [Google Play 2015] and VirusTotal [VirusTotal 2015]. We selected free applications in Google Play based on the descriptions, the number of downloads and the ratings given by

users: only the top ones are picked. There is no mean to prove the goodness of applications; even the official market can contain malware. Applications from Google Play offer, however, a high legitimacy compared to those from alternative sources [Zhou/Jiang 2012]. Each application taken from Google Play has been scanned by fifty-seven engines from renowned anti-viruses on VirusTotal [VirusTotal 2015] and only the ones that succeed all virus tests are considered as benign and kept inside the dataset of normal applications.

5.3.3 Applications for System Validation

Some applications constitute the dataset for evaluating and validating our security system. Normal applications have been collected in Google Play between 2013 and 2014, and the malicious ones from Contagio during the same period. According to [Allix/Bissyandé/Klein/Le Traon 2014] learning and testing datasets must be historically coherent for good performance of the malware detection scheme; this justifies the period for the collection of datasets.

5.3.4 Readjustment of the Normal Sample

We adopted a probabilistic approach to estimate possible occurrences of permissions in a sample with 6783 normal applications, since the size of the sample of the malicious ones is around five times the size of the normal applications. This solution is motivated by two reasons: the 1993 normal applications are of different categories; they are the most often downloaded ones and the most recommended by Google [Vennon/Stroop 2010]. These selection criteria guarantee that the way permissions are requested in the same proportion follow the same tendency of permission requests by other normal applications in Google Play [Castillo et al. 2011].

$$\forall i \in \{1, \cdots, |\, Perm \,|\}\,,\; p_i = \frac{occurrence(i, B^L)}{|B^L|}$$

where p_i represents the probability of the request of the permission i. Probable occurrences of permissions in a sample of 1260 normal applications will be estimated as follows:

$$\forall i \in \{1, \cdots, |\, Perm \,|\}\,,\; N_i = \lfloor |\, A^L \,| \times p_i \rfloor$$

N_i is the number of occurrences predicted for permission p_i in this equation. Table 5.1 illustrates the application on some permissions.

Permissions	Actual occurrences in the sample of 1993	Probabilities	Predicted occurrences in the sample of 6783
SEND_SMS	95	0.047	323
ADD_VOICEMAIL	0	0	0
RECEIVE_SMS	112	0.05	381

Table 5.1: Determination of predicted requests of permissions

Results obtained with the probability approach have been compared with those obtained in studies of [Sarma et al. 2012] and [Zhou/Jiang 2012]. We found similar results validating the readjustment method.

5.3.5 Tools for the Extraction

The Santoku platform [Santoku 2015] has been used for the extraction of applications. It is a free environment of integrated development based on the distribution Lubuntu. Santoku offers the usual functionalities of Linux systems; it has been particularly designed to offer tools for static and dynamic analysis of applications on mobile and desktop systems (Figure 5.2).

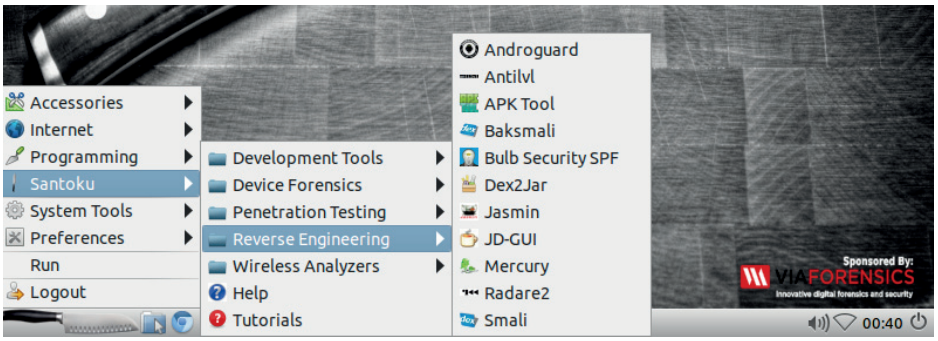


Figure 5.2: Overview of the Santoku environment

Only tools for re-engineering are used because the objective is to investigate files included in the package of an application independently of its execution, concerning our research. The used tools are consigned in Table 5.2.

Tools	Description
Unzip	To unzip the APK packages of the samples.
AXMLPrinter2	To readably render the binary Manifest file included in the package. The output of the operation can be saved into a text file.
Dex2Jar	To manipulate the executable .dex and the Java class files. It is used to transform an .apk package into a Java package .jar.
Apktool	To decode binary files included in applications. In so doing, the resources of an application are decoded almost identically to the original ones; after modifications they can be rebuilt to form the file. It is used to obtain a usable Manifest file.
JD-GUI	To visualise the content of the Java classes of an application from .jar.

Table 5.2: Tools for static analysis

Additionally, some scripts to automatise the extraction presented in the Appendix D have been developed and executed into the environment of the analysis to automatise the tasks of information extraction. These scripts allow constituting the set of permissions to be scrutinised. Samples are ready to be dissected at the end of this phase.

5.4 Layer 1: Relying on Discriminating Metrics

TLMDASA requires three different layers, each based on a specific feature, to classify an application. This section describes the model of the first layer, which takes the requested permissions and calculates their Discriminating Metrics (DM). The DM model is a novel method to weight a permission; its definition includes two objectives: The first one concerns a measure, which indicates the capacity for the permission to characterise malicious applications compared to normal ones. The second one is to evaluate the danger level, which may appear once the user grants this permission during installation of the application. The higher DM is, the more the permission is considered to be preferred by malicious applications; so it represents a high risk for devices. A question arises consequently: "From which value α, is the gap_i considered to be significant?" Two elements simultaneously guide the answer to this question:$|A^L|$ and the DM's scale. Intuitively, we expect a scale of ten measures for permissions looking for more fine-grained evaluation compared to Google Permission Levels (GPL), which define four scales to effectively used to discriminate applications. During the experimental test, we finally defineα as follows

$$\alpha = \lfloor \max\left(\frac{gap_i}{8}\right) \rfloor$$

As the scale goes until nine, we reserved the scale nine for permissions, which indicates specifically malware. This is the reason why we stop at eight.

We model DM by combining two strategies: The first one considers the occurrences of permissions in goodware and malware and the second one the proportion of requests for permission in malware.

5.4.1 First Strategy

Common methods widely used to analyse Android permissions are statistical ones, such as frequency counting [Sarma et al. 2012]; [Ping et al. 2014] and the probabilistic model [Peng et al. 2012]; [Frank/Dong/Felt/Song 2012]. Thus, we start from an initial analysis on the normal and malware datasets following the first aspect with the consideration of 222 permissions. The more a permission is requested by malicious applications in this scheme, the more risky it is inside the Manifest. The estimation of DM1 of permission i is given by:

$$\forall i \in \{1, \cdots, |\, Perm\,|\}\,,\; DM1_i = \begin{cases} 0 & if\ gap_i \leq 0 \\ \frac{gap_i}{\alpha} & if\ not. \end{cases}$$

If the gap is lower than zero, then permission i is more present in a normal application sample. As this thesis focuses on determining malware characteristics, DM1 is equal to zero.

Table 5.3 illustrates the determination of DM of some permissions by applying this method.

Permissions	Requests by malicious	Requests by normal
BROADCAST_SMS	15	4
READ_CALL_LOG	0	37
FACTORY_TEST	6	0

Table 5.3: Examples of determination of DM1

We observed however some pitfalls from the previous evaluation:

- The previous evaluation suffers from the fact that permissions with an important proportion of malware compared to normal applications can not be considered by themodel. The permission *BROADCAST_SMS* is for instance, requested 15 times by malware and four times by normal applications. As long as DM1 is zero this permission is not potentially risky.
- The permission *FACTORY_TEST* clearly characterises malicious actions, whereas it has not been observed with normal applications requesting it. But at the end of the evaluation, the DM1 is zero. This is the contrary to the semantics of DM1 since it means that this permission is considered safe.

These remarks limit the general objective of the model, which aims to consider every permission as risky since it is requested by applications. The new strategy to evaluate DM is considering the proportion of requests by malicious applications within the whole sample composed of normal and malicious applications.

5.4.2 Second Strategy

The estimation of DM2 of permission i is given by:

$$\forall i \in \{1, \cdots, |\, Perm \,|\}\,,\; DM2_i = \begin{cases} (proportion\,(i) - 0.5) \times 10 & if\ proportion\,(i) \geq 0 \\ 0 & if\ not. \end{cases}$$

The relevance is the proportion of malicious applications according to this formula, which requests a permission more than half times of the whole sample (normal and malicious). DM2 is scaled by ten from 0 to 9. The second pitfall of the first strategy still exists.

A set E of permissions has been identified that is requested only by malicious applications, those with no presence in normal applications. They are constituted of third party permissions, SignatureOrSystem, Normal and Dangerous. A fixed value of DM is attributed to these permissions because they allow characterising specifically malware to integrate this particularity.

Table 5.4 presents these permissions with their occurrences.

Permissions	Level	Occurrences
ACCESS_CACHE_FILESYSTEM	SignatureOrSystem	131
ACCESS_DRM	Unknown by Google: declared by third party application	31
ACCESS_GPS	Unknown by Google: declared by third party application	176
ACCESS_WIMAX_STATE	Normal	10
ADD_SYSTEM_SERVICE	Unknown by Google: declared by third party application	31
CHANGE_WIMAX_STATE	Dangerous	8
GLOBAL_SEARCH	SignatureOrSystem	98
GLOBAL_SEARCH_CONTROL	Signature	94
INSTALL_DRM	Unknown by Google: declared by third party application	32

Table 5.4: List of permissions found only in malware samples

The determination of *DM* by associating both strategies is as follows:

$$\forall i \in \{1, \cdots, |\, Perm \,|\}\,,\; DM_i = \begin{cases} 9 & if\ p_i \in E \\ \lceil max(DM1_i, DM2_i) \rceil & if\ not. \end{cases}$$

The discriminating metric of the permission is the *maximum* of the two first calculated. If the permission belongs to E, it is fixed to nine, which indicates a malicious profile. Appendix A summarises the calculation of the discriminating metrics for all 222 permissions.

5.4.3 Translation in Vector Space

We associate a vector of ten elements to all applications. The element i of an application a contains $n(a,i)$, the number of permissions requested by the application a with the discriminating metric equal to i. Table 5.5 illustrates the vector representation.

$n(a,0)$	$n(a,1)$	$n(a,2)$	$n(a,3)$	$n(a,4)$	$n(a,5)$	$n(a,6)$	$n(a,7)$	$n(a,8)$	$n(a,9)$

Table 5.5: Vector representation

Table 5.6 and 5.7 present specifically malicious and normal applications.

3	1	2	2	6	7	3	5	3	2

Table 5.6: Vector representing the malware package 030b481.apk

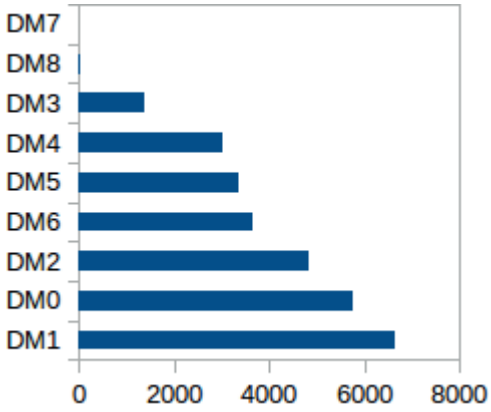
0	0	2	1	3	2	3	4	4	10

Table 5.7: Vector representing the normal package com.whatsapp.apk

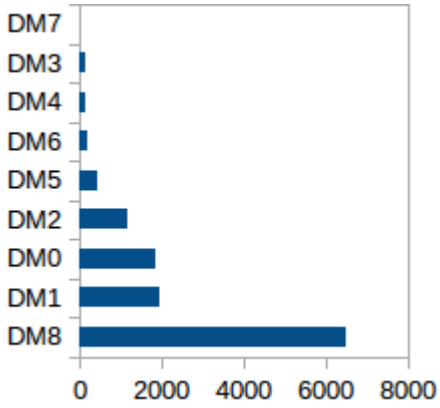
Learning detection uses the sets of vectors for malicious and normal applications.

5.4.4 Distribution of DMs and Discussion

Figure 5.3 derived from Table 5.8, presents discriminating metrics that are mostly used by malware and normal applications.



(a) Distribution of DM for normal applications



(b) Distribution of DM for malicious applications

Figure 5.3: Distribution of DM for the whole sample

	DM0	DM1	DM2	DM3	DM4	DM5	DM6	DM7	DM8
Malware	5748	6636	4856	1382	3011	3366	3675	0	3
Normal	1827	1923	1127	103	104	414	151	0	6465
Total	7575	8559	5983	1485	3115	3780	3826	0	73677

Table 5.8: DM statistics

*DM*9 is not represented because it is proper to malware.

Malicious applications tend to prefer some permissions with DM equal to one whereas normal applications prefer those with DM equal to eight (Table 5.9).

DM1	*ACCESS_FINE_LOCATION*
	BROADCAST_PACKAGE_REMOVED
	BROADCAST_PACKAGE_REMOVED
	BROADCAST_WAP_PUSH
	CALL_PHONE
	CHANGE_CONFIGURATION
	CHANGE_WIFI_STATE
	DEVICE_POWER
	EXPAND_STATUS_BAR
	GET_PACKAGE_SIZE
	HARDWARE_TEST
	INTERNET
	PROCESS_OUTGOING_CALLS
	READ_LOGS
	SEND_DOWNLOAD_COMPLETED_INTENTS
	SET_PREFERRED_APPLICATIONS
	SET_WALLPAPER_HINTS
	STATUS_BAR
	WRITE_EXTERNAL_STORAGE
	WRITE_OWNER_DATA
DM8	*READ_PHONE_STATE*
	SEND_SMS

Table 5.9: Permissions in DM1 and DM8

The set of permissions with DM equals to zero takes the second place in malware, whereas those with DM equals to one take the second place in normal applications. These results mean that permissions listed in DM1 (*INTERNET*, *WRITE_EXTERNAL_STORAGE*, *ACCESS_FINE _LOCATION*, *CALL_PHONE*, and *CHANGE_WIFI_STATE*) belong to the top 20 permissions mostly requested in malware [Zhou/ Jiang 2012]. We deduce that the remaining ones are used in association with other permissions to identify suspicious activities. The permissions *READ _PHONE_STATE* and *SEND_SMS* are essential for the differentiation of malware. They belong to the top 20 permissions mostly requested by normal as stated by [Zhou/Jiang 2012].

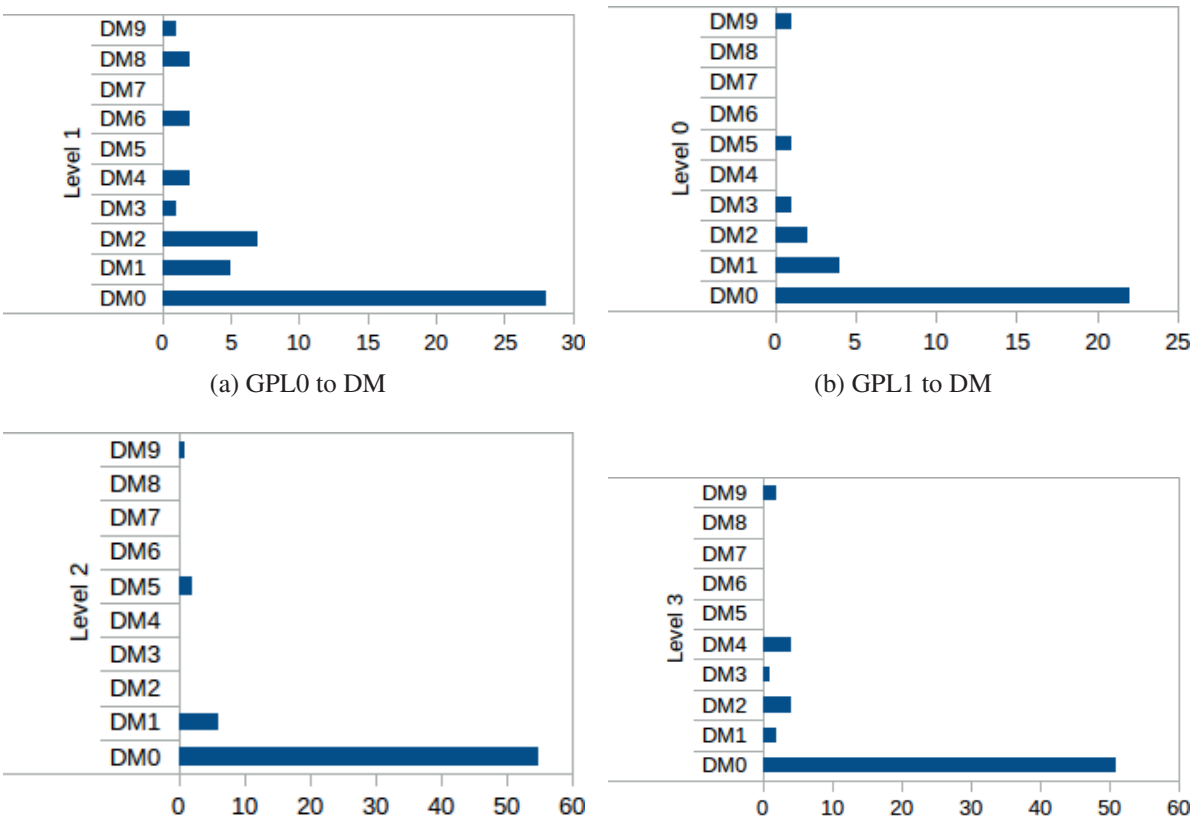
5.4.5 Comparison with the Google Protection Levels

Google defines four levels of permissions as described in Section 2.2.2.

- Level 0: normal with 31 permissions,

- Level 1: dangerous with 51 permissions,
- Level 2: signature with 64 permissions,
- Level 3: signature or system with 64 permissions.

Figure 5.4 illustrates correspondences between GPL and DM. It is particularly observed that GPL0 includes DM0, DM1, DM2, DM3, DM5 and DM9 (Appendix A1). It is further noted that twenty-two permissions of GPL0 remain in DM0, meaning that they cannot be used as elements to distinguish malicious and normal applications. The other nine permissions (*ACCESS_WIMAX_STATE*, *RECEIVE_BOOT_COMPLETED*, *RESTART_PACKAGES*, *ACCESS _LOCATION*, *ACCESS_WIFI_STATE*, *EXPAND_STATUS_BAR*, *GET_PACKAGE_ SIZE*, *PROCESS_OUT GOING_CALLS*, *SET_WALLPAPER_HINTS*) should not be considered as *normal* as defined by Google. Our result is confirmed by those found in the literature [Moonsamy/Rong/Liu 2013], [Sarma et al. 2012] concerning a higher degree of risk provided by the permission *RECEIVE_BOOT_ COMPLETED*. Five remain in DM1, concerning permissions that require user approval (GPL1) defined by Google. The signification can be that our model filters the one proposed by Google and stays with the minimum of being really dangerous: *ACCESS_FINE_LOCATION*, *CALL_PHONE*, *CHANGE_WIFI_STATE*, *INTERNET*, and *WRITE_EXTERNAL_STORAGE*.



(a) GPL0 to DM

(b) GPL1 to DM

(c) GPL2 to DM

(d) GPL3 to DM

Figure 5.4: Correlation GPLs to DMs

Figure 5.4 shows that most permissions in this level cannot be used as a pattern to differentiate between malicious and normal applications. In other words, they can be in GPL0

while considering the Google model. Figures 5.4c and 5.4d include permissions of DM0 in most cases. Although permissions in GPL2 and GPL3 are dedicated to pre-installed applications and those coming with the Android System Image, we see that third party applications request them. These requests are normally ignored by the system. These permissions can take effect, due to a vulnerability of the system. A point of view can be that malicious developers insist in mentioning them in the Manifest to scrutinise system failures.

5.5 Layer 2: Relying on the Risks for Sensitive Resources

This model aims to identify applications from a point of view of accessing resources through requested permissions. A permission denotes the ability to perform a particular operation in Android. The target operation can be anything from accessing a physical resource, such as the device's SD card, or shared data, such as the list of registered contacts, to the ability to start or access a component in a third party application. Android includes the possibility to group permissions according to resources and to certain security objectives. The message group contains, for instance, permissions that allow an application to send messages on behalf of the user or to intercept messages being received by the user. The objective is that these permissions can be used to make users spending money without their direct involvement. This mechanism is used to display information to the user at installation time, which means that if the user grants the *READ_SMS*, an icon representing the group will be displayed that will list its requested permissions. Users can be confused and not aware of the security risks related to granting permissions with this mechanism.

We define ten categories of resources, which could intuitively be targeted by malware presented in Appendix C. They include related permissions as well as their distinct combinations.

Messages (M): Users manipulate *SMS* and *MMS* messaging to communicate with each other. They can be sensitive for users if contents inside should be kept secret or should not being modified. Permissions in this category allow an application to send these resources on behalf of the user (*SEND_SMS*), to intercept them (*RECEIVE_SMS*) and to read or modify them (*READ_SMS*, *WRITE_SMS*). The only permission concerning *MMS* is *RECEIVE_MMS*, which allows monitoring, recording, and processing incoming *MMS* messages. Combinations in this group with *RECEIVE_MMS* are eliminated according to the previous point because we believe that it is worthless whether an application requires permissions related to *SMS* and *MMS* at the same time in order to harm the user. *SMS* can additionally be semantically seen as *MMS*. It would not be meaningful to consider their association, according to our logic.

Contacts (Co): This can be launched without the user's knowledge when someone has the capacity to access (private) user contacts, calls, or even messages. It is therefore fundamental to consider these resources. Permissions in this group are *READ_CONTACTS*, *WRITE_CONTACTS*, and *MANAGE_ACCOUNTS*, which respectively allow an application to read the user's contact data, to write (but not read) them, and to manage the list of accounts

in the *AccountManager*. We associate group accounts and contacts defined by Google separately. All combinations of the three permissions for this resource are considered.

Calls(Ca): Making calls represents one of the services mostly used on smartphones. They are associated with accessing contacts because calling requires having a phone number. Performing actions on calls without user consent can represent a privacy risk for him. Permissions investigated here are *PROCESS_OUTGOING_CALLS* (allowing an application to monitor, modify, or abort outgoing calls), *READ_CALL_LOG* (allowing an application to read the user's call log), *WRITE_CALL_LOG* (allowing an application to write (but not read) the user's contact data), *CALL_PHONE* (allowing an application to initiate a phone call without going through the dialler user interface confirming the call being placed) and *CALL_PRIVILEGED* (allowing an application to call any phone number, including emergency numbers, without going through the dialler user interface confirming the call being placed). Google normally defines the group of permissions *Calls*, which is not limited to call-related permissions but also to permissions associated with accessing and modifying the telephony state. This definition is too coarse-grained and therefore blurs the real sense: Calls can be launched without manipulating the telephony state. We therefore create a distinct group *telephony state*.

Telephony state (T): It includes *MODIFY_PHONE_STATE* and *READ_PHONE-_STATE* permissions, which respectively allow the modification of the phone state (such as power on, reboot) and allow read-only access to the phone state. All combinations are considered in this case such as *MODIFY_PHONE_STATE* and *READ_PHONE_STATE*.

Calendar (Cl): Users save events on a calendar to be reminded later. It can be harmful for the user if one can modify user events without any consent. In this case, meetings can be misleadingly missed or cancelled. The associated permissions are *READ_CALENDAR* and *WRITE_CALENDAR*, which respectively allow an application to read the user's calendar data and allow an application to write (but not read) it. The only association is {*READ_CALENDAR*, *WRITE_CALENDAR*}.

Location (L): This is a resource that is used to know the current location of the device owner. The access of this resource is often granted by default; in this case the user can be tracked physically. *ACCESS_FINE_LOCATION* that allows an application to access the precise location from location sources such as GPS, cell towers, and Wifi, *ACCESS_COARSE _LOCATION* that allows an application to access an approximate location derived from a network location such as *Wifi*, *INSTALL_LOCATION_PROVIDER* (that allows an application to install a location provider into the *Location Manager*), *LOCATION_HARDWARE* (that allows an application to use location features in hardware). This group includes sixteen combinations of permissions integrating the previous ones.

Wifi (Wi): Google defines a group network used for permissions that provide access to networking services. We decide to create a group for Wifi and Bluetooth network resources independently, to detect effectively, which network is frequently used by applications. This resource is mainly used for mobile data communication; if one can take the control of it,

sensitive data can be transferred from or to the device without the user's knowledge. Permissions are: *ACCESS_WIFI_STATE* (that allows applications to access information about Wifi networks), and *CHANGE_WIFI_STATE* (that allows applications to change the Wifi connectivity state). We add moreover the *CHANGE_WIFI_MULTICAST_STATE* permission taken from the group *AFFECTS_BATTERY* defined by Google to complete the present group because it allows changing a property of the Wifi resource. It allows specifically applications to enter the Wifi multicast mode connectivity state; the battery consumption is big in this case.

Bluetooth (B): This is a technology that lets your phone communicate wirelessly over short distances; it is similar to Wifi in many ways. It itself is not a danger to your phone, but it enables an application to send and receive data from other devices. Permissions are *BLUETOOTH* (that allows applications to connect to paired Bluetooth devices), and *BLUE TOOTH_ADMIN* (that allows applications to discover and pair Bluetooth devices). The only combination is {*BLUETOOTH*, *BLUETOOTH_ADMIN*}.

Network (N): This information concerns network socket states, open or closed, and the connectivity state, on or off. It is crucial for accessing a remote server via Internet sending sensitive data from a smartphone. Permissions included are: *CHANGE_NETWORK_STATE* (that allows applications to change the network connectivity state), *ACCESS_NETWORK_STA TE* (that allows applications to access information about network connectivity), and *INTER-NET* (that allows applications to open network sockets).

Webtraces (We): Users usually save sensitive information like password, login, banking codes, etc. consciously when browsing across the Internet. Malicious applications try to gather this resource. Permissions included are *WRITE_HISTORY_BOOKMARKS* (that allows an application to write (but not read) the user's sensitive data) and *READ_HISTORY_ BOOKMARKS* (that allows an application to read (but not write) the user's browsing history and bookmarks).

We define permission risks as follows:

- $Risk_1(R_1)$: It represents the capability given by a permission to an application to directly read confidential information in the device. It is equal to one for the positive case and zero otherwise.
- $Risk_2(R_2)$: It represents the capability given by a permission to an application to directly modify user resources in the device. It is equal to one for the positive case and zero otherwise.
- $Risk_3(R_3)$: It represents the capability given by a permission to an application to perform some actions without knowledge of the user. It is equal to one for the positive case and zero otherwise.
- $Risk_4(R_4)$: It represents the capability given by a permission to an application to charge the user without any consent. It is equal to one for the positive case and zero otherwise.

The weight of a combination C_{ij} of permission j for the resource i is defined by

$\forall j \leq n_c(i)$, $W(C_{ij}) = \sum_{i=1}^{10} R_j(C_{ij})$, $n_c(i)$ represents the number of combinations for the resource i.

To better understand the weight calculation of an application we present the Bluetooth example.

The first line (C17 in Appendix C) shows the permission *BLUETOOTH*; it allows an application to modify information such as the *BLUETOOTH* state according to its definition. The pairing can be done furtively with this permission. Then SMS, MMS or other file types can be sent from compromised devices to other devices. Additionally, compromised phones can infect other phones via Bluetooth with downloaded malicious code or packages. Risks 2 to 4 are present and equal to 1.

The same principle is applied to the second line (C27 in Appendix C). The difference is that *BLUETOOTH* requires the participation of the user to activate Bluetooth through a window unlike *BLUETOOTH_ADMIN* activates it automatically without any consent. Risks 2 to 4 are present and equal to 1.

Risks of the last line (C37 in Appendix C) are straightforward determined from the previous lines as follows:

$$Risk_i(P_1, P_2, \ldots, P_n) = OR\,(Risk_i(P_1), Risk_i(P_2), \ldots.., Risk_i(P_n))$$

OR is the logical *OR* function.

In this case,

$Risk_1$(*BLUETOOTH*, *BLUETOOTH_ADMIN*)=

($Risk_1$(*BLUETOOTH*), $Risk_1$(*BLUETOOTH_ADMIN*))

= *OR*(1,1)= 1

5.5.1 Translation in Vector Space

Thirty-one permissions are considered for the second model, because they are related to resources, which can be deserving protection. Their weights have been determined based on some security risks linked to the user. This section determines how to represent an application into the vector space based on this point of view. The process to construct the vector characteristic for the second model is shown in Algorithm 5.1.

Algorithm 5.1 Construction of the vector for the second model

Input:

- An application a
- Cij: set of combinations i belonging to resource j

Output: The Vector V associated to a

Variables:

- $S = \emptyset$, the set of weight values

Begin
For resource j do
  For Cij of resource j do
    if $presence(Cij, a)$ then
    $S = S \cup W(Cij)$
    else
    $S = S \cup \{0\}$
    end if
  End For
    $V(j) = Maximum(S)$
    $S = \emptyset$
End For
End

Let consider the example of application in Table 5.10.

ACCESS_WIFI_STATE
READ_PHONE_STATE
RECEIVE_BOOT_COMPLETED
WRITE_EXTERNAL_STORAGE
ACCESS_NETWORK_STATE
INTERNET

Table 5.10: Example to illustrate the second model

We obtain the following steps while applying the Algorithm 5.1:

- Resource 1: C_{ij} has no SMS/MMS permissions. $S = 0_1, ..., 0_{16}. V(1) = MAX(S) = 0$
- Resource 2:$S = 0_1, ..., 0_7. V(2) = MAX(S) = 0$
- Resource 3: $S = 0_1, ..., 0_{32}. V(3) = MAX(S) = 0$
- Resource 4: $S = 0_1, ..., 0_4. V(4) = MAX(S) = 0$
- Resource 5 : $S = 0_1, ..., 0_{15}. V(5) = MAX(S) = 0$
- Resource 6 :$C_{ij} = C_{16}, C_{26}, C_{36} \cdots C_{76}. S = 1_1, 0_2, 0_3, 0_4, 0_5, 0_6, 0_7. V(6) = MAX(S) = 1$
- Resource 7: $S = 0_1, ..., 0_3. V(7) = MAX(S) = 0$
- Resource 8: $C_{ij} = C_{18}, C_{28}, \cdots C_{78}. S = 2_1, 1_2, 0_3, 3_4, 0_5, 0_6, 0_7. V(8) = MAX(S) = 3$
- Resource 9: $C_{ij} = C_{19}, C_{29}, C_{39}. S = 0_1, 1_2, 0_3. V(9) = MAX(S) = 1$
- Resource10: $S = 0_1, ..., 0_3. V(10) = MAX(S) = 0$

Table 5.11 is the the resultant vector from this example.

0	0	0	0	0	1	0	3	1	0

Table 5.11: The vector for the application in the previous example

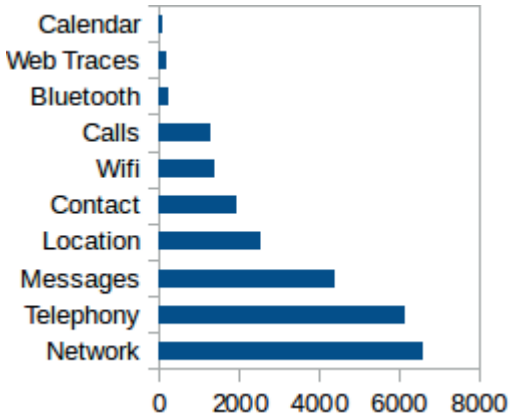
5.5.2 Results and Discussion

We computed the vectors for all applications in our dataset. Table 5.12 presents the most relevant results. They show that the Network category is the most requested by ninety-five percent of the malware dataset. This information reveals that developers mainly manipulate the permissions *INTERNET*, *ACCESS_NETWORK_STATE*, *CHANGE_NETWORK_STATE* and their association. Ninety-seven per cent of malware try to transmit user information remotely and to infiltrate malicious code and to compromise the device. *INTERNET* and *ACCESS_NETWORK_STATE* are the two mostly used by normal applications, as stated in [Zhou/ Jiang 2012]. Our results confirm this situation, as ninety-two per cent of the normal applications need this category.

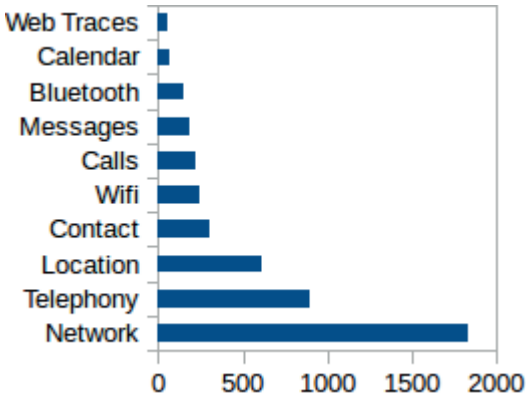
	Blue tooth	Calendar	Calls	Contact	Location	Messages	Network	Telephony	Wifi	Web Traces
Malware	220	89	1300	1924	2537	4376	6582	6124	1383	189
Normal	143	64	224	298	609	182	1839	894	241	52
Total	363	153	1524	2222	3146	4558	8421	7018	1624	214

Table 5.12: Resource statistics

Figure 5.5 represents the results from Table 5.12.



(a) Distribution of resources among normal applications



(b) Distribution of resources among malicious applications

Figure 5.5: Distribution of resources

Almost ninety per cent (88.84%) of malware require permissions for the telephony state category whereas only forty-four per cent of the normal applications do so according to Figure 5.6; sixty-four per cent of malware tend to request SMS/MMS messages while nine per cent of normal applications do so. These results are consistent with the fact that malware is interested to gather information about the user phone such as the unique device ID (IMEI), the current location of the device to have the full surveillance of the activities launched by the user device via network (ninety-five per cent) or Wifi (twenty per cent).

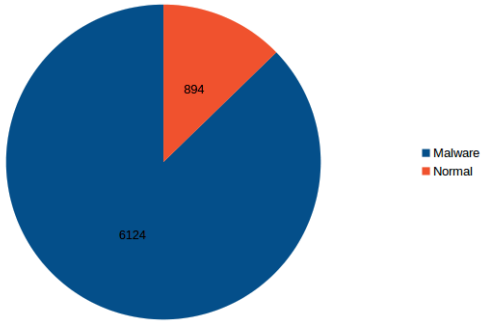


Figure 5.6: Repartition of resources of telephony in samples

The information monitored by malware is:

- Phone calls received and sent (category calls 19%, category contacts 28%)
- SMS received and sent including information about the sender and about the recipients (category messages 64%)
- Approximate location (category location 37%).

We notice moreover that normal and malicious applications have a similar trend in the use of:

- Calendar permissions (1% for malware, 3% for normal)
- Contact permissions (28% for malware, 14% for normal)
- Call permissions (19% for malware, 11% for normal)
- Bluetooth (3% for malware, 7% for normal)

This is due to the fact that these permissions are secondarily employed and related to resources, after the mostly used permissions (network, telephony) are correctly declared. The order representing somehow the importance of each permission category for each class of application is also interesting. For instance, message resources are as significant for malware as location resources are for normal applications. Location resources are likewise as significant for malware as contact resources for normal applications. The same principle exists for contact and Wifi, Wifi and calls, calls and messages.

Thirteen normal applications are oriented to the four security risks, according to Table 5.13.

		Weights							
		4		3		2		1	
		Mal	Nor	Mal	Nor	Mal	Nor	Mal	Nor
Categories of resources	Bluetooth	0	0	220	143	0	0	0	0
	Calendar	0	0	53	29	31	7	5	27
	Calls	0	13	1300	206	0	0	0	5
	Contact	0	0	0	0	748	178	1176	120
	Location	0	0	0	0	2537	609	0	0
	Message	1289	37	2872	75	0	0	215	70
	Network	0	0	4692	1587	1880	235	10	17
	Telephony state	0	0	0	0	161	37	5963	857
	Wifi	0	0	0	0	1372	234	11	7
	Web traces	0	0	0	0	147	16	42	9

Table 5.13: Distribution by resources and weights

This indicates clearly that they seem to be over-privileged by developers who are confused about the declaration of necessary permissions [Felt/Chin/Hanna/Song/Wagner 2011]. Applications are in this case vulnerable; malicious developers thus try to harm the user. A considerable number of applications (1326) are messages-risky requiring SMS permissions. Ninety-seven per cent of this proportion represent malware. The model illustrates that they use permissions devoted to manipulate the user's *SMS* or *MMS* furtively while possibly generating charges. Associations of permissions in this situation are (*SEND_SMS*, *RECEIVE_SMS*, *WRITE_SMS*), (*SEND_SMS*, *READ_SMS*, *WRITE_SMS*), which indicate

an obviously malicious manipulation of the user's SMS in order to send it remotely. Resources network, Bluetooth, calendar and contacts dispose a weight of three, meaning that exactly three security risks are respected. More than the double number of malware, which acts with weight four intends to use network capabilities. It means that they request effectively permission associations such as (*INTERNET*, *ACCESS_NETWORK_STATE*), (*INTERNET*, *CHANGE_NETWORK_STATE*) and (*INTERNET*, *ACCESS_NETWORK_STATE*, *CHANGE_ NETWORK_STATE*) to send a *SMS* or *MMS* to a remote server.

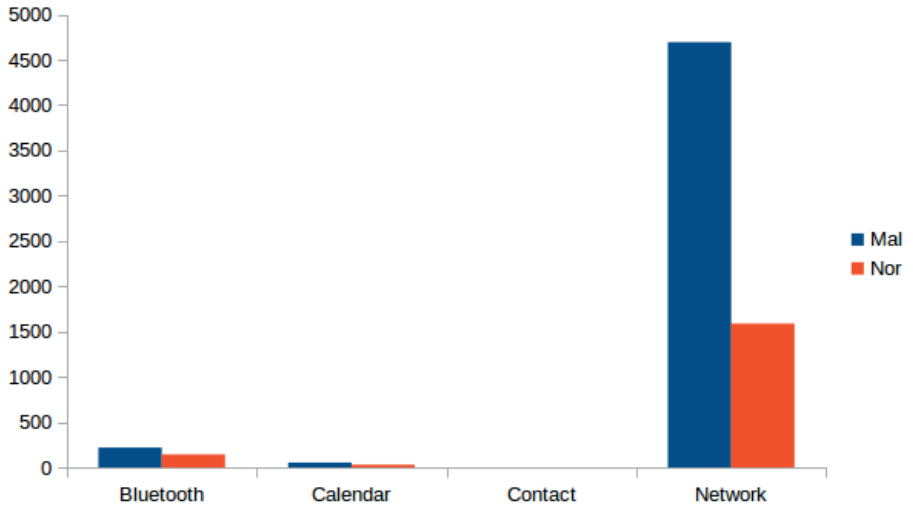


Figure 5.7: Distribution for the weight 2

Normal applications mistakenly use these combinations, which can be used adversely. Figure 5.8 indicates that within the categories of weight three, network resources are the far most targeted by malware followed by Bluetooth.

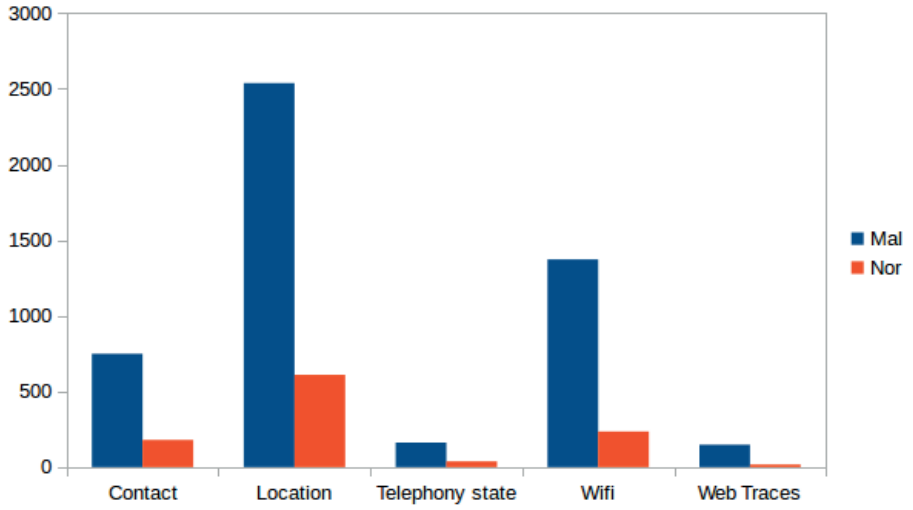


Figure 5.8: Distribution for the weight 3

The trend is similar with normal applications. Bluetooth permissions are requested in a similar way using *BLUETOOTH*, *BLUETOOTH_ADMIN* or both permissions to exchange wirelessly information. Resources concerned for weight two are: Wifi, telephony state, Web

traces, location, and contacts. Malware applications spy the user (in weight one) by getting location, browsing information, phone details, and contacts to transmit via Wifi or Internet.

Unfortunately, there are also normal applications susceptible to do the previous actions, but they are less interested in browsing information and phone details, as shown in the category of weight one. The latter category shows that there is no malicious application, which requests only *READ_CALL_LOGS* to gather calls already made by the user, nevertheless fifteen normal applications do so. A user can confirm the normal behaviour of an application only with dynamic analysis.

We investigate malicious applications, which do not use any permissions selected in the second layer. We select for that malicious applications with zero everywhere in the vector. Results come up with thirty-one candidates as shown in Table 5.14.

Applications	B	Cl	Ca	Co	L	M	N	T	Wi	We
0a02157[1]*	0	0	0	0	0	0	0	0	0	0
0c059ad*	0	0	0	0	0	0	0	0	0	0
0ed1ce6*	0	0	0	0	0	0	0	0	0	0
14c5f2f*	0	0	0	0	0	0	0	0	0	0
1ee4778*	0	0	0	0	0	0	0	0	0	0
277be51*	0	0	0	0	0	0	0	0	0	0
3470ace*	0	0	0	0	0	0	0	0	0	0
36748c5*	0	0	0	0	0	0	0	0	0	0
3aa4080*	0	0	0	0	0	0	0	0	0	0
52c6bac*	0	0	0	0	0	0	0	0	0	0
58f2bcf*	0	0	0	0	0	0	0	0	0	0
6216168*	0	0	0	0	0	0	0	0	0	0
7c0af89*	0	0	0	0	0	0	0	0	0	0
7dcb02d*	0	0	0	0	0	0	0	0	0	0
8407c19*	0	0	0	0	0	0	0	0	0	0
930bcc5*	0	0	0	0	0	0	0	0	0	0
94bb6ad*	0	0	0	0	0	0	0	0	0	0
9cae6a2*	0	0	0	0	0	0	0	0	0	0
a6fa2b6*	0	0	0	0	0	0	0	0	0	0
a8fcc1c*	0	0	0	0	0	0	0	0	0	0
ae7a206*	0	0	0	0	0	0	0	0	0	0
b234dc0*	0	0	0	0	0	0	0	0	0	0
bead195*	0	0	0	0	0	0	0	0	0	0
cdf9aa5*	0	0	0	0	0	0	0	0	0	0
da146c4*	0	0	0	0	0	0	0	0	0	0
dd344a6*	0	0	0	0	0	0	0	0	0	0
e328b00*	0	0	0	0	0	0	0	0	0	0
f440283*	0	0	0	0	0	0	0	0	0	0
f4bd314*	0	0	0	0	0	0	0	0	0	0
fa9d92d*	0	0	0	0	0	0	0	0	0	0
fac847e*	0	0	0	0	0	0	0	0	0	0

Table 5.14: List of applications with no requests of permissions

Twenty-one of them require only one permission at install-time taken in the following list:

- *WAKE_LOCK*: The requiring of this permission has been observed in normal samples.

Therefore, it has been excluded as a specific pattern for malware.

- *READ_LOGS*: The presence of this permission as the only one declared in the Manifest indicates a profile for malware.
- *WRITE_EXTERNAL_STORAGE*: We discovered its presence in both, malicious and normal applications during our experiments. Therefore, we exclude it as pattern for malware.
- *INSTALL_PACKAGES*: This permission is present in malicious applications only. It is a new sign to profile malware.
- *READ_USER_DICTIONARY*: It allows the user accessing calendar information. It appears in the Manifest of malicious applications alone but never in normal applications.

The second model helps to detect three other patterns (*READ_LOGS*, *INSTALL_PACKAGES*, *READ_USER_DICTIONARY*) fitting in the following rule: If the Manifest file declares only one system permission and if it is *READ_LOGS*, *INSTALL_PACKAGES*, or *READ_USER_DICTIONARY*, then the application is malicious.

5.6 Layer 3: The Combination of DM and Resource Risks

An application is represented in this model as the association of the two vectors from the first two modules. That means that the vector is represented as in Table 5.15 where the first layer determines the first ten features and the second layer the last ten. We then associate the two to obtain the vector characteristics for an application in this model.

n(a,0)	n(a,1)	n(a,2)	n(a,3)	n(a,4)	n(a,5)	n(a,6)	n(a,7)	n(a,8)	n(a,9)

(a) First part of the vector

bluetooth	calendar	calls	contact	location	message	network	telephony	Wifi	Webtrace

(b) Second part of the vector

Table 5.15: Representation of the application vector in layer 3

5.7 Learning Detection

The learning dataset has been trained in an environment suitable for classification learning during this phase.

5.7.1 Environment for Learning

Within the scope of this work, we use WEKA (Figure 5.9), a renown open source software issued under the General Public License (GPL). It is a scientifically well-proved system developed by the University of Waikato in New Zealand, which implements data mining algorithms using JAVA. WEKA is a state-of-the-art collection of machine learning algorithms and their application to real-world data mining problems. It is a collection of machine learning algorithms for data mining tasks. WEKA implements algorithms for data preprocessing, classification, regression, clustering, and association rules, which are directly applied to a dataset. The data file normally used by WEKA is in the ARFF file format, which consists of special tags indicating (foremost: attribute names, attribute types, attribute values, and the data) in the data file, but it is also possible to import a .csv file.

Figure 5.9: Overview of WEKA

The next step consists of choosing the classifiers, which are able to find the class of an unknown application based on the classification algorithm.

5.7.2 Classifiers

We consider the following metrics to select the classifiers: True Positive Rate, False Positive Rate, Precision, Recall, F-Measure, and AUC.

According to the official Website of WEKA[2], there are different packages of classifiers:

- Bayes: It contains Bayesian classifiers; NaiveBayes has been chosen.
- Lazy: Learning is done here during runtime; it consists of assigning weights to neighbours as calculated per distance. The algorithm IBk has been chosen.
- Functions: Support Vector Machine, Regression algorithms and Neural networks are such examples. Classifier LibSVM has been chosen.
- Meta: Meta classifiers, which use a base classifier as input such as boosting or bagging. Classifier AdaBoostM1 has been chosen.
- Rules: Includes rule-based classifiers. Classifiers Decision Table and PART have been chosen.
- Trees: Includes tree classifiers, such as decision trees. Classifiers J48 and RandomForest have been chosen.

[2]http://www.cs.waikato.ac.nz/ml/weka/

Table 5.16 summarises statistics concerning the preliminary evaluation of the models during the learning phase. For that, every model learns the whole dataset with eight classifiers to gather the capability of recognising the class of a known application.

	Classifier	TP Rate	FP Rate	Precision	Recall	F-Measure	AUC
Layer 1	NaiveBayes	0.828	0.139	0.871	0.828	0.839	0.904
	LibSVm	0.9	0.231	0.897	0.9	0.897	0.834
	IBk	0.926	0.122	0.927	0.926	0.926	0.979
	AdaBoostM 1	0.875	0.28	0.871	0.875	0.872	0.928
	DecisionTable	0.89	0.263	0.886	0.89	0.886	0.934
	PART	0.911	0.164	0.911	0.911	0.911	0.963
	J48	0.911	0.15	0.912	0.911	0.912	0.946
	RandomForest	0.924	0.119	0.926	0.924	0.925	0.977
Layer 2	NaiveBayes	0.842	0.347	0.835	0.842	0.837	0.858
	LibSVm	0.884	0.309	0.886	0.884	0.877	0.787
	IBk	0.895	0.275	0.892	0.895	0.89	0.941
	AdaBoostM1	0.86	0.366	0.853	0.86	0.851	0.885
	DecisionTable	0.879	0.304	0.874	0.879	0.873	0.915
	PART	0.888	0.285	0.884	0.888	0.883	0.927
	J48	0.885	0.296	0.882	0.885	0.88	0.899
	RandomForest	0.894	0.275	0.891	0.894	0.889	0.94
Layer 3	NaiveBayes	0.806	0.14	0.864	0.806	0.819	0.892
	LibSVm	0.912	0.209	0.91	0.912	0.911	0.852
	IBk	0.95	0.116	0.949	0.95	0.949	0.991
	AdaBoostM1	0.879	0.272	0.875	0.879	0.876	0.932
	DecisionTable	0.898	0.261	0.895	0.898	0.894	0.948
	PART	0.935	0.153	0.934	0.935	0.934	0.979
	J48	0.926	0.168	0.925	0.926	0.925	0.957
	RandomForest	0.948	0.104	0.948	0.948	0.948	0.989

Table 5.16: Results of classification

It is clearly shown that the best classifiers are IBk, RandomForest, and PART for the three layers. The first layer assimilates with a precision of around 92% and with an AUC, which tends to 98% with these classifiers. Layer 2 is less precise with around 89%; the AUC decreased to 94%. Layer 3 is more accurate with around 95% and with an AUC nearer to 1. All models are excellently able to assimilate profiles for normal and malicious applications according to these results, because they have an AUC greater than 90% [Singh/Kaur/Malhotra 2009]. The third layer is almost perfect while assimilating application patterns. A testing and validation phase should however be done using cross validation; an implemented system is needed to confirm the performance in each layer. It is developed in the next chapter.

All models are complementary and can be combined to classify an application. The

question is, which classification algorithm should be applied when an unknown application is assigned as normal or malicious.

5.7.3 Our Classifier Algorithm

An experiment has been conducted to study different possibilities to associate the models of the different layers. For that, we proceed as following:

Step 1) Selection of the association that minimises FPR and FNR. In case of the number of FP and FN remains the same, do step 2. The objective is here to investigate whether a misclassified application in a model can be truly classified in a different one. As we have three models, there are six associations possible to check:

- Model 1 - Model 2 - Model 3: Taken misclassified applications in model 1; transfer them to model 2 to determine if they get well classified; if not then they are transferred to model 3 for the same purpose.
- Model 1 - Model 3 - Model 2: Taken misclassified applications in model 1; transfer them to model 3 to determine if they get well classified; if not then they are transferred to model 2 for the same purpose.
- Model 2 - Model 1 - Model 3: Taken misclassified applications in model 2; transfer them to model 1 to determine if they get well classified; if not then they are transferred to model 3 for the same purpose.
- Model 2 - Model 3 - Model 1: Taken misclassified applications in model 2; transfer them to model 3 to determine if they get well classified; if not then they are transferred to model 1 for the same purpose.
- Model 3 - Model 1 - Model 2: Taken misclassified applications in model 3; transfer them to model 1 to determine if they get well classified; if not then they are transferred to model 2 for the same purpose.
- Model 3 - Model 2 - Model 1: Taken misclassified applications in model 3; transfer them to model 2 to determine if they get well classified; if not then they are transferred to model 1 for the same purpose.

Six possible association sets are obtained and they provide the same outputs *GoodClassifiedPositive* and *GoodClassifiedNegative* after applying algorithm 5.2. The second step is therefore performed.

Algorithm 5.2 Selection of the association

Input: $M = model1, model2, model3$

Output:

- *GoodClassifiedPositive*: applications misclassified as malware at the beginning but finally classified as normal
- *GoodClassifiedNegative*: applications misclassified as normal at the beginning but finally classified as malware

Variables:

- $f'_p = f'_n = \emptyset$
- fp_i: Set of applications belonging to FP for the model i
- fn_i: Set of applications belonging to FN for the model i

Begin
For m in M do
  $M = M \setminus \{m\}$
  $FalsePositive = fp_m$
  $FalseNegative = fn_m$
  For n in M do
    $fp' = fp_n \cap FalsePositive$
    $GoodClassifiedPositive = GoodClassifiedPositive \cup \{FalsePositive \setminus fp'\}$
    $FalsePositive = fp'$
    $fn' = fn_n \cap FalseNegative$
    $GoodClassifiedNegative = GoodClassifiedNegative \cup \{FalseNegative \setminus fn'\}$
    $FalseNegative = fn'$
  End For
    $GoodClassifiedPositive, GoodClassifiedNegative = \emptyset$
End For
End

Step 2) Selection of the model with the best precision

Model 3 has the best precision (around 0.94 of AUC), as shown in Table 5.17; model 1 follows with around 0.92 of AUC.

The selected association is therefore model 3 - model 1 - model 2. The whole classifier for the classification of an unknown application, *app*, requires sequentially three phases.

- Phase 1: Apply model 3 to *app*. Classify the application within this model. If *app* is found as malware, we believe it is malware. If app is classified as normal, we believe it is normal. In these cases, the classifier sends the results to the displaying module. If app has a profile that is not found within the rules defined in the model, then the classifier checks it in model 1.

- Phase 2: Apply model 1 to *app*. Classify the application within this model. If *app* is found as malware, we believe it is malware. If app is classified as normal, we believe it is normal. In these cases, the classifier sends the results to the displaying module. If app has a profile that is not found within rules defined in the model, then the classifier checks it in model 2.

- Phase 3: Apply *model 2* to *app*. Classify the application within this model. If *app* is found as malware, we believe it is malware. If *app* is classified as normal in the first two steps, we believe it is normal. In these cases, then the classifier sends the results to the displaying module. If *app* has a profile that is not found within the rules defined in the model, the classifier checks if *app* matches the rule *if the Manifest file declared only one system permission and if it is READ_LOGS, INSTALL_PACKAGES, or READ_USER_DICTIONARY, then the application is malicious*. The classifier transfers it to the alerting module defined in the next section, if *app* does not match with any permission pattern until this step.

5.8 Alerting

The alerting module receives applications, which do not succeed in the classification process. The only possibility is to involve the user expressing his security point of view and defining, which resources have to be considered as sensitive and then to be protected. The module retrieves the permission requested by the application according to this information and computes the features of model 2. Depending on the result, the modules define then types of alerts to display to the user and to send them to the displaying module. The type of alerts depends on the (group of) resources selected by the user and the answer to the question determined with the help of model 2: Does the application fit to the user's security requirements?

The following resources are displayed to the user with descriptions:

- SMS/MMS: It includes resources concerning user messages.

- Contact: It includes resources concerning user contacts.

- Agenda: It includes resources concerning user events and meetings.

- Call: It includes information related to user calls: caller contact, callee contact, etc.

- Location: It includes resources concerning the user's geographic position at any time.

- Telephony state: It includes resources used to track the user with his current location, his unique device ID and his phone number. They are accessed to modify the phone state such as to shut down the device or to intercept outgoing calls.
- Network: It includes resources accessed to use the Internet. They are also requested by an application to take user information to the Internet or to transfer sensitive information from the Internet to the user device. Therefore, user information can be leaked without his/her knowledge.
- Bluetooth: It includes resources manipulated in a user's open Bluetooth network to take information to a nearer mobile device or to transfer sensitive information from a nearer mobile device to the device. Therefore, user information can be leaked without his/her knowledge.
- Wifi: It includes resources, which open the communication to the Internet or to a remote device via Wifi. Therefore, user information can be leaked without his/her knowledge.
- Information for browsing: It is information saved by the user like passwords, logins, banking codes, online payment codes, etc. when browsing in the Internet.

5.9 Displaying

This module displays understandable classification results to the user. Additionally, it presents a clear description of the resources corresponding to the alerting system.

- The first activity presents security requirements for smartphone resources to the user. The user has to select the resources to protect them primordially within the context. The concept of preferences provided by Android is used for the sake of usability.
- The second activity allows selecting one or a group of applications to scan.
- The third activity illustrates results of the classification (normal or malicious). Then it scales results within the interval [1,4] to emphasise on security risks linked to the intention of the application compared to requirements specified by the user. There are actions that the user can apply on the application: To uninstall, to remove the application, to detail, to display application details, etc.

5.10 Install-Time Scanning (ITS)

Our system provides the functionality to scan an application that is being installed automatically at install-time by this module. The user can refuse this functionality. The module listens to messages broadcasted by Android by default signaling an installation or uninstallation. When the module receives a message (broadcast intent) indicating the addition of a

new application to the system, it retrieves this message. Information on the new package is extracted from the intercepted message. The user is notified with the application class after the module has transferred the package to the classifier. It is essential to note that the ITS module is operational even when our system is currently not working.

Chapter 6

Implementation and Evaluation

The previous chapter has presented the model to classify Android applications depending on requested permissions. This model is more useful if users can manipulate it in order to analyse their applications to be protected against potential threats. This chapter presents the architecture of the implementation Detecting and Alerting System for Android Malware (DASAMoid), technologies and tools employed. It additionally depicts results of experiments made to evaluate the detection system, followed by a discussion of the results.

6.1 Architecture

DASAMoid aims to analyse and to classify Android applications installed on smartphones to prevent users from malicious traits. To achieve these objectives and to improve its usability, it incorporates connected features, which are described briefly next.

- Displaying user applications: This feature helps to retrieve and to list all user applications, which have been installed.
- Complete analysis: It helps to analyse all applications simultaneously and provides results.
- Selective analysis: The user can select applications desired to scrutinise from the list of applications shown on the screen by marking the corresponding check boxes. One can also perform the complete analysis at once through this feature by selecting all the applications. The results from the analysis and the classification of selected applications are then presented to the user.
- Automatic analysis: This functionality is executed automatically during every new installation. It allows a real-time analysis offering a tool for practical decisions to users at install time. The user can then proceed immediately to the uninstallation or to a later analysis.
- Participative analysis: The user participates by selecting deliberately the category of resources to be protected. DASAMoid computes then a risk score of the application

according to this information. This possibility involves the user complementary to the overall classification. Risk scores are, by default, those defined in the second model. If the user does not select anyone, then all resources are considered, otherwise those concerning the selected resources are considered.

- Details of Analysis Results: This functionality outputs details on results from every analysis. This functionality assists the user by combining results of participative analysis and those from selective or complete analysis. The first case is related to user security concerns whereas the second case is automatic.

6.2 General Architecture of DASAMoid

The architecture of DASAMoid is based on five modules: the module of retrieving applications, the modules of selective and complete analysis, the module of automatic analysis, the module of interpretation and presentation of results, and the module of preferences. The user interface is the way of displaying information produced by the application, and the way in which users can access application functionalities [Sommerville 2011]. Everything the users can see and interact with belongs to the user interface. Modules as well as the associated codes and some screenshots from DASAMoid are presented. Figure 6.1 depicts different modules included in the implementation as well as the interaction between these modules.

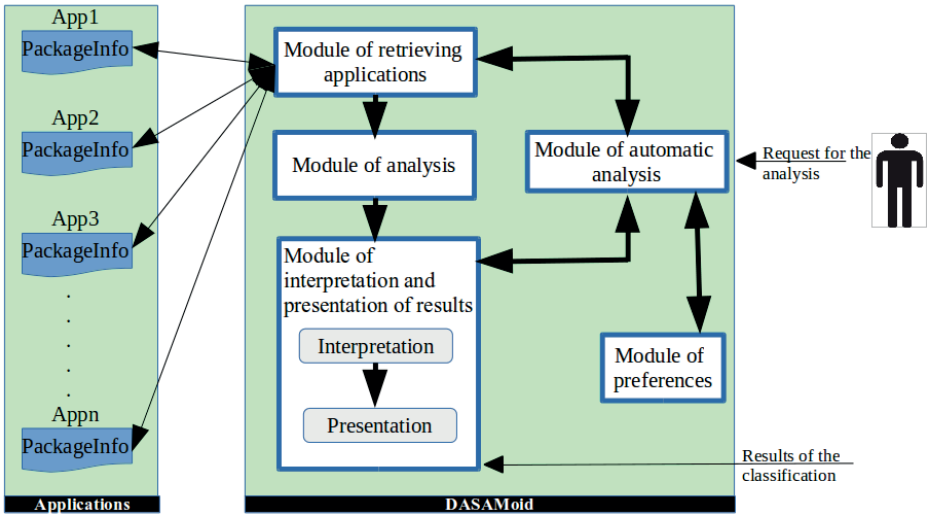


Figure 6.1: DASAMoid Architecture

6.2.1 Module Retrieving Applications

This module is responsible for the first functionality. Information on an Android package is included into a public class named *PackageInfo* associated to the application. An instance of the class *PackageInfo* includes data collected from the the Manifest. They correspond to the signature of the package, the name of the package, the name of the current version, the code of the version, the list of activities, the list of declared permissions, the list of

content providers, the list of requested permissions, the list of services, the date of the first installation, and the date of the last update. This module collects different instances of *PackageInfo* and transmits them to the module of analysis. Listing 6.1 is a portion of code to retrieve applications. The *getPackageManager()* method is getting an instance of *packageManager* responsible for application installations on Android. *PackageManager* retrieves the installed packages and stores user applications in list.

```
1  public void load() {
2  PackageManager packageManager = mContext.getPackageManager();
3  list = packageManager.getInstalledPackages(PackageManager.
       GET_PERMISSIONS);
4  if (list != null) {
5  try {
6  listApps.clear();
7  for (PackageInfo pi : list) {
8  boolean b = isSystemPackage(pi);
9  if (!b) {
10 try {
11 ajoutPackage1(pi);
12 } catch (NullPointerException e) {
13 e.printStackTrace(); }}}
14 } catch (Exception e) {
15 e.printStackTrace();
16 }}
17 if (listApps.size() == 1){
18 Toast.makeText(mContext, "One␣application␣retrieved", Toast.
       LENGTH_LONG).show();
19 }else {
20 Toast.makeText(mContext, listApps.size() + "␣applications␣
       retrieved", Toast.LENGTH_LONG).show();
21 }
22 }
```

Listing 6.1: Retrieving and displaying applications

Listing 6.2 allows to filter applications.

```
1  private boolean isSystemPackage(PackageInfo info) {
2      return ((info.applicationInfo.flags & ApplicationInfo.
           FLAG_SYSTEM) != 0)? true : false;
3  }
```

Listing 6.2: Filtering out user applications

Figure 6.3 represents a screenshot, which depicts a list of user applications. This interface appears after clicking on the *List Apps* button present in Figure 6.2.

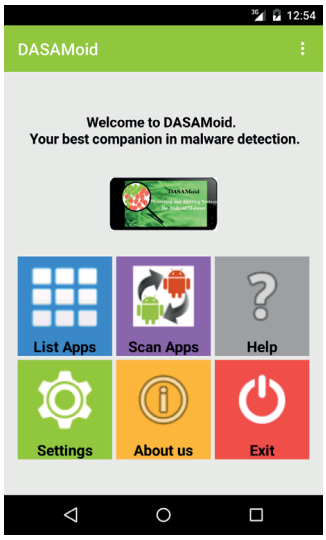


Figure 6.2: Home screen

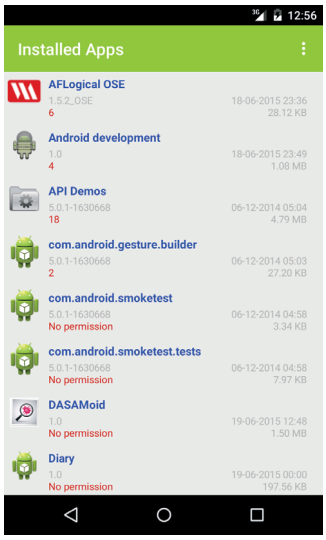


Figure 6.3: List of user applications

6.2.2 The Analysis Module

The module of analysis includes functions necessary to perform the analysis and classification of installed applications. It analyses the selective and complete by operating in three steps: gathering the list of requested permissions, formatting permissions, and analysing.

6.2.2.1 Gathering the List of Requested Permissions

The module of analysis focuses on system permissions, not the custom ones for every package on the list received from the module of retrieving applications. The list includes permissions, which follow the syntax of *android.permission.X*.

Listing 6.3 refers to portion of code used to retrieve the list of permissions requested by an application.

```
1  /*Help get the list of permissions required by an app*/
2  public String[] getPermissions(String name){
3      String[] perms = null;
4      try {
5          perms = manager.getPackageInfo(name,PackageManager.
               GET_PERMISSIONS).requestedPermissions;
6      }catch (PackageManager.NameNotFoundException e){
7          e.printStackTrace();
8      }
9      return perms;
10 }
```

Listing 6.3: Listing of required permissions

The method *getPackageInfo()* of the class *PackageManager* is used, which retrieves overall information about an application package that is installed on the system. The required information is the *requestedPermission* attribute.

6.2.2.2 Formatting Permissions

The objective of this step is to eliminate the prefix *android.permission.* from strings obtained from the step backwards in order to retain only the constant value associated to permissions i.e. remove the string *X* from *android.permission.X*. It is the function *formatListPermissions* of Listing 6.4, which takes a permission as parameter and returns only a suffix string.

```
1 public String[] formatListPermissions(String[] perms) {
2 if (perms != null) { String[] sorties = new String[perms.length];
3 String st;
4 for (int i = 0; i < perms.length; i++) { st = formatPerm(perms[i])
    ; sorties[i] = st;}
5 return sorties;
6 }else { Toast.makeText(contexte, "The list of permission is empty"
    , Toast.LENGTH_LONG).show();
7 return null;}
```

Listing 6.4: Formatting permissions

6.2.2.3 Analysis

The list of formatted permissions for each application is scrutinised and sent to the classification rules of the model TLMDASA, in this step. Classification rules as well as other patterns are obtained by applying our classifier defined in section 5.7.3. They are sequentially traversed and checked. The result of the checking is stored in a table of eleven elements as follows: the first one represents the status (0 for unclassified, 1 for normal and 2 for malicious), and the remaining elements correspond to the respective risk scores generated in model 2 following the order *Bluetooth, Calendar, Calls, Contacts, Location, Messages, Network, Telephony, Webtraces, and Wifi.* The function *analyseSingleApp* from Listing 6.5 takes an application, gets its permissions and launches the function analyse to classify the application according to *code_ci* value: 0 for unclassified, 1 for normal and 2 for malicious.

```
1 /*used to analyse a single app*/ public int[] analyseSingleApp(
    String name)verbatim
2   String[] perms = getPermissions(name);
3  int[] resultat = new int[12];
4 if (perms != null)verbatim   String[] finales = permsUtils.
    formatListPermissions(perms);
5 /*Then proceed to it analysis*/
6 resultat = analyse(finales);
7 return resultat;
```

Listing 6.5: Portion of code to analyse an application

Below are examples of analysis vectors.

1	0	0	0	0	0	0	0	0	0	0

Table 6.1: Normal application

0	0	0	0	0	0	0	0	0	0	0

Table 6.2: Unclassified application

2	0	1	0	0	0	2	0	1	1	0

Table 6.3: Malicious application

The function analyse in Listing 6.6 requires as argument the list of requested and formatted permissions of the application.

```
1  public int[] analyse(String[] permissionsFinales){
2          int[] res = {0,0,0,0,0,0,0,0,0,0,0,0};
3          int code_c3 = 0; // 0=unclassified; 1=normal; 2=malicious
4          int code_c1 = 0; // 0=unclassified; 1=normal; 2=malicious
5          int code_c2 = 0; // 0=unclassified; 1=normal; 2=malicious
6          int code_courant = 0;
7          //Elements of classification for model 1
8          int nine = 0 ;
9          int eight = 0 ;
10     //Elements of classification for model2
11         int bluetooth = 0;
12         int telephony = 0;
13         ...
14         int wifi = 0;
15     //counting each permission occurrences
16         for (String per : permissionsFinales) {
17             poidsCourant = permsUtils.getPoids(per);
18             switch (poidsCourant) {
19                             case 9:
20                     nine ++ ;
21                     break;
22                 case 2 :
23                     two ++ ;
24                     break;
25                 case 0 :
26                     zero ++ ;
27                     break;
28             } }
29 //Definition of resources scores
30         bluetooth = permsUtils.bluetooth(permissionsFinales);
31         calendar = permsUtils.calendar(permissionsFinales);
32         calls = permsUtils.calls(permissionsFinales);
33         ...
34         res[2] = bluetooth;
35         res[3] = calendar;
36         ...
37         res[8] = network;
```

Listing 6.6: Determining the attribute values for model 1 and 2

It determines then the attribute values of classification for model 1 (the number of permissions in each scale is 0 to 9) with the function *getPoids*(). It determines additionally attribute values of classification for model 2 corresponding to risk scores of sensitive resources. This is done by the functions *bluetooth()*, *calendar()*, *calls()*, *contact()*, *location()*, *message()*, *network()*, *telephony()*, *webtrace()*, and *wifi()* invoked to return risk values when the application uses respectively permissions in the categories *Bluetooth*, *Calendar*, *Calls*, *Contact*, *Location*, *Message*, *Network*, *Telephony*, *Webtrace*, and *Wifi*.

Listing 6.7 takes the generated rules from the classifier to match with vector features. We start by taking specifically model 3 to respect the order imposed by TLMDASA. Let consider the first rule of model 3. It is a conjunction of attributes of model 1 and model 2. These attributes are represented by variables *zero, four, and eight* for the first model as well as *message*, *network*, and *telephony* for the second model. If the condition if satisfied the variable *code_courant* is fixed to 2. The same principle is applied to model 1 and finally to model 2.

```
1  //Rules for classification based model 3
2          if ((eight > 0) && (zero <= 5) && (message > 0) && (
               message > 1) && (telephony > 0) &&  (network > 1) && (
               zero <= 3) && (four > 0)) {
3         code_courant = 2;
4          if(code_courant >= code_c3){
5   code_c3 = code_courant;
6            }}
7          if ((eight > 0) && (zero <= 5) && (message > 0) && (nine
               <= 0) && (calls <= 1) && (message > 1) && (eight > 1)
               && (message <= 3) && (network <= 2)) {
8             code_courant = 2;
9             if(code_courant >= code_c3){
10                code_c3 = code_courant;
11            }}
12 //Rules for classification based model 1
13         if ((eight > 0) && (zero <= 5) && (six > 0) && (zero <= 3)
               && (four <= 1) && (one <= 4) && (nine <= 0) && (zero
               <= 2)) {
14            code_courant = 2;
15            if(code_courant >= code_c1){
16
17                code_c1 = code_courant;
18            } }
19         ...
20     ...
21 //Rules for classification based model 2
22         if ((telephony > 0) && (message > 1) && (calendar <= 0) &&
               (telephony <= 1) && (bluetooth <= 0) && (network <= 2)
               && (network > 1)) {
23            code_courant = 1;
24            if(code_courant >= code_c2){
25            code_c2 = code_courant;
26            } }
27         ...
28         ...
```

Listing 6.7: Portion of code to analyse an application

The application traverses sequentially model 3, model 2, and model 1 rules to find a final decision about the class to which it belongs to, as designed in the model.

Listing 6.8 performs three tests: the first one succeeds for malicious classes because the whole code is equal to 2, which represents malicious applications. The second test succeeds for normal classes because the whole code is equal to 1, which represents normal applications and the last test concerns the case where the three models provide different classification values. The application remains unclassified, in this case but we advise the user to install antivirus software. The user can additionally see if the application meets its security requirements about sensitive resources.

```
1  int sum = 0;
2          if ((code_c3 == 2) || (code_c1 == 2) || (code_c2 == 2)){
3              sum = permissionsFinales.length;
4              res[0] = 2;
5              res[1] = sum;
6          }else {
7              if ((code_c3 == 1) || (code_c1 == 1) || (code_c2 == 1)
                   ){
8                  sum = permissionsFinales.length;
9                  res[0] = 1;
10                 res[1] = sum;
11             }else {
12                     res[0] = 0;
13                     res[1] = 0;
14             }}
15 return res;
```

Listing 6.8: Portion of code to analyse an application

Figure 6.4 represents an interface to perform the complete analysis. This interface is obtained after clicking on the button *Scan Apps* from the home screen. The item *Analyse all* is then selected to launch the complete analysis. The button *scan all now* is used to launch the complete scanning. The user can rather select the item *select apps* used to select specific applications for scanning as shown in Figure 6.5.

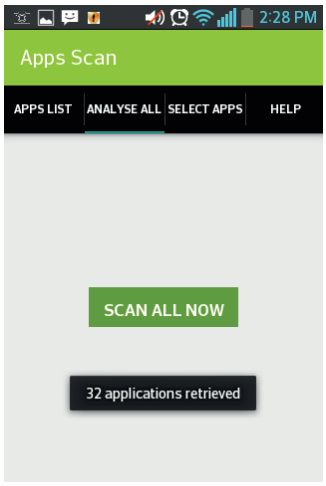


Figure 6.4: Complete analysis

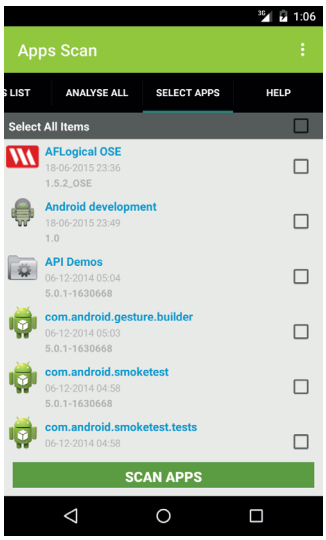


Figure 6.5: Selective analysis

There is a interface confirming the start of the analysis as shown in Figure 6.6. Figure 6.7 shows four actions that can be applied on one application.

- Scan: Analysing the application;
- Launch: Running the application;
- Unintall: Removing the installation;
- Cancel: No action.

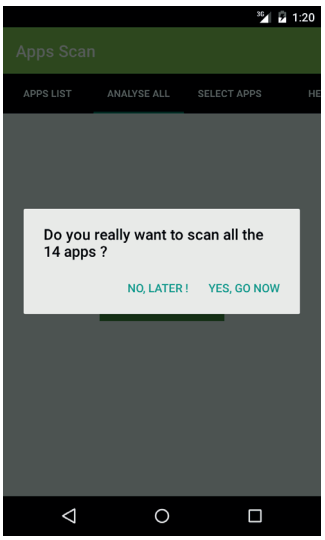


Figure 6.6: Confirmation for the analysis

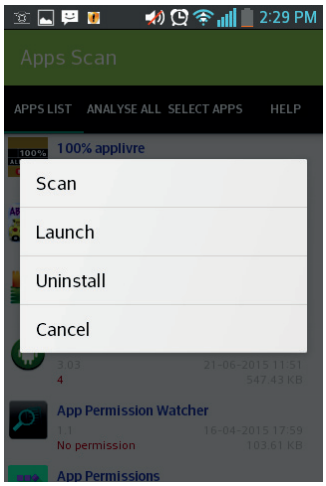


Figure 6.7: Possible operations on an application

6.2.2.4 Module Automatic Analysis

The dynamic analysis finds difficulties to be adopted on Android platforms because it often requires an important quantity memory, storage, and processor resources. Consequently, the user experience can be impacted. The alternative in DASAMoid is the automatic analysis of an application during installation. The user can decide not to use this functionality. DASAMoid listens messages broadcasted by the system by default to signal the installation or uninstallation of applications. Once this module receives a message (broadcast intent), indicating the addition of a new application to the system, it gathers the message. Information on the new package is then extracted from the intercepted message. The new application is analysed and the user is notified. The user can then check the status of the application. This module works even when DASAMoid is not running.

The Android system actually broadcasts corresponding intents to indicate actions, when a package is being added, changed or removed. There are:

- *ACTION_ PACKAGE_ADDED*
- *ACTION_ PACKAGE_CHANGED*

- *ACTION_ PACKAGE_REMOVED*

We specify the broadcast *ACTION_PACKAGE_ADDED* in our case. Our application gets then the list of permissions of the newly installed application using an instance of the *PackageManager* to get information on this application such as requested permissions. The class *AutomaticScan* extends the *BroadcastReceiver* to may perform implemented actions when the device sends a signal (Listing 6.9).

```
1  public class AutomaticScan extends BroadcastReceiver {
2  ...
3  ...
4   boolean b1 = prefs.getBoolean("pref_auto_anlyse", true);
5          permUtils = new Permission(contexte);
6          Uri data = intent.getData();
7          String packageName = data.toString().replaceAll("package:"
               , "");
8                  /*Notification*/
9          if (Intent.ACTION_PACKAGE_ADDED.equalsIgnoreCase(intent.
               getAction())) {
10              final PackageManager pm = context.getPackageManager();
11              ApplicationInfo info;
12              if (b1) {
13                  try {
14                      info = pm.getApplicationInfo(packageName,
                           PackageManager.GET_PERMISSIONS);
15                      String appName = info.loadLabel(pm).toString()
                           ;
16                                          /*notification of the
                                               addition*/
17                      Notification.Builder builder = new
                           Notification.Builder(contexte);
18                      builder.setWhen(System.currentTimeMillis())
19                              .setTicker("Ticket")
20                              .setSmallIcon(icon)
21                              .setContentTitle("DASAMoid")
22                             .setContentText(appName + "␣has␣been␣
                                   correctly␣added␣to␣the␣package␣list")
23                              .setContentIntent(pendingIntention);
24                      notifManager.notify(1234, builder.build());
```

Listing 6.9: Notifying a new installation

The function *analyseSingleApp* is then called with a formatted list of the requested permissions (Listing 6.10).

```
...
...
                                        /*Then it is the analysis
                                           */

                    String[] perms = getPermissions(packageName);
                    String st = "";
                    if (perms != null) {
                        String[] finales = permUtils.
                            formatListPermissions(perms);
                        int[] resultat = new int[2];
                        resultat = scanApps.analyseSingleApp(
                            appName);
                        if (resultat[0] == 0 || resultat[0] == 1)
                            {
                            st = appName + " is safe and without
                                risk";
                            icon =icon_bon;
                        } else {
                            st = appName + " is dangerous";
                            icon =icon_bad;

Resultat r = new Resultat(appName,resultat[0],resultat[1],resultat
    [2],resultat[3],resultat[4],resultat[5],resultat[6],resultat
    [7],resultat[8],resultat[9],resultat[10],resultat[11]);
                            ArrayList<Resultat> rr = new ArrayList
                                <Resultat>();
                            rr.add(r);

                            Intent in = new Intent(context,
                                ShowDialog.class);
                            in.putParcelableArrayListExtra("
                                dangerous",rr);
                            context.startActivity(in); } }
```

Listing 6.10: Automatic analysis of a new installation

We also declare a broadcast receiver with an intent filter in AndroidManifest.xml (Listing 6.11).

```
<receiver android:name=".utilitaries.packageaddedreceiver" >
 <intent-filter>
  <action android:name="android.intent.action.package_added"/>
  <action android:name="android.intent.action.package_changed"/>
  <action android:name="android.intent.action.package_removed"/>
  <data android:scheme="package" />
 </intent-filter>
</receiver>
```

Listing 6.11: Portion of the Manifest, which declares a receiver

6.2.2.5 Module Interpretation and Presentation of Results

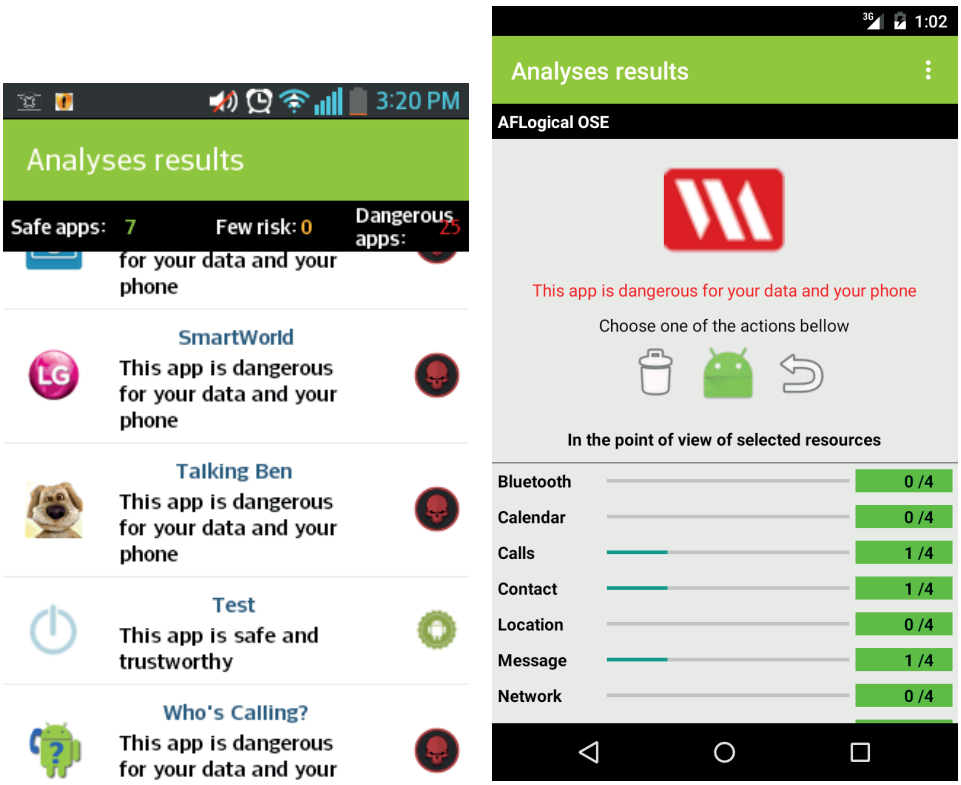
This module is responsible for the functionality *Details of Analysis Results*. It interprets results provided by the analysis module and presents them in a comprehensible manner to the user. It generates a comment, which resumes the category of the application and risks keeping it inside the device. The module finally presents results using an activity associated to an application, a scale, and an icon representing its status. This module offers the possibility to perform actions depending on the resulted status of the application. Applications classified as malicious can be removed on user demand. The user can obtain more details about results of the analysis from this module. He can consult results from the *Participative Analysis*. This module is based on alerting and on the displaying modules presented in sections 5.8 and 5.9.

Applications displayed to the user can be filtered according to the name, the size, and the date of creation (as shown in Listing 6.12).

```
1  String[] perms = getPermissions(packageName); String st = "";
2  if (perms != null) {
3  String[] finales = permUtils.formatListPermissions(perms);
4  int[] resultat = new int[2];
5  resultat = scanApps.analyseSingleApp(appName);
6  if (resultat[0] == 0 || resultat[0] == 1) {
7  st = appName + "␣is␣safe␣and␣without␣risk";
8  icon =icon_bon; } else {
9  st = appName + "␣is␣dangerous"; icon =icon_bad;
10 Resultat r = new Resultat(appName,resultat[0],resultat[1],resultat
       [2],
11 resultat[3],resultat[4],resultat[5],resultat[6],resultat[7]
12 ,resultat[8],resultat[9],resultat[10],resultat[11]);
13 ArrayList<Resultat> rr = new ArrayList<Resultat>();
14 rr.add(r); Intent in = new Intent(context, ShowDialog.class);
15 in.putParcelableArrayListExtra("dangerous",rr); context.
       startActivity(in); } }
```

Listing 6.12: Filtering criteria

Figure 6.8a depicts views for the analysis results. The first one displays applications with their status and an icon, which indicates it. The user clicks on the application icon to obtain Figure 6.8b, to go deeper into the results. This figure shows actions that the user can take according to results and to settings made on resources. A risk value under scale of 4 is displayed to indicate to the user whether his security requirements have been considered for each selected resource. The user can then decide to install.



(a) Results for all the user applications

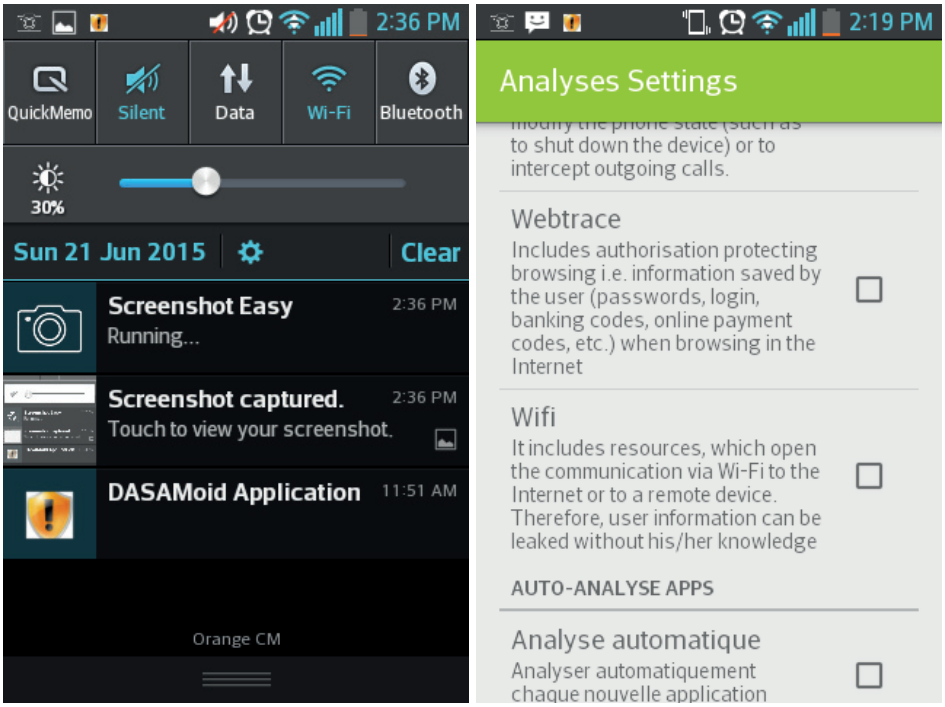
(b) Result details for one application

Figure 6.8: Scanning results

6.2.2.6 Preferences Module

Preferences help users to control some functionality of an application and fix settings such as colour, brightness, and sounds on Android. They allow to define the behaviour of an application according to some criteria. The user is invited, to specify how he will be informed about results of the automatic analysis: with a notification or with an alert dialog. Additionally, the user selects resources to evaluate application risks.

The user can activate the automatic analysis, as shown in Figure 6.9b. He is notified after a new installation or a new update (Figure 6.9a).



(a) A notification for new application (b) Settings concerning resources

Figure 6.9: Settings and preferences for DASAMoid

6.2.2.7 Fragment Concepts

The concept of fragments started from Android 3.0 (Honeycomb). We use it during the development phase. It allows performing multiple tasks by creating different activities, services or both. Fragments allows using multiple views in a single fragment activity.

Listing 6.13 represents a portion of code used by a user to confirm the complete installation. The name of a fragment is *FragmentScanAll* and extends the class *Fragment*.

```
1  public class FragmentScanAll extends Fragment {
2      public static final String TAG = "CustomListViewFragment";
3      private static final String ARG_SECTION_NUMBER = "section_number";
4      private AppList appList = null;
5      private List<String> listAnalyse = null;
6      private List<PackageInfo> listApps = null;
7       private PackageManager packageManager = null;
8       private Context context = null;
9      public  static FragmentScanAll newInstance(int sectionNumber) {
10         FragmentScanAll fragment = new FragmentScanAll();
11         Bundle args = new Bundle();
12         args.putInt(ARG_SECTION_NUMBER, sectionNumber);
13         fragment.setArguments(args);
14         return fragment;
15     }
16         ...
17 @Override
18     public void onActivityCreated(Bundle savedInstanceState) {
19         super.onActivityCreated(savedInstanceState);
20         Button button = (Button) getActivity().findViewById(R.id.button_go);
21         /*Initialize the different elements*/
22         context = getActivity();
23         packageManager = context.getPackageManager();
24         appList = AppList.getInstance(context);
25         listApps = appList.getList();
26         listAnalyse.clear();
27         Iterator<PackageInfo> it = listApps.iterator();
28         while (it.hasNext()){
29            PackageInfo packageInfo = it.next();
30             String name = packageInfo.packageName;
31             listAnalyse.add(name);         }
32         button.setOnClickListener(new View.OnClickListener() {
33             @Override
34             public void onClick(View view) {
35                 AlertDialog.Builder builder = new AlertDialog.Builder(context);
36                 builder
37                         .setTitle("Do␣you␣really␣want␣to␣scan␣all␣the␣"
38                          + listAnalyse.size() + "␣apps␣?")
39                         .setCancelable(true)
40                         .setPositiveButton("Yes,␣go␣now", new DialogInterface.
                                OnClickListener() {
41                             @Override
42                             public void onClick(DialogInterface dialogInterface, int i) {
43                                 new ScanApps(listAnalyse, packageManager, context).execute
                                        ();
44                             }})
45                         .setNegativeButton("No,␣later␣!", new DialogInterface.
                                OnClickListener() {
46                                     ...
47                             public void onClick(DialogInterface dialogInterface, int i) {
48                              startActivity(new Intent(context, Home.class)); }  })
49           .show();    }});
```

Listing 6.13: Fragment portion of code from the complete analysis

The next sections present the environment for the implementation and for the deployment.

6.3 Implementation Environment

Several tools are necessary for the implementation of DASAMoid.

6.3.1 JAVA and XML

Java is used to develop functionalities, whereas XML is used to highlight files related to application resources such as layouts, Manifest, and drawables. The Java Development Kit (JDK 6.0) has been installed, since the execution of applications including the Java code requires the presence of Java Virtual Machine (JVM). The JDK allows compiling and debugging the application code.

6.3.2 Android Studio

Android Studio is the official Integrated Development Environment (IDE), which is free of charge for Android developers. It is based on IntelliJ IDEA, an IDE that also offers a good Android development environment capable of advanced code completion, refactoring, and code analysis. The powerful code editor helps to have more productive Android application developers. Android studio is used for the development of DASAMoid. It allows to:

- Start projects using template code for patterns such as a navigation drawer and view pagers; import Google code samples from GitHub.
- Build applications for Android phones, tablets, Android wear, Android TV, Android Auto, and Google Glass;
- Create multiple APKs for your application with different features using the same project;
- Build APKs from Android Studio or the command line.

Figure 6.10, 6.11, and 6.12 give an overview of Android Studio.

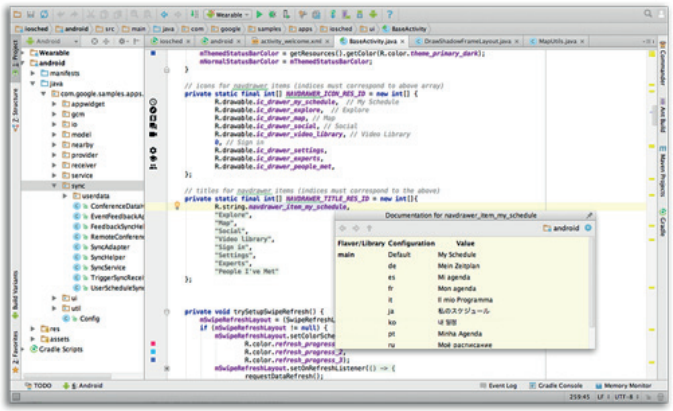


Figure 6.10: Intelligent code editor

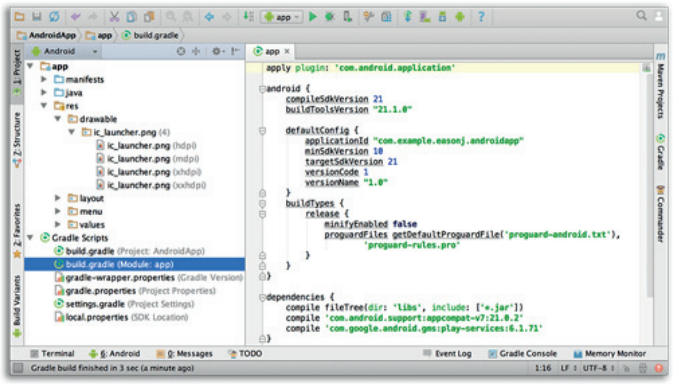


Figure 6.11: Android builds evolved, with Gradle

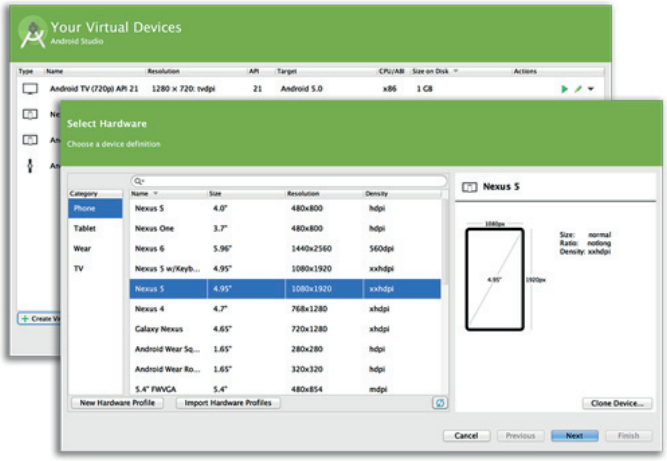


Figure 6.12: Multi-screen application development

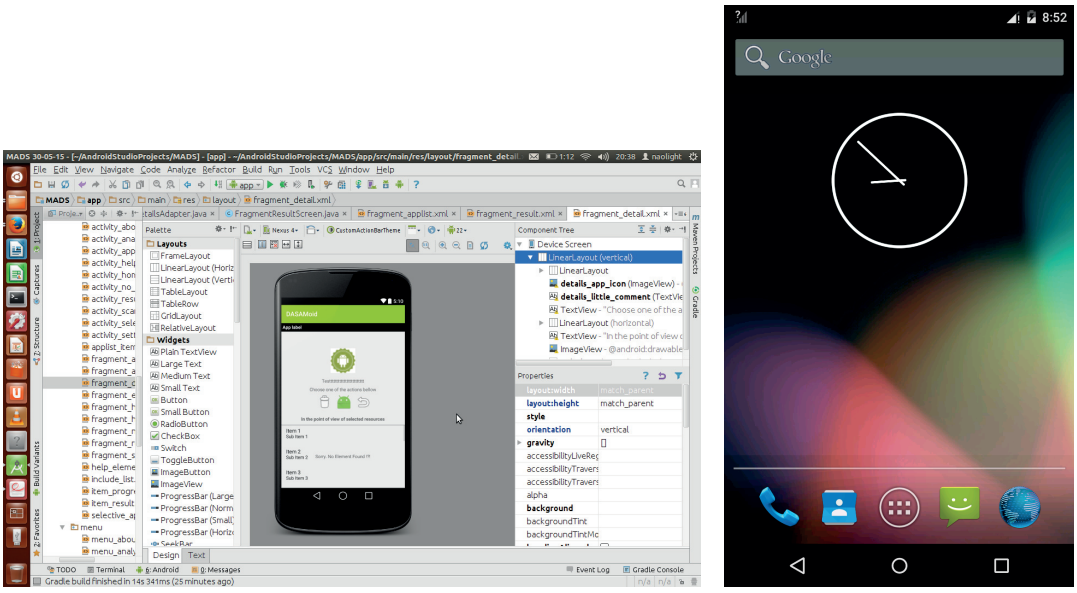
6.3.3 Android SDK

The Android Software Development Kit (SDK) is a set of development tools used to develop applications for the Android platform. The Android SDK includes the following:

- Required libraries;
- A Debugger environment Dalvik Debug Monitor Service (DDMS) with Android Debug Bridge (ADB);
- An environment for the construction of the application Android Asset Packaging Tool (AAPT);
- An emulator Android Virtual Device (AVD);
- Relevant documentation for the Android Application Programming Interfaces (APIs) ;
- Sample source code;
- Tutorials for the Android OS.

6.4 Deployment of the Environment

The tools cited in section 6.3 have been installed and configured on a DELL computer equipped with a processor Pentium ®bi-processor and a bi-core with a frequency 3.06GHZ per processor, 4GB of memory and 320GB of disk storage. The operating system is Debian Squeeze version 32 bits. The whole development and testing phase have been performed on this computer. We used the AVD to emulate Nexus 4 phone in order to visualise and test the application, while implementing (Figure 6.13b). We then deploy on real devices: a Samsung tablet with Android 4.2.2 and a LG-E440 smartphone with Android 4.1.2. The execution of DASAMoid on different Android platforms reveals no major difference. The interface should however be redefined for tablets.



(a) An overview of our Android Studio environment (b) An overview of Nexus 4 AVD

Figure 6.13: Deployment environment

6.5 Evaluation and Discussion

An empirical evaluation of the efficiency of DASAMoid and the detection model TLMDASA are presented and discussed, after presenting the implementation and the deployment environment to execute the system in detail. The evaluation consists of several objectives:

- Determine whether our learning dataset can be generalised to an unknown dataset;
- Know if the detection system is globally significant;
- Evaluate the performance of DASAMoid after deployment;
- Compare our detection system with similar ones;
- Compare our detection system with renowned antiviruses;
- Compare our results to related works.

6.5.1 Generalisation for the Learning Dataset

The first layer used in TLMDASA is based on the frequency of permissions in the dataset of normal and malicious applications. Probability to generate equal proportions has been used, because it is sometimes not evident to have the same size in both datasets. We calculate the weights of permissions based on this approach and the determination of the threshold α (it depends also on the size of datasets). We evaluate whether this method can be generally applied with an unknown dataset while keeping best results in this section. Otherwise, can the generated permission dataset associated to weights be representative in other datasets?

The methodology used to answer this question includes the steps shown in Figure 6.14.

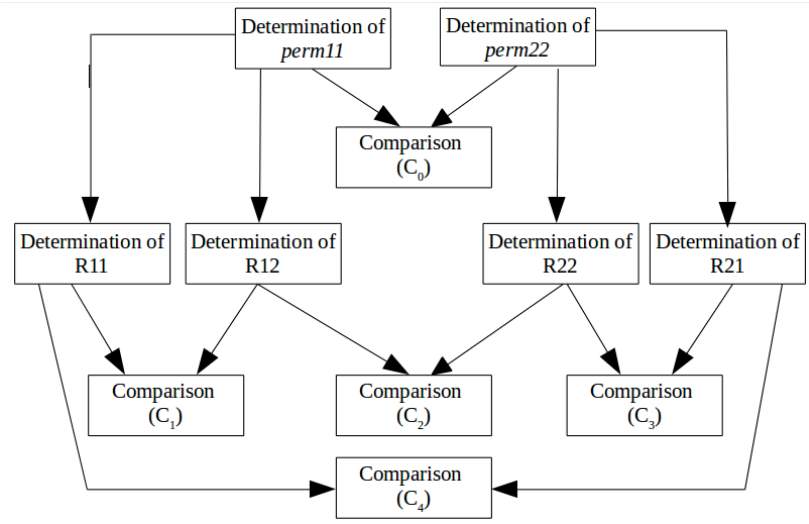


Figure 6.14: Steps to discuss the generalisation of the learning dataset

We consider the following definitions:

- *Mal*1: The sample consisting of 1260 samples of malware released by Zhou and Jiang.

- $Norm1$: The sample of 500 normal applications from Google Play. We were guided by their rating, their number of downloads and their descriptions.
- $Mal2$: The sample of 6783 pieces of malware constituted in chapter 5.
- $Norm2$: The sample of 1993 normal applications constituted in Chapter 5.
- $F1 = mal1 + norm1$, $F2 = mal2 + norm2$
- $Perm11$ is the association of permission weights determined when applying the first model to F1 (Experiment 1).
- $Perm22$ is the association of permission weights determined when applying the first model to F2 (Experiment 2).
- $R11$ represents the learning results determined by applying $perm11$ to $F1$.
- $R12$ represents the learning results determined by applying $perm11$ to $F2$ (Experiment 3).
- $R21$ represents the learning results determined by applying $perm22$ to $F1$ (Experiment 4).
- $R22$ represents the learning results determined by applying $perm22$ to $F2$.

Table 6.4 summarises the experiments.

	With probability and α	$F2$	$F1$	Without probability and α	Output Dataset of Permission- weights determined	$Perm11$	$Perm22$
Experiment 1	✓			✓	$Perm11$		
Experiment 2	✓	✓			$Perm22$		
Experiment 3		✓				✓	
Experiment 4				✓			✓

Table 6.4: Experiments for the generalisation of the learning dataset

6.5.1.1 Discussion for C_0

Appendix A illustrates $perm11$ and $perm22$. There is obviously a difference because the determination of these values relies on the size of the datasets: $F1$ is smaller than $F2$. We observe moreover some disparate values in $perm11$ and $perm22$, which demonstrate the size gap.

Figure 6.15 depicts similar tendency in different points. It indicates some stability and robustness from *model* 1.

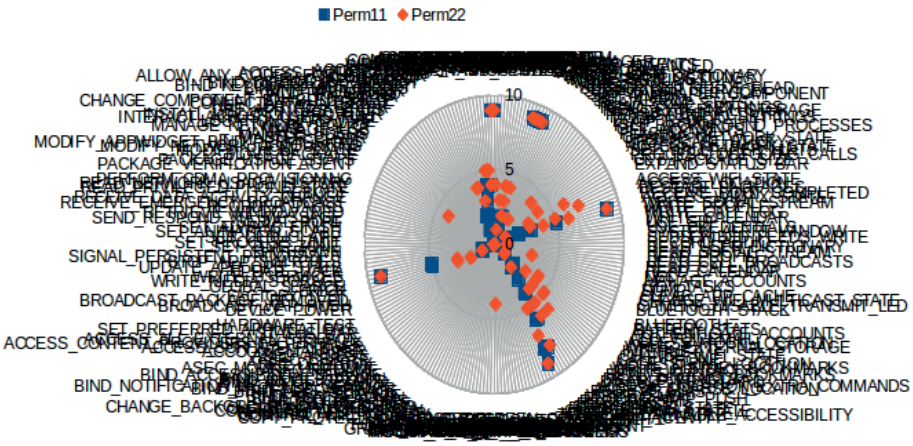
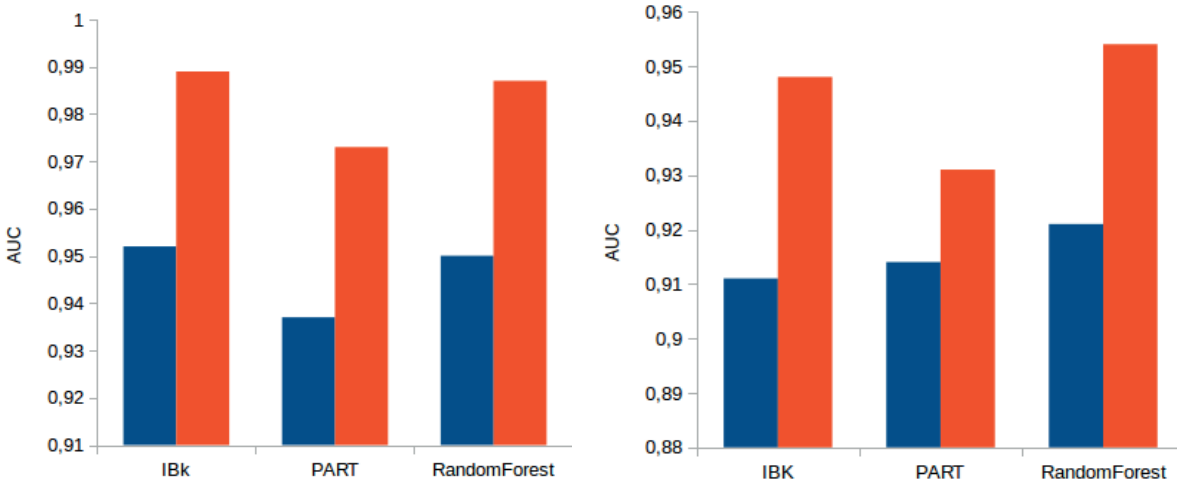


Figure 6.15: Convergence of perm11 and perm22

6.5.1.2 Discussion for C_1

The metric considered for these comparison cases is the *AUC* since it generalises performance independently to sample properties. *AUC* evolves generally with the sample while considering *perm11*. *Perm11* gives good results while it is applied to both $F1$ and $F2$, with the classifiers *IBK*, *PART*, and *RandomForest*. This classification results remain the same with the learning set and the cross validation (Figure 6.16).



(a) Using learning dataset

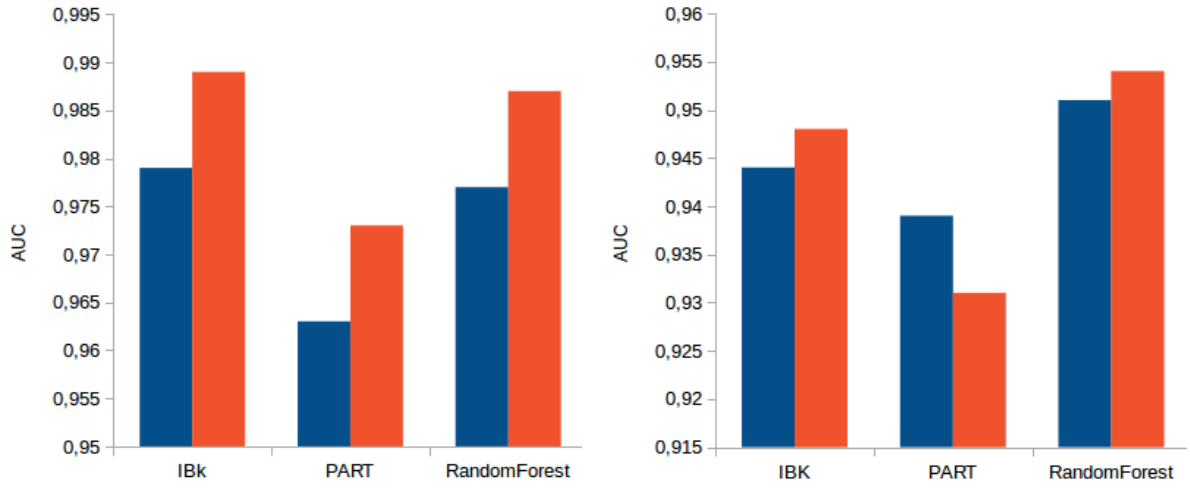
(b) Using 10-fold cross validation

Figure 6.16: Comparison between experiment 1 and 3

6.5.1.3 Discussion for C_2

We attempt to see whether $R12$ is better than $R22$, i.e. to respond to the question: is it preferable to apply *perm*11 to $F2$ or perm22 to $F2$?

Figure 6.17 indicates that better results ($R12$) are obtained when $perm11$ is used for the samples. $perm11$ generated with $F1$ is more representative that $perm22$ generated with $F2$.

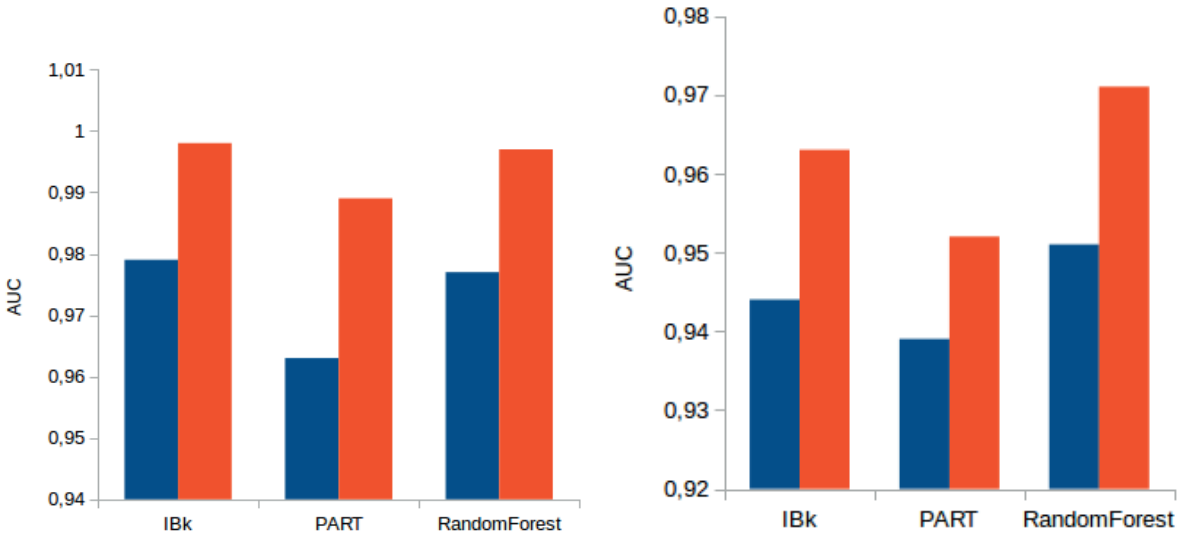


(a) Using learning dataset (b) Using 10-fold cross validation

Figure 6.17: Comparison between experiment 2 and 3

6.5.1.4 Discussion for C_3

We check here whether $R22$ is better than $R21$, i.e. the opposite of $C1$ starting from $R2$ to $R1$ as illustrated in Figure 6.18. The use of smaller dataset does not obligatory degrade the performance of the system. The model remains on the contrary exclusively robust.

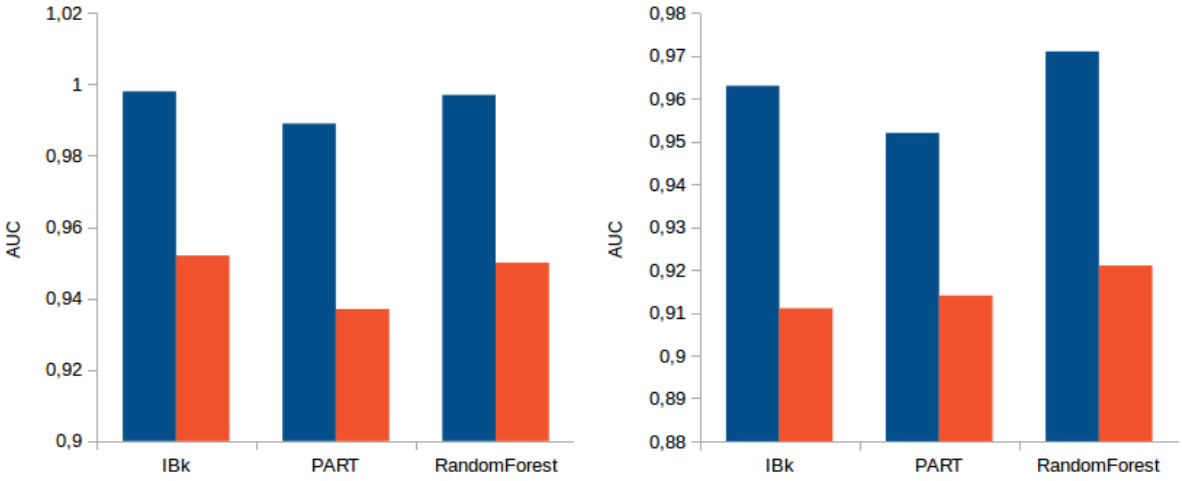


(a) Using learning dataset (b) Using 10-fold cross validation

Figure 6.18: Comparison between experiment 2 and 4

6.5.1.5 Discussion for C_4

We check here whether $R21$ is better than $R11$. To apply *perm22* to $R1$ is preferable than to apply *perm11* to $F1$ as depicted in Figure 6.19.



(a) Using learning dataset (b) Using 10-fold cross validation

Figure 6.19: Comparison experiment 4 and 1

Four considerations can be made from the previous experiments:

1. $F1$ represents more concisely malware behaviour (the recent ones).
2. *Perm*11 can be used to generalise unknown datasets.
3. The first model allows building stable and robust malware profiles.

Previous analyses are based only on the first model. Figure 6.20 reveals that *perm22* is the best representative support for training, if we consider the whole system combining *model 1* and the *model 2*. Another consideration can therefore be formulated as:

4. *Perm*22 and *perm*11 are related. The use of *perm*22 is therefore able to maintain performance. This is confirmed by Figure 6.20, which gives the tendency of the whole system combining both models.

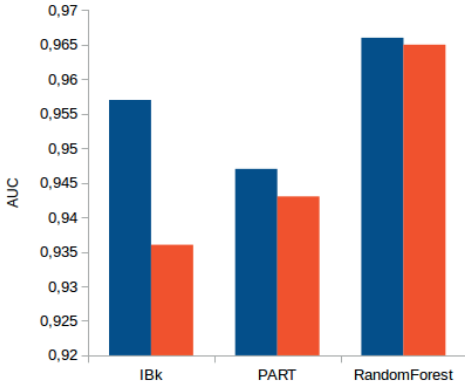


Figure 6.20: Comparison experiment 2 and 3 on the whole system

6.5.2 Performance in Detection and Prediction of TLMDASA

The first step in this section consists of evaluating the detection performance of TLMDASA on known samples provided during the training. We just consider classifiers IBK, PART and RandomForest. Table 6.5 presents the detailed results.

	TP	FN	FP	TN	TPR	FPR	Precision	ACC	AUC
IBK	6628	155	286	1707	97.7 %	14.4 %	95.86 %	94.97 %	99.1 %
PART	6589	194	378	1615	97.1 %	19,00%	94.6 %	93.48 %	97.9 %
RandomForest	6580	203	251	1742	97.00%	12.6 %	96.32 %	94.82 %	98.9 %

Table 6.5: Detection results obtained with the known dataset

TLMDASA is able to detect 97% of the malware samples used in the training with 99% of AUC. This proves that the model is outstanding with a precision of 95% at least. But, what is the situation when it has to predict the class of unknown samples? We build then experiments to evaluate the prediction performance of TLMDASA. We determine first of all the performance with the 10-fold cross validation, a case of the k-fold cross validation method, described as applying the classifier to the data 10 times, and each time with a 90-10 configuration, i.e. 90% of data for training and 10% for testing; Table 6.6 summarises the average of these 10 iterations. We keep the same metrics and the same classifier used to determine detection performance.

	TP	FN	FP	TN	TPR	FPR	Precision	ACC	AUC
IBK	6468	315	497	1496	95.4 %	24.9 %	92.9 %	90.74 %	95.7 %
PART	6427	356	418	1575	94.8 %	21,00%	93.9 %	91.18 %	94.7 %
RandomForest	6475	308	432	1561	95.5 %	21.7 %	93.7 %	91.56 %	96.6 %

Table 6.6: Prediction results simulating the unknown dataset

TLMDASA remains outstanding; it is able to detect 95 % of the malware samples with 93% precision. We randomly split the partitions into known and unknown ones. We select specifically three cases:

- Known partition (60%) and unknown one (40%),
- Known partition (66%) and unknown one (34%),
- Known partition (70%) and unknown one (30%).

We repeat them ten times and take the average results. The partitioning cases ensure that the reported results refer to the capacity of the system to predict unknown malware during the learning phase.

The results of these experiments are consigned in Table 6.7, where only the AUC metric with classifiers IBK, PART and RandomForest is considered.

Partitions	Classifier	AUC
Splitting 60-40	IBk	0.952
	PART	0.952
	RandomForest	0.964
Splitting 66-34	IBk	0.957
	PART	0.95
	RandomForest	0.965
Splitting 70-30	IBk	0.955
	PART	0.95
	RandomForest	0.966

Table 6.7: Values of AUC for every partition

The system is efficiently able to detect unknown malware with 95% to 97% of AUC, corresponding to ninety-five to ninety-seven samples of unknown malware when installing 100 applications. It is an excellent model according to [Hosner/Lemeshow/Sturdivant 2000]. The performance increases from 0.964 according to the partition size (Figure 6.21 and Figure 6.22).

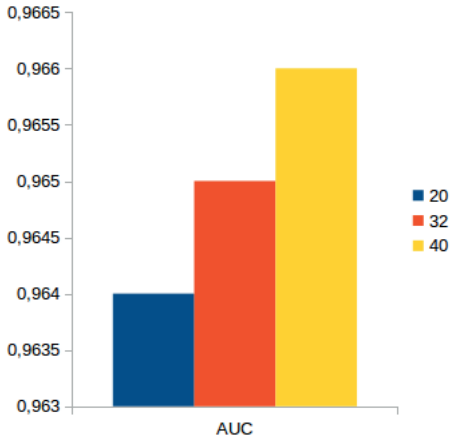


Figure 6.21: Evolution of AUC in the partitioning dataset

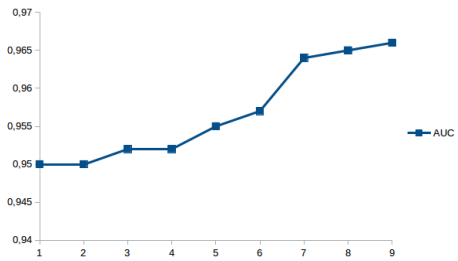


Figure 6.22: Graphical representation of Table 6.7

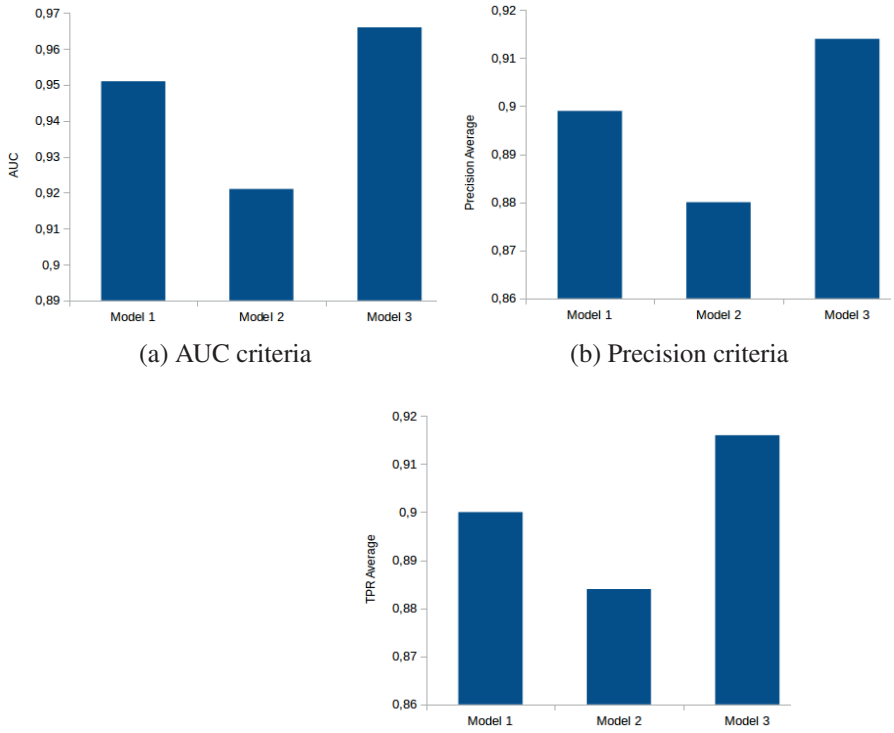
6.5.2.1 Model Validation

We collected a testing dataset including 51 malicious applications published at the end of 2014 by antivirus companies and research groups, and 34 normal applications from Google Play, to achieve the validation of TLMDASA. Normal applications have been tested in VirusTotal to confirm their normality. After eliminating duplicates and removing corrupted packages, we remain with 30 malicious applications and 33 normal applications listed in Appendix B. The results obtained are the following:

- DASAMoid detects correctly 30 pieces of malware out of 30.
- DASAMoid detects correctly 25 normal applications among 33; eight are misclassified among which are the antivirus software AVAST, AVG, McAfee, F-SECURE Mobile Security, which require accessing the user's whole sensitive information of the user: personal information (accounts, phone calls, messages, personal information, location, services that cost money), hardware information (network communication, storage, hardware controls, and system tools). They request respectively 42, 57, 70 and 59 permissions, too much for an application.

6.5.3 Comparison between Models

The objective of this section is to determine, which model offers better independently results. Figure 6.23 (a, b, c) illustrates respectively AUC, precision and TPR results on these models for the best classifier RandomForest.



(a) AUC criteria

(b) Precision criteria

(c) TPR criteria

Figure 6.23: Model 3 outperforms models 1 and 2 with RandomForest

The third model outperforms the two others, according to AUC, precision, and TPR results. We discover however an exception with classifier PART about the precision criteria. It indicates that model 1 is more precise than the others (Figure 6.24).

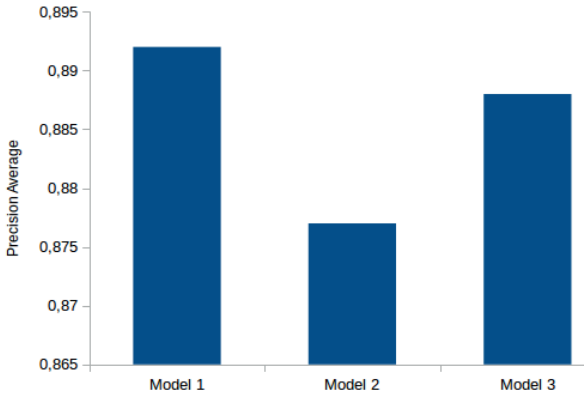


Figure 6.24: Model 1 is more precise than models 2 and 3 in PART

6.5.4 Detection of Malware Families

The malware samples of Zhou and Jiang [Zhou/Jiang 2012] reflect the behaviour of nowadays malware, according the to results found in Section 6.5.1. Another important experiment should be therefore evaluating specifically the performance detection of every sample for the forty-nine families of malware from the Genome Project. The family names and the number of samples for each family are listed in Table 6.8. The detection performance of the whole system for each family is illustrated in Figure 6.26. Our classifier is able to reliably detect all families with an average accuracy of 99.20% (1250/1260) at a false positive of 0.79% (10/1260). All families can be perfectly identified at 100%, except three of them: Asroot, Basebridge and Droiddeluxe. Basebridge shows a detection rate of more than 95.72% (112 correctly detected out of 119), Asroot shows a detection rate of 75% (6 correctly detected out of 8) and Droiddeluxe (with just one sample) cannot be detected. These families commonly rely on the root privilege to function well. They leverage known root exploits (rageagainstthecage, asroot) without asking the user to grant the root privilege to these samples to escape from the built-in security sandbox. The fact of using root exploits without user permission, escapes from our system based on static analysis. Dynamic analysis for mitigation should be associated, to scrutinise the runtime behaviour of the installed application. The system rather detects perfectly other families with samples performing privilege escalation and remote control presented in Table 6.8.

	Families	Detection(%)	Privilege escalation	Remote control
F1	ADRD	100		✓
F2	AnserverBot	100		✓
F3	Asroot	75	✓	
F4	Asroot	95.72	✓	✓
F5	BaseBridge	100		✓

	Families	Detection(%)	Privilege escalation	Remote control
F6	BeanBot	100		✓
F7	BgServ	100		✓
F8	CoinPirate	100		✓
F9	Crusewin	100		
F10	DogWars	100	✓	✓
F11	DroidCoupon	0	✓	
F12	DroidDeluxe	100	✓	✓
F13	DroidDream	100		✓
F14	DroidDreamLight	100	✓	✓
F15	DroidKungFu1	100	✓	✓
F16	DroidKungFu2	100	✓	✓
F18	DroidKungFu3	100		✓
F19	DroidKungFu4	100	✓	✓
F20	DroidKungFu5	100		
F21	DroidKungFuUpdate	100		✓
F22	Endofday	100		
F23	FakeNetflix	100		
F24	FakePlayer	100		
F25	GamblerSMS	100		✓
F26	Geinimi	100		
F27	GGTracker	100	✓	✓
F28	GingerMaster	100		✓
F29	GoldDream	100		
F30	Gone60	100		
F31	GPSSMSSpy	100		
F32	HippoSMS	100		
F33	Jifake	100		✓
F34	jSMSHider	100		✓
F35	KMin	100		
F36	Lovetrap	100		
F37	NickyBot	100		✓
F38	Nickyspy	100		✓
F39	Pjapps	100		✓
F40	Plankton	100		✓
F41	RogueLemon	100		
F42	RogueSPPush	100		
F43	SMSReplicator	100		
F44	SndApps	100		✓
F45	Spitmo	100		
F46	TapSnake	100		

	Families	Detection(%)	Privilege escalation	Remote control
F47	Walkinwat	100		✓
F48	YZHC	100	✓	
F49	Zhash	100		

Table 6.8: Malware families

Figure 6.25 summarises the detection performance of malware families.

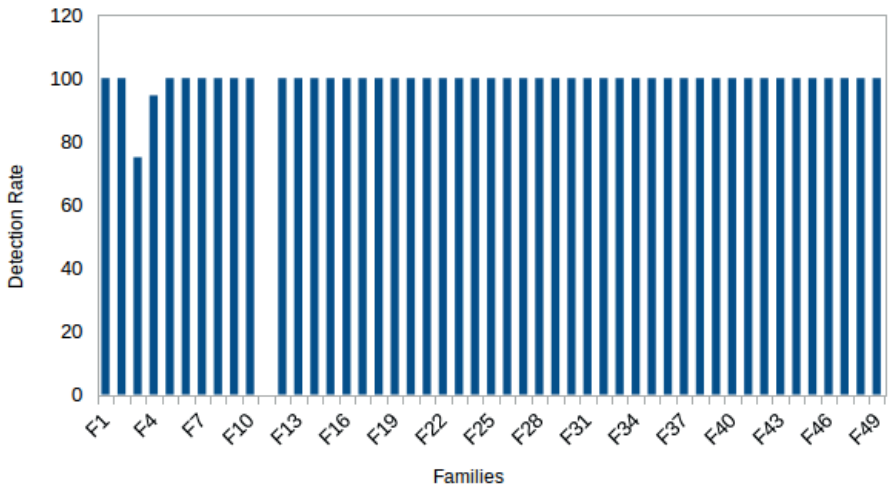


Figure 6.25: Detection per malware family

Drebin authors [Arp et al. 2014] investigated similarly the detection of malware families but only on twenty selected ones. The following point has been drawn in a conjoint comparison:

- Our system detects perfectly the Kmin family like Drebin;
- Our system outperforms Drebin in the detection of other families with 100% of the detection rate. Drebin stands with average 90% detection of those families.

6.5.5 Runtime Performance

The computing performance of mobile devices is still very low compared to Desktop computers, although it is rising very fast. Therefore, any detection model designed to be implemented directly on mobile devices must take into account this constraint to operate efficiently. We measure the average execution time required to analyse 63 applications using DASAMoid and three antiviruses on a smartphone LG-E440 including Android version 4.1.2 with a 1GHZ of processor, 2 GB internal storage and 512 MB RAM. The DASAMoid runtime is additionally measured on a SAMSUNG tablet (Case 2) including Android version 4.2.2 with a 1GHZ of processor, 1GB internal storage and 512 MB RAM with ten applications.

Five successive executions of the complete analysis on DASAMoid, have been executed on these two smartphones (SAMSUNG tablet and LG smartphone). Tests have been done with 63 applications installed on the LG-E440 smartphone and ten ones installed on the SAMSUNG tablet. DASAMoid is able to analyse with 63 applications in 500 milliseconds on average; it needs for a given application, eight milliseconds on the smartphone and two milliseconds on the tablet. Scanning 125 applications in one second on the smartphone or 500 applications in one second on the tablet is theoretically equivalent.

The analysis of 63 applications with the Virus Scanner of Avast requires on average one minute. To scan an application completely with it needs one second on average. AVG and F-SECURE analyse applications and files. They require therefore higher execution times for scanning (Table 6.9). F-SECURE particularly analyses an application in 30 seconds; AVG requires 14 seconds for an application.

These experiments show that DASAMoid requires 100 times less time than the quicker antivirus for scanning. It can thus be easily used to analyse a high quantity of information. DASAMoid does not influence the system because it mobilises only a few system resources (e.g: permissions as feature), unlikely to most of antiviruses. It is therefore an ideal candidate for the real-time analysis on application markets such as Google Play.

	DASAMoid		AVG	Avast	F-Secure
	Case 1	*Case 2*			
Time 1	609 ms	223 ms	4 minutes	1:02 minutes	9 minutes
Time 2	504 ms	224 ms	5 minutes	1:01 minutes	9 minutes
Time 3	471 ms	220 ms	3 minutes	1:05 minutes	7 minutes
Time 4	479 ms	212 ms	3 minutes	1:04 minutes	6 minutes
Time 5	470 ms	242 ms	3 minutes	0:54 seconds	8 minutes
Average Time	506 ms	224.2 ms	3 minutes 60s	1:02 minutes	Almost 8 minutes

Table 6.9: Comparison of the runtime

6.5.6 Comparison with Antivirus Scanners

We compare the detection performance of DASAMoid with the Android versions of three renowned antiviruses AVG, Avast and F-Secure. Table 6.10 shows that DASAMoid is more lightweight in terms of content; this enables a very quick detection results.

	Package size	Disk storage occupied after installation
AVG	14.80 MB	26.85 MB
Avast	10.49MB	18.29 MB
F-Secure	7.41MB	13.82 MB
DASAMoid	1.5MB	3.14 MB

Table 6.10: Comparison of the package size

It depicts details of the detection results obtained with DASAMoid, AVG, Avast and F-

Secure on 30 unknown samples of malware and 33 unknown normal applications collected for validation (Appendix B). Table 6.11 presents the detection results for DASAMoid and antiviruses.

No		AVG	AVAST	F-Secure				DASAMoid
				MP	*SP*	*FPr*	*NP*	
1	ING Bank N.V.	×	✓		●			✓
2	AlfSafe	×	✓			●		✓
3	Android System	×	✓	●				✓
4	Awesome Jokes	✓	✓				●	✓
5	BaDoink	×	✓				●	✓
6	BaseApp	×	✓			●		✓
7	Battery Doctor	×	✓			●		✓
8	Battery Improve	×	✓				●	✓
9	Black Market Alpha	×	✓			●		✓
10	Business Calendar Pro	✓	✓				●	✓
11	Chibi Fighter	×	×				●	✓
12	com.android.tools.system	×	✓			●		✓
13	Dendroid	×	✓	●				✓
14	Détecteur de Carrier IQ	×	✓				●	✓
15	FlvPlayer	×	✓	●				✓
16	Install	×	✓				●	✓
17	Jelly Matching	×	✓		●			✓
18	Mobile Security	×	✓		●			✓
19	o5android	×	✓			●		✓
20	PronEnabler	✓	✓			●		✓
21	Radardroid Pro	×	✓			●		✓
22	SberSafe	×	✓			●		✓
23	Se-Cure Mobile AV	×	✓				●	✓
24	SoundHound	×	✓		●			✓
25	SPL Meter FREE	×	✓	●				✓
26	System Service	×	✓			●		✓
27	VkSafe	×	✓			●		✓
28	41CA3EFD	×	✓		●			✓
29	sb.apk	×	✓				●	✓
30	ThreatJapan_D09	×	✓			●		✓
×Misclassified ✓Correctly classified								

Table 6.11: Detection results for unknown malware.

It is noticed that DASAMoid correctly detects 30 samples of malware whereas AVG only alerts to three samples: Awesome Jokes, Business Calendar Pro, and PronEnabler. AVG

provides therefore a TP of three, a TN of thirty-three, a FN of twenty seven and a FP of null. Avast detects twenty nine malware samples and fails to detect the Chibi Fighter malware. Avast gets therefore a TP of twenty-nine, a TN of thirty-three, a FN of one and FP of null. The applications are classified in four categories: Many privacy issues (MP), some privacy issues (SP), few privacy issues (FPr), no privacy issues (NP) concerning F-SECURE Mobile Security. Only four malware samples are correctly classified. Five samples are detected as applications with some privacy issues and twelve ones are classified as applications with few issues. F-SECURE Mobile Security incorrectly classifies nine samples as applications with NP. F-SECURE Mobile Security has thus TP of nine (we consider MP and SP as malicious classes), a TN of twenty-four (NP), FP of nine (normal applications belonging to MP, SP, and FPr), and a FN of nine (malware belonging to NP).

Table 6.12 summarises these results. AVG and Avast determine correctly the whole normal applications, whereas F-SECURE and DASAMoid have respectively 27% and 24.24% of FPR. Our scheme is the best in determining malware with 100% of TPR, followed by Avast, which fails to determine just one malware sample. The accuracy indicates that DASAMoid records the best performance after Avast. Our DASAMoid should therefore be considered as reliable compared to existing antiviruses.

	TP	FN	FP	TN	TPR	FPR	ACC
AVG	3	27	0	33	10,00%	0,00%	57.14 %
Avast	29	1	0	33	96.66%	0,00%	98.41%
F-Secure	9	21	9	24	30,00%	27,00%	52.38 %
DASAMoid	33	0	8	25	100,00%	24.24 %	88,00%

Table 6.12: Detection results

6.5.7 Comparing Different Methods

We compare the performance of our method with two specific approaches in the literature, based also on requested permissions as features. These are Kirin [Enck/Ongtang/McDaniel 2012] and from RCP+RPCP [Sarma et al. 2012]. Kirin identifies nine rules for applications to be considered as potentially malicious. The performance of RCP+RPCP is generated with rule #RCP(2)+#RPCP(1)$\geq\theta$, the best performing one. We obtain the detection performance results in terms of TPR, FPR, Precision, Accuracy and AUC, after having applied these methods to our sample datasets, all consigned in Table 6.13.

	TP	FN	FP	TN	TPR	FPR	Precision	ACC	AUC
Kirin	4076	2707	271	1722	60.09 %	13.5 %	93.76 %	57.52 %	66.9 %
#RCP(2)+#RPCP(1)$\geq\theta$	5657	1126	177	1816	83.39 %	8.88 %	96.96 %	85.15 %	58.5 %
Our scheme: TLMDASA	6580	203	251	1742	97,00%	12.6 %	96.32 %	94.82 %	99,00%

Table 6.13: Classification performances

We can observe that our method has a better classification performance than other meth-

ods. Kirin only has nine manually defined security rules, not enough to distinguish malicious applications from benign ones. #RCP(2)+#RPCP(1) uses 26 critical permissions to generate the risk signal for an application. This approach appears to be arbitrary because we can see many benign test applications are considered to be malware. Our method uses machine learning technology and captures the requested permission patterns of both benign and malicious applications. We consider security risks related to sensitive resources, besides the requested permissions; therefore we have a better performance with this combination. It is however shown that #RCP(2)+#RPCP(1) detects 74 normal applications more than TLMDASA. These two methods are similarly precise, although TLMDASA outperforms #RCP(2)+#RPCP(1) and Kirin concerning the accuracy and the AUC, giving the capacity to predict unknown samples.

6.5.8 Comparison with Related Approaches

The discussion is sub-grouped in three classes:

- Class 1: Works relying only on requested permissions like ours;
- Class 2: Approaches combining permissions with other features;
- Class 3: Approaches based on dynamic analysis.

We use 222 permissions including some from third party applications and those published on the Android Github in our work. Works in Class 1, unlike, differ from the number of selected permissions. Some rely on permissions published by Google from a fixed API level [Ping et al. 2014], and others rely on a fixed set of permissions [Enck/Ongtang/McDaniel 2012], [Wang et al. 2014], [Sarma et al. 2012]. The first option is not scalable because the Android version evolves and the second option is not precise and arbitrary. These considerations are not sufficient to identify malicious samples and are subject to false results, this is why TLMDASA considers every permission as suspicious. Class 1 works determine generally the frequency count of permissions in malicious and normal samples; the function vector associated to each sample includes 0 or 1 to refer to whether a permission is present. TLMDASA performs differently. Functions used on permissions correspond to the number of permissions at each level from 0 to 9. It also uses the XOR function on model 2 to determine the risk score of an application. Our scheme is more robust with 96.6 % of AUC [Wang et al. 2014] has an average of 90% of AUC, [Ping et al. 2014] has 94.6% of AUC. We have additionally proposed a solution that works directly on real Android smartphone. [Enck/Ongtang/McDaniel 2012] and [Rovelli/Vigfússon 2014] use a central server to perform training and learning. [Ping et al. 2014] build their own classifier called enclamad, similar to TLMDASA. However, our classifier relies on well-known learning algorithms whereas, theirs depends on contrasting patterns i.e. permissions, which do not allow to discriminate between normal and malicious application.

There exist works associating requested permissions to other features. MALDroidDe [Torregrosa 2015] gives a good performance despite the smaller samples. This work provides

an AUC of 99.4% with nearly perfect precision. However, they consider only a few amount of permissions as susceptible to harm the user and they do not validate their method with an implemented system. Our method can complete theirs because we consider every permission as risky. We have moreover determined an association of permissions weights that can be reused in MALDroidDe. Some works are close to ours in terms of AUC such as [Grampurohit/Kumar/Rawat/Rawat 2014] although they consider the category of an application as feature, which is not deterministic. This feature can give false results when an application is badly categorised. Drebin [Arp et al. 2014] and MAST [Chakradeo/Reaves/Traynor/Enck 2013] are two solutions providing respectively 94% and 99.8% of TPR, whereas DASAMoid provides 95.5 % and 97 % with larger datasets. DASAMoid is based on Drebin malware samples; DASAMoid is better in detecting malware families compared to Drebin, as demonstrated in section 6.2.4. Permissions are always present among features. This indicates their key place in the security of Android.

Research works on dynamic analyses ([Burguera/Zurutuza/Nadjm-Tehrani 2011], [Shabtai et al. 2012], [Wu et al. 2012]) are based on system calls, API calls, CPU consumption, packets sent through network, data and control flow. DASAMoid has interesting results compared to these works, although it is limited only to static analysis of the Manifest file. Shabtai et al. output TPR results: 0.825, 0.913, 0.907, and 0.999. They indicate that their approach is not efficient for predicting malware compared to the TPR of TLMDASA providing 95.5% in predicting unknown malware and 97% in detecting known one. Crowdroid [Shabtai et al. 2012] uses a crowdsourcing system to obtain the traces of the application behaviour. This approach is limited to some applications for testing. TLMDASA gives better results with larger data.

Wu et al. [Wu et al. 2012] learn with 238 Android malicious and 1500 benign applications whereas TLMDASA works with 6783 malicious and 1993 normal applications. They provide better results than TLMDASA concerning accuracy (97.87%) and precision (96.74%). However, their TPR is just of 87.39% compared to 95.5% in TLMDASA if we consider the prediction. TLMDASA is more efficient in predicting malware.

These works can however detect zero permission attacks because they study the behaviour of applications during runtime. An association of their approach with TLMDASA can provide promising results and is easier to implement than dynamic analyses.

We can summarise that the prerequisites are not only the selection of features but the good and justified selection. Features are not contributing significantly to classification and give false results, if they are not well determined. The next section elucidates limitations of this thesis.

6.5.9 Limitations

The previous evaluation demonstrates the efficacy of our method in detecting recent malware on the Android platform. DASAMoid cannot generally prohibit infections with malicious applications, since it is built on the concepts of static analysis and lacks the capabilities of a runtime analysis. Some strains of malware make use of obfuscation or load code dy-

namically, which hinders any static inspection. The performance of DASAMoid detection depends additionnally on known dataset samples used for training. The change of the input dataset can update results. TLMDASA is less accurate in the classification of normal applications than in the classification of malicious ones, according to results in section 6.2.2. Two cases can occur in terms of classification errors: A normal application may be misclassified as malicious and a malicious application may be misclassified as normal. We can consider the latter case as more crucial for our problem since we think that it is more important to prevent a malicious application reaching the device than excluding a normal application from the distribution chain.

Chapter 7

Conclusions and Perspectives

The use of smartphones and other mobile devices to manage professional and personal interaction is now ubiquitous for large enterprises and government agencies as well as for small businesses and consumers. Mobile devices mostly in form of smartphones and tablets have become the new personal computer, storing as much data as a PC but providing greater flexibility and portability. Smart devices equipped with powerful computing, sensing, and networking capabilities have increasingly become the platform of choice for many users, outselling the number of PCs worldwide.

Online banking, commerce, and other business applications put daily business and financial transactions at user fingertips. Users are at every turn stipulated to download productivity and entertainment applications for further increasing the value of their mobile devices. As mobile devices grow in popularity, so do the incentives for attackers. Mobile malware is clearly on the rise, as attackers experiment with new business models by targeting mobile phones. This increase is in some cases accompanied by sophisticated techniques purposely designed to overcome security architectures (such as the permission system) and detection mechanisms. This thesis examines the problem of such smart malware and addresses several fundamental issues when automating its analysis in large-scale scenarios. The present work provides a flexible machine learning-based mechanism to effectively detect Android malware based only on requested permissions. This chapter covers the conclusions of this dissertation. We first summarise the main contributions and discuss how they meet the established objectives. Secondly, we summarise limitations found in our system. Finally, we identify and discuss a number of challenging open issues that should be tackled in future work.

7.1 Thesis Contributions

We summarise next the contributions made in this work and discuss the main conclusions that arise from them:

1. We have explained the Android ecosystem for a better understanding of the Android security limitations in chapter 2 before going deeper into the problems of Android malware. We have dissected the application environment in terms of its architecture,

its structure, its composition in terms of components, and its version evolution from 2008 to 2015. The motivation to understand how attackers decompile an application guides us to elucidate reversing techniques that can be used for retrieving the original source code of the developer from the package. Since permissions are the key feature in our scheme and since the Android security model is based mainly on permissions, we tried to describe principles around permissions with related protection levels. The chapter explained the process from distributing to installing Android applications from Google Play or third party sources to the user device. After describing how security is implemented on Android and on Google Play, we have drawn open issues that arise from the the design of the permission model, exposing the user to malicious actions.

2. A deeper understanding of its infiltration method, its payload, its techniques are obligatory for proposing a concrete solution against malware. Malware has to be infiltrated on the user device to perform maliciously. Chapter 3 has helped to describe and understand several important aspects of the malware landscape. The first one has been to decrypt each phase of the malware life cycle from its creation to its destruction by its activation and reproduction. The different types of malware related to particular threats have then been presented. It has been shown that malware employs mostly social engineering and privilege escalation techniques to steal user information. A variety of attacks have been reported. Most of them exploit the vulnerabilities and the limitations of the Android system presented in chapter 2. The comprehensive analysis presented on the evolution of malware in smart devices motivates the need for intelligent instruments to automate their analysis. We have provided an overview of analysis models used by security researchers in current platforms for smart devices, to understand the behaviour of an application and to search for suspicious components. Static analysis is particularly the one selected in the present work. Machine learning is a set of techniques that we have used to learn and train application profiles, then to detect and to predict their status: malicious or normal. It is characterised by some performance criteria. The understanding of the previous actions and their implementation refer to chapter 4. We have additionally proposed a study of research works, which aim to detect Android malware using permissions and machine learning techniques. We have extracted advantages and limitations of these approaches.

3. Chapter 5 defines the TLMDASA system to detect Android malware based on 222 permissions and structured in three layers. The first layer is supported by a model, which focuses on the discriminating metric based on the frequency of permissions and the proportion of requests by malicious applications within the whole sample. The second layer uses a model, which relies on security risks related to granting permissions. The last layer uses a model, which characterises an application based on an association of vectors derived from the two first layers. An evaluation has shown that the four protection levels of permissions defined by Google are coarse-grained hiding the real sense of permissions. Our permission classification is fine-grained and

more precise in terms of permission semantics. We have collected a dataset, which includes a collection of 6783 cases of malware and 1993 normal applications, which have been tested and validated. Profiles for each sample have been generated for the whole system, to be used as input for learning and training. Eight classifiers have been applied to the models to output performance results. IBK, PART, and RandomForest remain the good ones; they have been used to define our classifier, which provides an outstanding performance in detection and prediction. A dataset of associations of permissions to weights has been released that can be reused in another research.

4. An efficient and lightweight implementation of TLMDASA, called DASAMoid, has been implemented that can be embedded into an Android hand-held device for real-time detection. Details of this implementation are provided in chapter 6 while specifying architecture, modules and interfaces. The evaluation provided in this chapter indicates that TLMDASA, which supports DASAMoid is one of the best tools with requested permissions as only feature. It is able to detect around 99.20% of 1260 samples of malware released by the Genome project, which represents behaviour of nowadays malware. Additionally has been found that the model is more precise and accurate than works based on dynamic analysis and those, which combine other features with permissions. Another interesting contribution is that the dataset of associations of permissions with weights generated in this study can be applied to an unknown dataset of samples to be classified while keeping good performance. Our framework is good in detecting with around 98% and in predicting with around 96% of the true positive rate. This means that it is capable to discriminate almost all cases of malware in detection and prediction. The AUC is between 97% and 99%, which confers the property of the outstanding model according to [Hosner/Lemeshow 2000].

In summary, contributions provided in this work are the following:

- A mechanism for malware detection based on the frequency of permissions and the proportion of requests by malicious applications within the whole sample;
- A mechanism for malware detection, which relies on security risks related to granting permissions;
- The TLMDASA system for malware detection, which combines the previous two;
- A representative feature dataset of 222 associations between permissions and weights, which is applicable to an unknown dataset; this classification scales from zero to nine and has been found more precise than Google's permission classification.
- An efficient and lightweight implementation of the system DASAMoid that can be installed into an Android hand-held device for real-time detection, to prevent users from malicious actions.

7.2 Thesis Limitations

DASAMoid shows a high efficacy in detecting recent real-world malware on the Android platform as demonstrated in chapter 6. However, as any tool, DASAMoid can potentially limit its effectiveness as a framework for malware detection. For instance,

- DASAMoid's detection performance critically depends on the availability of representative malicious and benign applications.
- Machine Learning techniques can help us to recognise similar and quasi-clone malware but they cannot identify completely brand new malware.
- DASAMoid as a purely static method suffers from the inherent limitations of this type of analysis. Several attacks are based on sophisticated techniques, which use reflection and encryption to become undetectable by a detection system such as DASAMoid as shown in chapter 3. DASAMoid may also fail in detecting malware that uses an updating technique as attack vector, because only the Manifest is analysed.
- DASAMoid is not able to detect zero permission attacks, in which some root exploit does not need to request any permission. Our system considers applications with no permission as normal. This could lead to false negatives.

7.3 Future Opportunities for Research

Malware in Android smartphones still poses many challenges and a number of important issues needed to be further studied and addressed with novel solutions. This section identifies some open issues where research is needed.

- Since a smartphone do not dispose enough resources for learning and determining classification rules, the framework presented here may potentially be applied to develop a Web portal malware detection system [Shiraz/Gani/Khokhar/Buyya 2013], [Shiraz/Abolfazli/Sanaei/Gani 2013]. A server will then be responsible for classifying an application's behaviour as either benign or malicious; the client is responsible for extracting the permissions declared by an application and send them to the server-side application. Then it receives an answer about the classification and visualises the result. Users can also perform the classification and visualise it directly on the desktop via this interface.
- Given that diverse variants and new types of malware are arising, additional research on combining our scheme to dynamic analysis, for example system calls, and other features should be explored. Detecting zero permission attacks should be investigated in this case.
- The present framework determines the risk of allowing permissions. Thus, it should be interesting to look at the approach pioneered by Erlingsson and Schneider [Erlingsson/Schneider 2000] called Inline Reference Monitor (IRM). The idea is to rewrite the

Manifest and other package elements of an untrusted application in such a way that the code monitoring the application is directly embedded into its code.

- Tchakounté et al. [Tchakounté/Dayang/Nlong/Check 2014] conducted a survey to examine the awareness of Cameroonian Android users about security. They found that Android users are not aware of security risks and therefore can be exposed directly to attackers because they do not take the necessary measures to protect their sensitive data. Users have different types of privacy and security concerns. Warnings will likely be more effective if they are directed to specific concerns of users about applications. Someone can worry a lot about people knowing the location of their children via shared phone, whereas another user concerns only about whether applications will excessively drain the phone's battery [Felt et al. 2012]. The challenge is to identify user concerns in Cameroon. It might be possible to learn, which warnings are relevant to particular users or classes of users in order to improve malware detection. This approach can be applied to every user in the country.

References

Print Resources

[Agrawal/Srikant 1994]
Agrawal, R. and R. Srikant: *Fast algorithms for mining association rules*. In: Bocca, J.B., M. Jarke, and C. Zaniolo (eds.), Proc. 20th Int. Conf. Very Large Data Bases (VLDB'94). Morgan Kaufmann, Burlington, MA 1994, pp. 487-499.

[Aha/Kibler 1991]
Aha, D. and D. Kibler: *Instance-based learning algorithms*. Machine Learning 6 (1991), pp. 37-66.

[Allix/Bissyande/Klein/Le Traon 2014]
Allix, K., T.F.D.A. Bissyande, J. Klein, and Y. Le Traon: *Machine Learning-Based Malware Detection for Android Applications: History Matters!*. University of Luxembourg, Walferdange 2014.

[Andrus et al. 2011]
Andrus, J., C. Dall, A.V. Hof, O. Laadan, and J. Nieh: *Cells: A virtual mobile smartphone architecture*. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11). ACM, New York, NY 2011, pp. 173-187.

[Armando/Merlo/Migliardi/Verderame 2012]
Armando, A., A. Merlo, M. Migliardi, and L. Verderame: *Would you mind forking this process? A denial of service attack on Android (and some countermeasures)*. In: Proceedings of Information Security and Privacy Research. Springer, Berlin 2012, pp. 13-24.

[Arp et al. 2014]
Arp, D., M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck: *DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket*. In: Proceedings of 17th Network and Distributed System Security Symposium (NDSS). The Internet Society, San Diego, CA 2014.

[Arzt/Rasthofer/Bodden 2013]
Arzt, S., S. Rasthofer, and E. Bodden: *Susi: A tool for the fully automated classification and categorization of Android sources and sinks*. Technical Report TUD-CS-2013-0114, EC SPRIDE, Technische Universität Darmstadt 2013.

[Aung/Zaw 2013]
Aung, Z. and W. Zaw: *Permission-Based Android Malware Detection*. International Journal of Scientific and Technology Research 2 (2013) No. 3, pp. 228-234.

[Avancini/Ceccato 2013]
Avancini, A. and M. Ceccato: *Security testing of the communication among Android applications*. In: Proceedings of the 8th International Workshop on Automation of Software Test (AST). IEEE, San Francisco, CA 2013, pp. 57-63.
[Backes et al. 2012]
Backes, M., S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky: *Appguard - real-time policy enforcement for third-party applications*. Technical Report A/02/2012, Max Planck Institute for Software Systems, Kaiserslautern 2012.
[Barrera/Kayacik/Oorschot/Somayaji 2010]
Barrera, D., 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: Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10). ACM, New York, NY 2010, pp. 73-84.
[Bergeron et al. 2001]
Bergeron, J., M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi : *Static detection of malicious code in executable programs*. In: Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS' 01). Indianapolis, IN 2001, pp. 184-189.
[Bishop 2002]
Bishop, M. A.: *The Art and Science of Computer Security*. Addison-Wesley, Boston, MA 2002.
[Bläsing et al. 2010]
Bläsing, T., A.D. Schmidt, L. Batyuk, S.-A. Camtepe, and S. Albayrak: *An Android application sandbox system for suspicious software detection*. In: Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE'2010). IEEE, Nancy 2010, pp. 55-62.
[Bloch/Wolfhugel 2006]
Bloch L. and C. Wolfhugel: *Sécurité informatique - Principes et méthodes*. Eyrolles, Paris 2006.
[Breiman 2001]
Breiman L.: *Random forests*. Machine Learning 45 (2001) No. 1, pp. 5-32.
[Bugiel et al. 2011a]
Bugiel, S., L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry: *Scalable and lightweight domain isolation on Android*. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM). ACM, New York, NY 2011.
[Bugiel et al. 2011b]
Bugiel, S., L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi: *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*. Technical Report TR-2011-04, Technische Universität Darmstadt 2011.

[Bugiel/Heuser/Sadeghi 2012]
Bugiel, S., S. Heuser, and A.R. Sadeghi: *myTunes: Semantically Linked and User-Centric Fine-Grained Privacy Control on Android*. Technical Report TUD-CS-2012-0226, Center for Advanced Security Research Darmstadt (CASED), Technische Universität Darmstadt 2012.
[Bugliesi/Calzavara/Spanò 2013]
Bugliesi, M., S. Calzavara, and A. Spanò: *Lintent: towards security type-checking of Android applications*. In: Proceedings of Formal Techniques for Distributed Systems. Springer, Berlin 2013, pp. 289-304.
[Burguera/ Zurutuza/Nadjm-Tehrani 2011]
Burguera, I., U. Zurutuza, and S. Nadjm-tehrani: *Crowdroid: behavior-based malware detection system for Android*. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices (SPSM '11). ACM, New York, NY 2011, pp. 15-26.
[Cai/Chen 2011]
Cai, L. and H. Chen: *Touchlogger: inferring keystrokes on touch screen from smartphone motion*. In: Proceedings of the 6th USENIX Conference on Hot Topics in Security, HotSec'11, USENIX Association, Berkeley, CA 2011, p. 9.
[Calandro/Stork/Gillward 2012]
Calandro, E., C. Stork, and A. Gillwald: *Internet Going - Mobile Internet access and usage in 11 African countries*. RIA Policy Brief (2012), No. 2.
[Canfora/Mercaldo/Visaggio 2013]
Canfora, G., F. Mercaldo, and C.A. Visaggio: *A Classifier of Malicious Android Applications*. In: Proceedings Eighth International Conference on Availability, Reliability and Security (ARES). IEEE 2013, pp. 607-614.
[Castillo 2011]
Castillo C.A.: *Android malware past, present, and future*. White Paper of McAfee Mobile Security Working Group, 2011.
[Castillo/Gutiérrez/Hadi 1997]
Castillo, E., J.M. Gutiérrez, and A.S. Hadi: *Expert Systems and Probabilistic Network Models*. Springer, Berlin 1997.
[Cendrowska 1987]
Cendrowska, J.: Prism: *An algorithm for inducing modular rules*. International Journal of Man-Machine Studies 27 (1987) No. 4, pp. 349-370.
[Chakradeo/Reaves/Traynor/Enck 2013]
Chakradeo, S., B. Reaves, P. Traynor, and W. Enck : *Mast: triage for market-scale mobile malware analysis.* In: Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks. ACM, New York, NY 2013, pp. 13-24.
[Chandramohan/Tan 2012]
Chandramohan, M. and H.B.K. Tan: *Detection of mobile malware in the wild*. Computer 45 (2012) No. 9, pp. 65-71.

[Chia/Yamamoto/Asokan 2012]
Chia, P.H., Y. Yamamoto, and N. Asokan: *Is this app safe? a large scale study on application permissions and risk signals*. In: Proceedings of the 21st International Conference on World Wide Web. ACM, New York, NY 2012, pp. 311-320.
[Chen et al. 2013]
Chen, K.Z., N.M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E.X. Wu, M. Rinard, and D.X. Song: *Contextual policy enforcement in Android applications with permission event graphs*. In: Proceedings of NDSS. The Internet Society, Reston, VA 2013.
[Cheng/Wong/Yang/Lu 2007]
Cheng, J., S.H. Wong, H. Yang, and S. Lu: *Smartsiren: virus detection and alert for smartphones*. In: Proceedings of the 5th International Conference on Mobile Systems, Applications and Services (MobiSys'07). ACM, New York, NY 2007, pp. 258-271.
[Chin/Felt/Greenwood/Wagner 2011]
Chin, E., A.P Felt, K. Greenwood, and D. Wagner: *Analyzing Inter-Application Communication in Android*. In: Proceedings of the 9th Annual International Conference on Mobile Systems, Applications and Services (MobiSys'11). ACM, New York, NY 2011,pp. 239-252.
[Cohen 1995]
Cohen, W.W.: *Fast effective rule induction*. In: Prieditis, A. and S. Russell (eds.), Proc. of the 12th International Conference on Machine Learning. Morgan Kaufmann, Tahoe City, CA 1995, pp. 115-123.
[Dunham 2009]
Dunham K.: *Mobile Malware Attacks and Defense*. Syngress Publishing, Rockland, MA 2009.
[Conti/Nguyen/Crispo/McDaniel 2011]
Conti, M., V. T. N. Nguyen, and B. Crispo: *CRePE: Context-related policy enforcement for Android*. In: Proceedings of the 13th Conference on Information Security (ISC'10). Springer, Berlin 2011, pp. 331-345.
[Davi/Dmitrienko/Sadeghi/Winandy 2011]
Davi, L., A. Dmitrienko, A.R. Sadeghi, and M. Winandy: *Privilege escalation attacks on Android*. In: Proceedings of Information Security. Springer, Berlin 2011, pp. 346-360.
[Davis/S/Khodaverdian/Chen 2012]
Davis, B., B. S, A. Khodaverdian, H. Chen: *I-arm-droid: A rewriting framework for in-app reference monitors for android applications*. In: Proceedings of Mobile Security Technologies 2012 (MoST 12), 2012.
[Dietz et al. 2011]
Dietz, M., S. Shekhar, Y. Pisetsky, A. Shu, and D.-S. Wallach: *Quire: lightweight provenance for smart phone operating systems*. In: Proceedings of the 20th USENIX Conference on Security (SEC'11). USENIX Association, Berkeley, CA 2011, pp. 23–23.

[Dini/Martinelli/Saracino/Sgandurra 2012]
Dini, G., F. Martinelli, A. Saracino, and D. Sgandurra: *Madam: a multi-level anomaly detector for Android malware*. In: Proceedings of the 6th International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security. MMM-ACNS-12. Springer, Berlin 2012, pp. 240-253.
[Dini et al. 2012]
Dini, G., F. Martinelli, I. Matteucci, M. Petrocchi, A. Saracino, and D. Sgandurra: *A Multi-Criteria-Based Evaluation of Android Applications*. In: Proceedings of Trusted Systems. Springer, Berlin 2012, pp. 67-82.
[Di Cerbo/Girardello/Michahelles/Voronkova 2012]
Di Cerbo, F., A. Girardello, F. Michahelles, and S. Voronkova: *Detection of malicious applications on Android OS*. In: Proceedings of Computational Forensics. Springer, Berlin 2011, pp. 138-149.
[Dixon/Jiang/Jaiantilal/Mishra 2011]
Dixon, B., Y. Jiang, A. Jaiantilal, and S. Mishra: *Location based power analysis to detect malicious code in smartphones*. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices. ACM, New York, NY 2011, pp. 27-32.
[Egners/Meyer/Marschollek 2012]
Egners, A., U. Meyer, and B. Marschollek: *Messing with Android's Permission Model*. In: Proceedings of the 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'12). IEEE, Washington, DC 2012, pp. 505-514.
[Ehringer 2010]
Ehringer, D.: *The Dalvik Virtual Machine Architecture*, Technical Report, March 2010.
[Enck/Ongtang/McDaniel 2009]
Enck, W., Ongtang, D., McDaniel P.: *On lightweight mobile phone application certification*. In: Proceedings of the 16 th ACM Conference on Computer and Communications Security. ACM, New York NY 2009, pp. 235-245.
[Enck 2011]
Enck, W.: *Defending users against smartphone apps: Techniques and future directions*. In: Proceedings of Information Systems Security. Springer, Berlin 2011, pp. 49-70.
[Enck/Octeau/McDaniel/Chaudhuri 2011]
Enck, W., D. Octeau, P. McDaniel, and S. Chaudhuri: *A Study of Android Application Security*. In: Proceedings of the 20th USENIX Security Symposium, Berkeley, CA 2011.
[Enck et al. 2010]
Enck, W., P. Gilbert, B.-G. Chun, L.P. Cox, J. Jung, P. Mcdaniel, and A.-N. Sheth: *TaintDroid: an information flow tracking system for realtime privacy monitoring on smartphones*. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10. USENIX Association, Berkeley, CA 2010, pp. 1-6.
[Erlingsson/Schneider 2000]
Erlingsson U. and F. Schneider: *IRM enforcement of Javastack inspection*. In: Proceedings of 2000 IEEE Symposium on Security and Privacy. IEEE, Oakland 2000, pp. 246-255.

[Fang/Weili/Yingjiu 2014]
Fang, Z., H. Weili, and L. Yingjiu: *Permission based Android security: Issues and countermeasures*. Computers & Security 43 (2014), pp. 205–218.
[Felt et al. 2011a]
Felt, A.P., E. Chin, S. Hanna, D. Song, and D. Wagner: *Android Permissions Demystified*. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). ACM, New York, NY 2011, pp. 627-638.
[Felt et al. 2011b]
Felt, A.P., E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner: *Android Permissions: User Attention, Comprehension, and Behavior*. In: Proceedings of the Eighth Symposium on Usable Privacy and Security (SOUPS '12). ACM, New York, NY (2012) No. 3, pp. 1-14.
[Felt et al. 2011c]
Felt, A.P., H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin: *Permission Re-Delegation: Attacks and Defenses*. In: Proceedings of the 20th USENIX Security Symposium (SEC'11). USENIX Association, Berkeley, CA 2011, pp. 1-16.
[Felt/Greenwood/Wagner 2010]
Felt, A.P., K. Greenwood, and D. Wagner: *The effectiveness of install-time permission systems for third-party applications*. Technical report, University of California at Berkeley, UCB/EECS-2010-143, Berkeley, CA 2010.
[Felt et al. 2011d]
Felt, A.P., M. Finifter, E. Chin, S. Hanna, and D. Wagner: *A survey of mobile malware in the wild*. In: Proceedings of the 1st ACM workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11). ACM, New York, NY 2011, pp. 3-14 .
[Felt/Greenwood/Wagner 2012]
Felt, A.P., S. Egelman, M. Finifter, D. Akhawe, and D. Wagner: *How to ask for permission*. In: Proceedings of USENIX Workshop on Hot Topics in Security. USENIX Association, Berkeley, CA 2012, pp. 7-7.
[Filiol 2005]
Filiol, E.: *Évaluation des logiciels antiviraux : quand le marketing s'oppose à la technique*. Journal de la securite informatique MISC 2005, No. 21.
[Fix/Hodges 1952]
Fix, E., and J.L. Hodges: *Discriminatory Analysis-nonparametric Discrimination: Small Sample Performance*. Rep. 11, Project No. 21-49-004. USAF School of Aviation Medecine, Randolph Field, Texas 1952.
[Fragkaki/Bauer/Jia/Swasey 2012]
Fragkaki, E., L. Bauer, L. Jia, and D. Swasey: *Modeling and enhancing Android's Permission System*. In: Proceedings of Computer Security–ESORICS 2012. Springer, Berlin 2012, pp. 1-18.
[Frank/Witten 1998]
Frank, E. and I.H. Witten: *Generating accurate rule sets without global optimisation*. In: Proceedings of the Fifteenth International Conference on Machine Learning (ICML' 98).

Morgan Kaufmann, San Francisco, CA 1998, pp. 144-151.
[Freund/Schapire 1999]
Freund Y. and Robert E. Schapire: *Large margin classification using the perceptron algorithm*. Machine Learning 37 (1999) No. 3, pp. 277-296.
[Breiman 2001]
Breiman L.: *Random forests*. Journal of Machine Learning 45 (2001) No. 1, pp. 5-32.
[Freund/Schapire 1996] Freund, Y. and R.E. Schapire: *Experiments with a new boosting algorithm*. In: Proceedings of the Thirteenth International Conference on Machine Learning (ICML '96). Morgan Kaufmann, San Francisco, CA 1996, pp. 148-156.
[Fritz et al. 2013]
Fritz, C., S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel: *FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps*. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14). ACM, New York, NY 2014, pp. 259-269.
[Goodman/Harris 2010]
Goodman, S. and A. Harris: *The coming African Tsunami of Information Insecurity*. Communications of the ACM 53 (2010) No.12, pp. 24-27.
[Grace et al. 2012]
Grace, M., Y. Zhou, Q. Zhang, S. Zou, and X. Jiang: *Riskranker: scalable and accurate zero-day android malware detection*. In: Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services. ACM, New York, NY 2012, pp. 281-294.
[Grace/Zhou/Wang 2012]
Grace, M., Y. Zhou, Z. Wang, and X. Jiang: *Systematic detection of capability leaks in stock Android smartphones*. In: Proceedings of the 19th Annual Symposium on Network and Distributed System Security (NDSS' 12). The Internet Society, San Diego, CA 2012.
[Grampurohit/Kumar/Rawat/Rawat 2014] Grampurohit, V., V. Kumar, S. Rawat and S. Rawat : *Category Based Malware Detection for Android*. In: Security in Computing and Communications. Springer, Berlin 2014, pp. 239-249.
[Gudeh/Pirretti/Hoeper/Buskey 2011]
Gudeth, K., M. Pirretti, K. Hoeper, and R. Buskey: *Delivering secure applications on commercial mobile devices: the case for bare metal hypervisors*. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile devices (SPSM '11). ACM, New York, NY 2011, pp. 33-38.
[Grzymala-Busse 2010]
Grzymala-Busse, J.W.: *Rule Induction. Data Mining and Knowledge Discovery Handbook*. Springer, New York, NY 2010, pp. 249-265.
[Hanley/McNeil 1982] Hanley, J.A. and B.J. McNeil: *The meaning and use of the area under a receiver operating characteristic (ROC) curve*. Radiology 143 (1982) No. 1, pp. 29-36.

[Han et al. 2013]
Han W., Z. Fang, L.T. Yang, G. Pan, and Z. Wu: *Collaborative policy administration*. IEEE Transactions on Parallel and Distributed Systems (TPDS) 25 (2014) No. 2, pp. 498-507.
[Hardy 1988]
Hardy, N.: *The confused deputy*. ACM SIGOPS Operating Systems Review 22 (1988) No. 4, pp. 36–38.
[Holte 1993]
Holte, R.C.: *Very simple classification rules perform well on most commonly used datasets*. Machine Learning 11 (1993) No 1, pp. 63–90.
[Holavanalli et al. 2013]
Holavanalli, S., D. Manuel, V. Nanjundaswamy, B. Rosenberg, F. Shen, S.Y. Ko, and L. Ziarek. *Flow permissions for Android*. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013). Palo Alto, CA 2013, pp. 652 – 657.
[Hosner/Lemeshow/Sturdivant 2013]
Hosmer, D., S. Lemeshow, and R. Sturdivant: *Applied Logistic Regression*, 3rd ed. Wiley-Blackwell, Hoboken, NJ 2013.
[Hornyack et al. 2011]
Hornyack, P., S. Han, J. Jung, S. Schechter, and D. Wetherall: *These aren't the droids you're looking for: retrofitting Android to protect data from imperious applications*. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11). ACM, New York NY 2011, pp. 639–652.
[Huang/Tsai/Hsu 2013]
Huang, C.Y., Y.T. Tsai, and C.H. Hsu: *Performance Evaluation on Permission-Based Detection for Android Malware*. In: Proceedings of Advances in Intelligent Systems and Applications. Springer, Berlin 2013, pp. 111-120.
[Jacoby/Davis 2004]
Jacoby, G.A. and N.J. Davis IV: *Battery-based intrusion detection*. In: Proceedings of the IEEE Global Telecommunications Conference 2004 (GLOBECOM'04). IEEE 2004, pp. 2250-2255.
[Jamaluddin/Zotou/Edwards/Coulton 2004]
Jamaluddin, J., N. Zotou, R. Edwards, and P. Coulton: *Mobile phone vulnerabilities: A new generation of malware*. In: Proceedings of the 2004 IEEE International Symposium on Consumer Electronics. IEEE, Reading, UK 2004, pp.199-202.
[Jeon/Kim/Lee/Won 2011]
Jeon, W., J. Kim, Y. Lee, and D. Won: *A practical analysis of smartphone security.* In: Proceedings of Human Interface and the Management of Information, Interacting with Information. Springer, Berlin 2011, pp.311-320.
[John/Langley 1995]
John, G.H. and P. Langley: *Estimating continuous distributions in Bayesian classifiers*. In: Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence (UAI'95). Morgan Kaufmann, San Francisco, CA 1995, pp. 338–345.

[Kelley/Cranor/Sadeh 2013]
Kelley, P.G., L.F. Cranor, and N. Sadeh: *Privacy as part of the app decision-making process*. In: Proceedings of the 2013 ACM Annual Conference on Human factors in Computing Systems (CHI'13). ACM, New York, NY 2013, pp. 3393-3402.
[Kelley et al. 2012]
Kelley, P.G., S. Consolvo, L.F. Cranor, J. Jung, N. Sadeh, and D. Wetherall: *A conundrum of permissions: Installing applications on an android smartphone*. In: Proceedings of Financial Cryptography and Data Security. Springer, Berlin 2012, pp. 68-79.
[Kohavi 1996]
Kohavi, R.: *Scaling up the accuracy of naive-bayes classifiers: a decision-tree hybrid*. In: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. AAAI Press, Portland, OR 1996, pp. 202-207.
[Lane 2012]
Lane, M: *Does the Android permission system provide adequate information privacy protection for end-users of mobile apps?*. In: Proceedings of the 10th Australian Information Security Management Conference (SECAU 2012). Edith Cowan University, Mount Lawley, WA 2012, pp. 67-74.
[Lange et al. 2011]
Lange, M., S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter: L4Android: *A generic Operating System framework for secure smartphones*. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11). ACM, New York, NY 2011, pp. 39-50.
[Leavitt 2005]
Leavitt, N.: *Mobile phones: the next frontier for hackers?*. Computer 38 (2005) No. 4, pp. 20–23.
[Liang/Du 2014] Liang, S. and X. Du: *Permission-combination-based scheme for Android mobile malware detection*. In: Proceedings of IEEE International Conference on Communications (ICC). Sydney 2014, pp. 2301-2306.
[Ligatti/Bauer/Walker 2005]
Ligatti, J., L. Bauer, and D. Walker: *Edit automata: Enforcement mechanisms for run-time security policies*. International Journal of Information Security 4 (2015) No 1-2, pp. 2-16.
[Littlestone 1988]
Littlestone, N.: *Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm*. Journal of Machine Learning 2 (1988) No. 4, pp. 285–318.
[Liu/Liu 2014]
Liu X. and J. Liu: *A Two-layered Permission-based Android Malware. Detection Scheme*. In: Proceedings of the 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MOBILECLOUD '14). IEEE, Washington, DC 2014, pp. 142-148.

[Marforio/Francillon/Capkun 2011]
Marforio, C., A. Francillon, and S. Capkun: *Application collusion attack on the permission-based security model and its implications for modern smartphone systems*. Technical Report, Department of Computer Science, ETH Zurich 2011.
[Marforio/Ritzdorf/Francillon/Capkun 2012]
Marforio C, H. Ritzdorf, A. Francillon, and S. Capkun: *Analysis of the communication between colluding applications on modern smartphones*. In: Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC 12). ACM, New York, NY 2012, pp. 51-60.
[Michalski/Carbonell/Mitchell 1983]
Michalski, R. S., J. G. Carbonell, and T. M. Mitchell (eds.): *Machine Learning: An Artificial Intelligence Approach*. Springer, Berlin 1983.
[Moser/Kruegel/Kirda 2007]
Moser, A., C. Kruegel, and E. Kirda: *Limits of Static Analysis for Malware Detection*. In: Proceedings of Annual Computer Security Applications Conference (ACSAC). IEEE, Miami Beach, FL 2007, pp. 421-430.
[Moonsamy/Rong/Liu 2014]
Moonsamy, V., J. Rong, and S. Liu: *Mining permission patterns for contrasting clean and malicious android applications*. Future Generation Computer Systems 36 (2014), pp. 122-132.
[Nauman/Khan/Zhang 2010]
Nauman, M., S. Khan, and X. Zhang: *Apex: extending Android permission model and enforcement with user-defined runtime constraints*. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS'10). ACM, New York, NY 2010, pp. 328-332.
[Ontgang/Butler/McDaniel 2010]
Ongtang, M., K. Butler, and P. McDaniel: *Porscha: policy oriented secure content handling in Android*. In: Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10). ACM, New York, NY 2010, pp. 221-230.
[Ongtang/McLaughlin/Enck/McDaniel 2012]
Ongtang, M., S.E. McLaughlin, W. Enck, and P. McDaniel: *Semantically rich application-centric security in Android*. In: Proceedings of the 2009 Annual Computer Security Applications Conference (ACSAC '09). IEEE, Washington, DC, 2012, pp. 658–673.
[Orthacker et al. 2012]
Orthacker, C., P. Teufl, S. Kraxberger, G. Lackner, M. Gissing, A. Marsalek, J. Leibetseder, and O. Prevenhueber: *Android Security Permissions–Can We Trust Them?*. In: Proceedings of Security and Privacy in Mobile Information and Communication Systems. Springer, Berlin 2012, pp. 40-51.

[Pearce/Felt/Nunez/Wagner 2012]
Pearce, P., A.P. Felt, G. Nunez, and D. Wagner: *AdDroid: Privilege Separation for Applications and Advertisers in Android*. In: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (AsiaCCS). ACM, New York, NY 2012, pp. 71-72.
[Pearl 1982]
Pearl, J.: Reverend Bayes on Inference Engines: *A Distributed Hierarchical Approach*. In: Proceedings of the American Association of Artificial Intelligence (AAAI' 82). AAAI Press, Portland, OR 1982, pp. 133-136.
[Peng et al. 2012]
Peng, H., C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy: *Using probabilistic generative models for ranking risks of Android apps*. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. ACM, New York, NY 2012, pp. 241-252.
[Ping et al. 2014]
Ping, X., W. Xiaofeng, N. Wenjia, Z. Tianqing, and L. Gang: *Android Malware Detection with Contrasting Permission Patterns*. Communications China 11 (2014) No. 8, pp. 1–14.
[Portokalidis/Homburg/Anagnostakis/Bos 2010]
Portokalidis, G., P. Homburg, K. Anagnostakis, and H. Bos: *Paranoid Android: Versatile protection for smartphones*. In: Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC). ACM, New York, NY 2010, pp. 347-356.
[Protsenko/Müller 2014]
Protsenko, M. and T. Müller: *Android Malware Detection Based on Software Complexity Metrics*. In: Proceedings of the 11th International Conference on Trust, Privacy and Security in Digital Business (TrustBus'14). Springer, Munich 2014, pp. 24-35.
[Quinlan 1993]
Quinlan, J.R.: *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Francisco, CA 1993.
[Rassameeroj/Tanahashi 2011]
Rassameeroj, I. and Y. Tanahashi: *Various approaches in analyzing Android applications with its permission-based security models*. In: Proceedings of 2011 IEEE International Conference on Electro/Information Technology (EIT). IEEE, Mankato, MN 2011, pp. 1-6.
[Rastogi/Chen/Jiang 2013]
Rastogi, V., Y. Chen, and X. Jiang: *Droidchameleon: evaluating android anti-malware against transformation attacks*. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. ACM, New York, NY 2013, pp. 329-334.
[Rastogi/Chen/Enck 2013]
Rastogi, V., Y. Chen, and W. Enck: *Appsplayground: Automatic security analysis of smartphone applications*. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy. ACM, New York, NY 2013, pp. 209-220.

[Roesner et al. 2012]
Roesner, F., T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan: *User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems*. In: Proceedings of 2012 IEEE Symposium on Security and Privacy (SP). IEEE, San Francisco, CA 2012, pp. 224-238.
[Rosen/Qian/Mao 2013]
Rosen, S., Z. Qian, and Z.M. Mao: *Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users*. In: Proceedings of the third ACM Conference on Data and Application Security and Privacy. ACM, New York, NY 2013, pp. 221-232.
[Rovelli/Vigfússon 2014]
Rovelli, P. and Y. Vigfússon: *PMDS: Permission-Based Malware Detection System*. In: Proceedings of the 10th International Conference on Information Systems Security (ICISS). Springer, Hyderabad, India 2014, pp. 338-357.
[Russelo/Crispo/Fernandes/Zhauniarovich 2011]
Russello, G., B. Crispo, E. Fernandes, and Y. Zhauniarovich: *YAASE: Yet Another Android Security Extension*. In: Proceedings of the third 2011 IEEE International Conference on Privacy, Security, Risk and Trust (PASSAT), and the third 2011 IEEE International Conference on Social Computing (SocialCom). IEEE, Boston, MA 2011, pp. 1033-1040.
[Sato/Chiba/Goto 2013]
Sato, R., D. Chiba, and S. Goto: *Detecting Android Malware by Analyzing Manifest Files*. In: Proceedings of the Asia-Pacific Advanced Network (Asia JCIS). IEEE, Tokyo 2013, pp. 23-31.
[Saltzer 1974]
Saltzer, J.H.: *Protection and the control of information sharing in Multics*. Communications of the ACM 17 (1974) No. 7, pp. 388-402.
[Sanz et al. 2013a]
Sanz, B., I. Santos, C. Laorden, X. Ugarte-Pedrero, P.G. Bringas, and G. Álvarez: *PUMA: Permission Usage to Detect Malware in Android*. In: Proceedings of the International Joint Conference CISIS'12-ICEUTE'12-SOCO'12 Special Sessions. Springer, Berlin 2013, pp. 289-298.
[Sanz et al. 2013b]
Sanz, B., I. Santos, C. Laorden, X. Ugarte-Pedrero, J. Nieves, P.G. Bringas, and G. Álvarez Marañón: *MAMA: Manifest Analysis For Malware Detection in Android*. Journal of Cybernetics and Systems 44 (2013), No. 6-7, pp. 469–488.
[Sanz et al. 2012]
Sanz, B., I. Santos, C. Laorden, X. Ugarte-Pedrero, and P.G. Bringas: *On the Automatic Categorisation of Android Applications*. In: Proceedings of IEEE Consumer Communications and Networking Conference (CCNC). IEEE, Las Vegas, NV 2012, pp. 149-153.

[Sarma et al. 2012]
Sarma, B.P., N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy: *Android permissions: a perspective combining risks and benefits*. In: Proceedings of the 17th ACM Symposium on Access Control Models and Technologies (SACMAT '12). ACM, New York, NY 2012, pp. 13-22.
[Sbîrlea et al. 2013]
Sbîrlea, D., M.G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar: *Automatic detection of inter-application permission leaks in Android applications*. IBM Journal of Research and Development 57 (2013) No.6, pp. 1-12.
[Schmidt et al. 2008]
Schmidt, A.D., F. Peters, F. Lamour, C. Scheel, S. Albayrak, and S. Ahmet: *Monitoring smartphones for anomaly detection*. In: Proceedings of the 1st International Conference on MOBILe Wireless Operating Systems, and Applications (MiddleWARE '08). Brussels 2007.
[Schlegel et al. 2011]
Schlegel, R., K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang: *Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones*. In: Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS). The Internet Society, San Diego, CA 2011.
[Shabtai et al. 2012]
Shabtai, A., U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss: *Andromaly: a behavioral malware detection framework for android devices*. Journal of Intelligent Information Systems 38 (2012) No. 1, pp. 161-190.
[Shabtai/Fledel/Elovici 2010]
Shabtai, A., Y. Fledel, and Y. Elovici: *Automated static code analysis for classifying android applications using machine learning*. In: Proceedings of 2010 International Conference on Computational Intelligence and Security (CIS). IEEE, Nanning 2010, pp. 329-333.
[Shankar/Karlof 2006]
Shankar, U. and C. Karlof: *Doppelganger: Better browser privacy without the bother*. In: Proceedings of the 13th ACM Conference on Computer and Communications Security. ACM, New York, NY 2006, pp. 154-167.
[Sheng/Ling 2005]
Sheng S. and C.X. Ling: *Hybrid Cost-sensitive Decision Tree*. In: Proceedings of 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD 2005). Springer, Porto 2005.
[Shin et al. 2010]
Shin, W., S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka: *A small but non-negligible flaw in the android permission scheme*. In: Proceedings of the 2010 IEEE International Symposium on Policies for Distributed Systems and Networks (PILICY). IEEE, Fairfax, VA 2010, pp. 107-110.

[Shiraz/Gani/Khokhar/Buyya 2013]
Shiraz, M., A. Gani, R.H. Khokhar, and R. Buyya: *A Review on Distributed Application Processing Frameworks in Smart Mobile Devices for Mobile Cloud Computing*. IEEE Communications Surveys and Tutorials 15 (2013) No.3, pp. 1294-1313.
[Shiraz/Abolfazli/Sanaei/Gani 2013]
Shiraz, M., S. Abolfazli, Z. Sanaei, and A. Gani: *A study on virtual machine deployment for application outsourcing in mobile cloud computing*. The Journal of Supercomputing 63 (2013) No. 3, pp. 946-964.
[Sommerville 2011]
Sommerville, I.:
Software Engineering. Pearson, 9th rev. ed., Amsterdam 2011.
[Spreitzenbarth 2013]
Spreitzenbarth, M.: *Dissecting the Droid: Forensic Analysis of Android and its malicious Applications*. PhD Thesis, University of Erlangen-Nürnberg 2013.
[Struse et al. 2012]
Struse, E., J. Seifert, S. Uellenbeck, E. Rukzio, and C. Wolf: *PermissionWatcher: Creating User Awareness of Application Permissions in Mobile Systems*. In: Proceedings of Ambient Intelligence. Springer, Berlin 2012, pp. 65-80.
[Su/Chang 2014]
Su, M.-Y. and W.-C. Chang: *Permission-based Malware Detection Mechanisms for smartphones*. In: Proceedings of IEEE 2014 International Conference on Information Networking (ICOIN). Phuket, Thailand 2014, pp. 449-452.
[Sven et al. 2011]
Sven, B., D. Lucas, D. Alexandra, H. Stephan, S. Ahmad-reza, and S. Bhargava: *Practical and lightweight domain isolation on Android*. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11). ACM, New York, NY 2011, pp. 51-62.
[Talha/Alper/Aydin 2015]
Talha, K. A., D. I. Alper, and C. Aydin: *APK Auditor: Permission-based Android malware detection system*. Digital Investigation 13 (2015), pp. 1-14.
[Tan/Steinbach/Kumar 2005]
Tan, P.-N., M. Steinbach, and V. Kumar: *Introduction to Data Mining*. Addison-Wesley, Boston, MA 2005.
[Tang/Jin/He/Jiang 2011]
Tang, W., G. Jin, J. He, and X. Jiang: *Extending Android security enforcement with a security distance model*. In: Proceedings of 2011 International Conference on Internet Technology and Applications (iTAP). IEEE 2011, pp. 1-4.
[Tchakounté/Dayang 2013a]
Tchakounté, F. and P. Dayang: *Qualitative Evaluation of Security Tools for Android*. International Journal of Science and Technology 2 (2013) No. 11, pp. 754-838.

[Tchakounté/Dayang 2013b]
Tchakounté, F. and P. Dayang: *System Call Analysis of Malware on Android*. International Journal of Science and Technology 2 (2013) No. 9, pp. 669-674.
[Tchakounté/Dayang/Nlong/Check 2014]
Tchakounte, F., P. Dayang, J. M. Nlong, and N. Check. *Understanding of the Behaviour of Android Smartphone Users in Cameroon: Application of Security*. Open Journal of Information Security and Applications 1 (2014) No. 2, pp. 9-20.
[Torregrosa 2015]
Torregrosa, B.: *A framework for detection of malicious software in Android handheld systems using machine learning techniques*. Master thesis, Universitat Oberta de Catalunya, Barcelona 2015.
[Vapnik 2000]
Vapnik, V.: *The Nature of Statistical Learning Theory.* Information Science and Statistics. Springer, Berlin 2000.
[Vidas/Zhang/Christin 2011]
Vidas, T., C. Zhang, and N. Christin: *Towards a general collection methodology for Android devices*. The International Journal of Digital Forensics & Incident Response 8 (2011), pp. 14-24.
[Vidas/Votipka/Christin 2011]
Vidas, T., D. Votipka, and N. Christin: *All your droid are belong to us: a survey of current Android attacks*. In: Proceedings of the 5th USENIX Workshop on Offensive Technologies (WOOT'11). USENIX Association, Berkeley, CA 2011, pp. 81-90.
[Vidas/Christin/Cranor 2011]
Vidas, T., N. Christin, and L. Cranor: *Curbing Android permission creep*. In: Proceedings of the 2011 Web 2.0 Security and Privacy Workshop (W2SP 2011). Oakland, CA 2011.
[Vennon/Stroop 2010]
Vennon T. and D. Stroop: *Android Market: Threat analysis of the Android Market*. White paper, Smobile Systems 2010.
[Wain/Au/Zhou/Huang/Lie 2011]
Wain, K., Y. Au, Y.F. Zhou, Z. Huang, and D. Lie: *PScout: Analyzing the Android Permission Specification*. In: Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12). ACM, New York, NY 2012, pp. 217-228.
[Wang et al. 2014]
Wang, W., X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang: *Exploring Permission-Induced Risk in Android Applications for Malicious Application Detection*. IEEE Transactions on Information Forensics and Security 9 (2014) No. 11, pp. 1869-1882.
[Wei/Gomez/Neamtiu/Faloutsos 2012a]
Wei, X., L. Gomez, I. Neamtiu, and M. Faloutsos: *Malicious Android applications in the enterprise: What do they do and how do we fix it?*. In: Proceedings of 2012 IEEE 28th International Conference on Data Engineering Workshops (ICDEW). IEEE, 2012, pp. 251-254.

[Wei/Gomez/Neamtiu/Faloutsos 2012b]
Wei, X., L. Gomez, I. Neamtiu, and M. Faloutsos: *ProfileDroid: multi-layer profiling of Android Applications*. In: Proceedings of the 18th annual International Conference on Mobile Computing and Networking (Mobicom'12). ACM, New York, NY 2012, pp.137-148.
[Wei/Gomez/Neamtiu/Faloutsos 2012c]
Wei, X., L. Gomez, I. Neamtiu, and M. Faloutsos: *Permission evolution in the Android Ecosystem*. In: Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12). ACM, New York, NY 2012, pp. 31-40.
[Witten/Eibe/Hall 2011]
Witten, I. H., F. Eibe, and M.A. Hall: *Data Mining Practical Machine Learning Tools and Techniques (Third Edition)*. Morgan Kaufmann, San Francisco, CA 2011.
[Wu/Mao/Wei/Lee 2012]
Wu, D.J., C.H. Mao, T.E. Wei, H.M Lee, and K.P. Wu: *Droidmat: Android malware detection through manifest and API calls tracing*. In: Proceedings of 2012 Seventh Asia Joint Conference on Information Security (Asia JCIS). IEEE, 2012, pp. 62-69.
[Xie/Zhang/Seifert/Zhu 2010]
Xie, L., X. Zhang, J.-P. Seifert, and S. Zhu: *pBMDS: A behavior-based malware detection system for cellphone devices*. In: Proceedings of the third ACM Conference on Wireless Network Security (WiSec'10). ACM, New York, NY 2010, pp. 37-48.
[Xu/Saïdi/Anderson 2012]
Xu, R., H. Saïdi, and R. Anderson: *Aurasium: Practical policy enforcement for Android Applications*. In: Proceedings of the 21st USENIX Security Symposium (Security'12). USENIX Association, Berkeley, CA 2012, pp. 27-27.
[Yadav et al. 2011]
Yadav, K., P. Kumaraguru, A. Goyal, A. Gupta, and V. Naik: *SMSAssassin: Crowdsourcing driven mobile-based system for SMS spam filtering*. In: Proceedings of the 12th Workshop on Mobile Computing Systems & Applications (HotMobile'11). ACM, New York, NY 2011, pp. 1-6.
[Yang et al. 2012]
Yang, L., N. Boushehrinejadmoradi, P. Roy, V. Ganapathy, and L. Iftode: *Enhancing user's comprehension of Android Permissions*. In: Proceedings of the second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. ACM, New York, NY 2012, pp. 21-26.
[Zhauniarovich 2014]
Zhauniarovich, Y.: *Improving the security of the Android ecosystem*. PhD thesis, University of Trento, Trento 2014.
[Zhang et al. 2013]
Zhang, Y., 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: Proceedings of the ACM SIGSAC Conference on Computer & Communications Security (CCS '13). ACM, New York, NY 2013, pp. 611-622.

[Zheng/Lee/Lui 2013]
Zheng, M., P.P. Lee, and J.C. Lui: *Adam: An automatic and extensible platform to stress test android anti-virus systems*. In: Proceedings of Detection of Intrusions, Malware and Vulnerability Assessment. Springer, Berlin 2013, pp. 82-101.
[Zhou/Jiang 2012]
Zhou, Y. and X. Jiang: *Dissecting Android Malware: Characterization and Evolution*. In: Proceedings of IEEE Symposium on Security and Privacy (SP'12). IEEE, Washington, DC 2012, pp. 95-109.
[Zhou/Jiang 2013]
Zhou, Y. and X. Jiang: *Detecting passive content leaks and pollution in Android applications*. In: Proceedings of the 20th Annual Symposium on Network and Distributed System Security (NDSS'13). The Internet Society, San Diego, CA 2013, pp. 1-16
[Zhou/Zhang/Jiang 2011]
Zhou, Y., X. Zhang, X. Jiang, and V.W. Freeh: *Taming information-stealing smartphone applications (on Android)*. In: Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST'11). Springer, Berlin 2011, pp. 93-107.
[Zhou/Zhou/Jiang/Ning 2012]
Zhou, W., Y. Zhou, X. Jiang, and P. Ning: *Detecting repackaged smartphone applications in third-party Android marketplaces*. In: Proceedings of the second ACM Conference on Data and Application Security and Privacy. ACM, New York, NY 2012, pp. 317-326.
[Zhou/Wang/Zhou/Jiang 2012]
Zhou, Y., Z. Wang, W. Zhou, and X. Jiang: *Hey, you, get off of my Market: Detecting malicious apps in official and alternative Android Markets*. In: Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS' 2012). The Internet Society, San Diego, CA 2012, pp. 5-8.

Online Sources

[African Union 2014]
African Union: *The Member States of the African Union: African Union Convention on Cyber Security and Personal Data Protection*, 2014.
http://pages.au.int/sites/default/files/en_AU%20Convention%20on%20CyberSecurity%20Pers%20Data%20Protec%20AUCyC%20adopted%20Malabo.pdf. Visited: July 2014.
[AAC 2015]
AAC: *Africa Android Challenge*. http://androidchallenge.org/. Visited: January 2015.
[Abyster 2014]
Abyster: *Solution de paiement en ligne*. www.abyster.com/pages/home.html. Visited: March 2014.
[Afriland First Bank 2013]
Afriland First Bank: *Mobile Money*. https://www.afrilandfirstbank.com/mobile-money.html. Visited: November 2013.

[Android 2014]
Android: *Gmail - Android Market*.
https://web.archive.org/web/20110328201807/https://market.android.com/details?id=com.google.android.gm. Visited: March 2014.
[Android 2015]
Android: *Manifest Permissions*.
http://developer.android.com/reference/android/Manifest.permission.html. Visited: January 2015.
[Android Developers 2014]
Android Developers: *Signing your applications*.
http://developer.android.com/tools/publishing/app-signing.html. Visited: February 2014.
[Android Forums 2010]
Android Forums: *Rooting the Droid without rsd lite up to and including FRG83D*.
http://androidforums.com/droid-all-things-root/171056-rooting-droid-without-rsd-lite-up-including-frg83d.html. Visited: February 2014.
[Android Source 2015]
Android Source: https://github.com/android/platform_frameworks_base. Visited: June 2015.
[Andrubis 2012]
Andrubis: *A Tool for Analyzing Unknown Android Applications*.
http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/. Visited: March 2014.
[Anubis 2012]
Anubis: *Malware Analysis for Unknown Binaries*. http://anubis.iseclab.org/.
Visited: May 2014.
[AOSP 2014]
AOSP: *Welcome to the Android Open Source Project!* http://source.android.com.
Visited: March 2014.
[Honeynet Project 2014]
Honeynet Project: *Android Reverse Engineering (A.R.E.) Virtual Machine available for download now!*.
www.honeynet.org/node/783. Visited: February 2014.
[Armando/Merlo/Verderame 2012]
Armando, A., Merlo, A., and L. Verderame: *Security issues in the Android Cross-layer Architecture*.
http://arxiv.org/pdf/1209.0687.pdf. Visited: July 2015.
[Bit9 2011]
Bit9: *The not-so-smartphones of 2011: Orphan Android Phones are Security Risk*.
http://www.bit9.com/files/Bit9_Orphan_Android_NotSo_Smartphones_2011.pdf.
Visited: February 2014.

[Bradley 2015]
Bradley, T.: *DroidDream becomes Android Market Nightmare*.
http://www.pcworld.com/businesscenter/article/221247/droiddream_becomes_android_market_nightmare.html. Visited: February 2015.
[Brähler 2010]
Brähler, S.: *Analysis of the Android Architecture*.
https://os.itec.kit.edu/downloads/sa_2010_braehler-stefan_android-architecture.pdf. Visited: July 2015.
[Brodeur 2013]
Brodeur, P.: *Zero-permission Android applications*.
http://leviathansecurity.com/blog/archives/17-Zero-Permission-Android-Applications.html. Visited: March 2015.
[Canalys 2013]
Canalys: *Over 1 billion Android-based smart phones to ship in 2017*.
http://www.canalys.com/newsroom/over-1-billion-android-based-smart-phones-ship-2017. Visited: 2013.
[Canvas 2015]
Canvas: *Owning Android*.
http://partners.immunityinc.com/movies/Lightning_Demo_Android.zip. Visited: June 2015.
[CarrierIQ 2013]
CarrierIQ: *Know your customer experience*.
http://www.carrieriq.com/know-your-customer-experience/. Visited: July 2013.
[Ccjernigan 2013]
Ccjernigan: *Security aw in power control widget opens protected settings*.
http://code.google.com/p/android/issues/detail?id=7890. Visited: March 2015.
[Contagio 2015]
Contagio. http://contagiodump.blogspot.com/.Visited: July 2015.
[Cluley 2012]
Cluley, G.: *Android Malware poses as angry birds space game*.
http://nakedsecurity.sophos.com/2012/04/12/android-malware-angry-birds-space-game/. Visited: June 2015.
[CuteCircuit 2015]
CuteCircuit: T–shirtos: *The future is getting closer*.
http://www.cutecircuit.com/tshirtos-the-future-is-getting-closer/. Visited: January 2015.
[Dandumont 2015]
Dandumont, P.: *Bouncer : La validation à posteriori sur l'Android Market*.
http://www.presence-pc.com/actualite/bouncer-google-46571/. Visited: September 2015.
[Degusta 2015]
Degusta, M.: *Android orphans: Visualizing a sad history of support*.
theunderstatement.com/post/11982112928/android-orphans-visualizing-a-sad-history-of-support. Visited: March 2015.

[Dex2jar 2015]
Dex2jar: *Tools to work with Android .dex and Java .class files*. http://code.google.com/p/dex2jar/. Visited: September 2014.
[Dietterich/Langley 2003]
Dietterich, T. and P. Langley: *Machine learning for cognitive networks: technology assessments and research challenges*, Draft of May 11, 2003.http://core.ac.uk/download/pdf/10195444.pdf, 2003. Visited: March 2015.
[Ericsson 2014]
Ericsson: *Ericsson Mobility Report Appendix in Sub Saharan Africa*.
http://www.ericsson.com/res/docs/2014/emr-june2014-regional-appendices-ssa.pdf. Visited: February 2015.
[Express Union 2014]
Express Union: *Express Union Mobile*. http://www.expressunion.net/en/eum1.php. Visited: November 2014.
[Felt/Wagner 2011]
Felt, A.P. and D. Wagner: *Phishing on Mobile Devices*. In: Proceedings of Workshop on Web 2.0 Security and Privacy (W2SP) 2011. http://w2spconf.com/2011/papers/felt-mobilephishing.pdf. Visited: July 2015.
[Fortinet 2013]
Fortinet: *FortiGuard Midyear Threat Report*.
http://www.fortinet.com/sites/default/files/whitepapers/FortiGuard-Midyear-Threat-Report-2013.pdf. Visited: August 2013.
[F-Secure 2013]
F-Secure: *Threat Description: Trojan:android/droidkungfu.c*.
http://www.f-secure.com/v-descs/trojan_android_droidkungfu_c.shtml. Visited: February 2013.
[F-Secure 2012]
F-Secure: *Mobile Threat Report Q1 2012*. Technical report, F–Secure.
http://www.f-secure.com/weblog/archives/MobileThreatReport_Q1_2012.pdf. Visited : April 2014.
[Fontaine 2015]
Fontaine, P.: *99sécurité majeure*. http://www.01net.com/rub/actualites/10005/actualites/securite/. Visited: January 2015.
[Fuchs/Chaudhuri/Foster 2005]
Fuchs, A.P., A. Chaudhuri, and J.S. Foster: *SCanDroid: Automated security certification of Android applications, 2005*. http://www.cs.umd.edu/~avik/projects/scandroidascaa. Visited: April 2015.
[Gartner 2011]
Gartner: *Gartner Says Worldwide Smartphone Sales Soared in Fourth Quarter of 2011 With 47 Percent Growth*.
http://www.gartner.com/it/page.jsp?id=1924314. Visited: March 2015.

[Gestwicki 1997]
Gestwicki, P.: *Id3: History, Implementation, and Applications*, 1997.
http://citeseer.ist.psu.edu/gestwicki97id.html. Visited: February 2015.
[Gomez/Neamtiu 2011]
Gomez, L. and I. Neamtiu: *A Characterization of Malicious Android Applications.*
http://www.lorenzobgomez.com/publications/MaliciousAppsTR.pdf, Visited: July 2015.
[Google 2014a]
Google: *Android Git Repositories*. https://android.googlesource.com/. Visited: February 2014.
[Google 2015]
Google: *Google Glass*. http://http://www.google.com/glass/. Visited: January 2015.
[Google Mobile Blog 2015]
Google Mobile Blog: *Android and Security*.
http://googlemobile.blogspot.com/2012/02/android-and-security.html. Visited: July 2015.
[Helfer/Lin 2012]
Helfer, J. and T. Lin.: *Giving the User Control over Android Permissions*.
http://css.csail.mit.edu/6.858/2012/projects/helfer-ty12.pdf. Visited: July 2015.
[IDA 2015]
IDA: *IDA About*. http://www.hex-rays.com/products/ida/index.shtml. Visited: January 2015.
[IDC 2013]
IDC:*Smartphone Uptake Gaining Pace in Africa as IDC Tips Shipments to Double Over Next Four Years*.
http://www.idc.com/getdoc.jsp?containerId=prAE24404513. Visited: December 2013.
[ITU 2013]
International Telecommunication Union (ITU): Percentage of Subscriptions per Country.
http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2013/Mobile_cellular_2000-2011.xls. Visited: September 2013.
[JD-GUI 2015]
JD-GUI: *Standalone Java Decompiler GUI.*
https://github.com/java-decompiler/jd-gui. Visited: January 2015.
[Jiang 2013]
Jiang, X.: *An Evaluation of the Application ("App") Verification Service in Android 4.2*.
http://www.cs.ncsu.edu/faculty/jiang/appverify. Visited: January 2015.
[Jeon et al. 2012]
Jeon, J., K.-K. Micinski, J.-A Vaughan, N. Reddy, Y. Zhu, J.-S. Foster, and T. Millstein: *Dr. Android and Mr. Hide: Fine-grained Security Policies on unmodified Android*.
http://drum.lib.umd.edu/bitstream/1903/12852/1/CS-TR-5006.pdf. Visited: July 2015.
[Juniper 2012a]
Juniper: *2011 Mobile Threats Report*.
http://www.juniper.net/us/en/local/pdf/additional-resources/jnpr-2011-mobile-threats-report.pdf. Visited: July 2015.

[Juniper 2012b]
Juniper: *Malicious mobile threats report 2010/2011: an objective briefing on the current mobile threat landscape based on Juniper networks global threat center research*.
http://direito.folha.uol.com.br/uploads/2/9/6/2/2962839/malicious_mobile_threats_report_2010-2011.pdf. Visited: July 2015.
[Kalige/Burkey 2012]
Kalige E. and D. Burkey: *A Case Study of Eurograbber: How 36 Million Euros was Stolen via Malware*. VERSAFE
Ltd and Check Point Software Technologies, December 2012.
http://www.cs.stevens.edu/~spock/Eurograbber_White_Paper.pdf. Visited: June 2015.
[Kindsight 2013]
Kindsight: *Security Labs Malware Report - Q2 2013*.
http://www.kindsight.net/sites/default/files/Kindsight-Q2-2013-Malware-Report.pdf. Visited: July 2013.
[Kravets 2011]
Kravets, D.: *Researcher's Video shows secret software on millions of phones logging everything*.
http://www.wired.com/threatlevel/2011/11/secret-software-logging-video/. Visited: July 2015.
[Lineberry/Richardson/Wyatt 2010]
Lineberry, A., D.L. Richarson, and T. Wyatt: *These aren't the permissions you're looking for*.
https://www.defcon.org/images/defcon-18/dc-18-presentations/Lineberry/DEFCON-18-Lineberry-Not-The-Permissions-You-Are-Looking-For.pdf. Visited: July 2015.
[Lineberry/Richardson/Wyatt 2010a]
Lineberry, A., D.L. Richarson, and T. Wyatt: *Circumventing Android Permissions.*
http://dtors.org/2010/08/06/circumventing-android-permissions/. Visited: July 2015.
[Llamas/Reith/Shirer 2013]
Llamas, R., R. Reith, and M. Shirer: *Apple Cedes Market Share in Smartphone Operating System Market as Android Surges and Windows Phone Gains, According to IDC*.
http://www.idc.com/getdoc.jsp?containerId=prUS24257413. Visited: August 2013.
[Lockheimer 2012]
Lockheimer, H.: Android and Security.
http://googlemobile.blogspot.nl/2012/02/android-and-security.html. Visited: February 2015.
[Lookout 2011]
Lookout: *Lookout Mobile Threat Report August 2011*.
www.mylookout.com/_downloads/lookout-mobile-threat-report-2011.pdf. Visited: August 2014.
[Lookout Update 2012]
Lookout Update: *Security alert: Hacked Websites serve suspicious Android apps (not compatible)*.
http://blog.mylookout.com/blog/2012/05/02/security-alert-hackedwebsites-serve-suspicious-android-apps-noncompatible/. Visited: April 2015.

[McAfee 2011]
McAfee Labs: *McAfee Threats Report: Second quarter 2011*.
www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2011.pdf. Visited March 2015.
[MTN 2014]
MTN: *MTN Mobile Money*. http://www.mtncameroon.net/LoadedPortal. Visited: November 2014.
[National Security Agency 2014]
National Security Agency: *SEAndroid Project Page*. http://selinuxproject.org/page/SEAndroid, Visited: July 2014.
[Newcomb 2015]
Newcomb, D.: *Weblink aims to bridge the nagging smartphone-cardisconnect*.
http://www.wired.com/autopia/2013/03/weblink-abalta-auto-apps. Visited: January 2015.
[Nielsen 2012]
Nielsen: *State of the Appnation – A Year of Change and Growth in U.S. Smartphones*.
http://www.nielsen.com/us/en/insights/news/2012/state-of-the-appnation-a-year-of-change-and-growth-in-u-s-smartphones.html. Visited: July 2015.
[Nielsen 2015]
Nielsen: *In U.S. Market, new smartphone buyers increasingly embracing Android*.
blog.nielsen.com/nielsenwire/online_mobile/. Visited: March 2015.
[NSCU 2014a]
Security Alert: *New DroidKungFu Variant – AGAIN! – Found in Alternative Android Markets*.
http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu3/. Visited: July 2014.
[NSCU 2014b]
Security Alert: *AnserverBot, New Sophisticated Android Bot Found in Alternative Android Markets*.
http://www.csc.ncsu.edu/faculty/jiang/AnserverBot/. Visited: July 2014.
[Oberheide/Miller 2012]
Oberheide J. and C. Miller: *Dissecting the Android Bouncer*. SummerCon 2012.
https://jon.oberheide.org/files/summercon12-bouncer.pdf. Visited: January 2015.
[Oberheide 2010]
Oberheide, J.: *Remote kill and install on Google Android*.
http://jon.oberheide.org/blog/2010/06/25/remote-kill-and-install-on-google-android/.
Visited: March 2014.
[OHA 2014a]
OHA: *Open Handset Alliance Releases Android SDK Press release*.
http://www.openhandsetalliance.com/press_111207.html. Visited: March 2014.
[OHA 2014b]
OHA: *Open Handset Alliance Website*. http://www.openhandsetalliance.com. Visited: 2014.
[OHA 2014c]
OHA: *Industry Leaders Announce Open Platform for Mobile Devices Press release*.
http://www.openhandsetalliance.com/press_110507.html. Visited: 2014.

[Orange 2015]
Orange: *Orange Money*. http://www.orange.cm/FR/article.php?aid=156. Visited: May 2015.
[Raphael 2010]
Raphael, J.: *Android 2.2 Upgrade List: Is your Phone getting Froyo? – Computerworld Blogs*.
http://blogs.computerworld.com/16310/android_22_upgrade_list. Visited: September 2014.
[Rubin 2012]
Rubin, A.: *Google+ post on the Android ecosystem*.
https://plus.google.com/u/0/112599748506977857728/posts/Btey7rJBaLF. Visited: May 2014.
[Russakovskii 2011]
Russakovskii, A.: *Massive Security Vulnerability in HTC Android Devices (EVO 3D, 4G, Thunderbolt, others) exposes phone numbers, GPS, SMS, Emails Addresses, much more*.
http://www.androidpolice.com/2011/10/01/massive-security-vulnerabilityin-htc-android-devices-evo-3d-4g-thunderbolt-others-exposes-phonenumbers-gps-sms-emails-addresses-much-more/. Visited: August 2014.
[Sabnani 2008]
Sabnani, S.V.: *Computer Security: A Machine Learning Approach*. Technical Report, RHUL-MA-2008-09, Department of Mathematics, Royal Holloway, University of London 2008.
https://www.ma.rhul.ac.uk/static/techrep/2008/RHUL-MA-2008-09.pdf. Visited: July 2015.
[Samsung 2015]
Samsung: *Samsung Smart TV*. http://www.samsung.com/us/2012-smart-tv/. Visited: January 2015.
[Santoku 2015]
Santoku: *Santoku 0.5 – Packaged & Delivered*. https://santoku-linux.com/. Visited: January 2015.
[Shabtai et al. 2009]
Shabtai, A., Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev: *Google Android: A state-of-the-art review of security mechanisms*. http://arxiv.org/ftp/arxiv/papers/0912/0912.5101.pdf. Visited: July 2015.
[Smali 2015]
Smali: *A Disassembler for Android's Dex Format*.http://code.google.com/p/smali/. Visited: January 2015.
[Smalley 2012a]
Smalley, S.: *SE Android release. SELinux Mailing List, Mailing List Archives*.
http://marc.info/?l=selinux&m=132588456202123&w=2. Visited: February 2014.
[Smalley 2012b]
Smalley, S.: *The Case for SE Android*.
http://selinuxproject.org/~jmorris/lss2011_slides/caseforseandroid.pdf. Visited: February 2014.

[SocialCompare 2015]
SocialCompare: Android versions comparison. http://socialcompare.com/en/comparisons. Visited: June 2015.
[Sony 2015]
Sony: Smartwatch. http://www.sonymobile.com/us/products/accessories/smartwatch/. Visited: January 2015.
[Statistica 2009]
Statistica: Global Market Share of Smartphone Operating Systems by quarter 2009-2012. http://www.statista.com/statistics/73662/quarterly-worldwide-smartphone-market-share-by-operating-system-since-2009/. Visited: April 2014.
[Statistica 2014a]
Statistica: Global Market Share held by Smartphone Operating Systems. http://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.
Visited: March 2014.
[Statistica 2014b]
Statistica: Global Smartphone Shipments Forecast. http://www.statista.com/statistics/263441/global-smartphone-shipments-forecast/.
Visited: March 2014.
[Svajcer 2015]
Svajcer, S.: *Sophos Mobile Security Threat Report*. http://www.sophos.com/enus/medialibrary/PDFs/other/sophos-mobile-security-threat-report.ashx. Visited: January 2015.
[Sverdlove 2011]
Sverdlove, H.: *The most vulnerable smartphones of 2011*. Bit9 report, November 2011. https://www.bit9.com/download/reports/Bit9Report_SmartPhones2011%282%29.pdf. Visited: July 2015.
[Symantec 2011]
Symantec: Android.rufraud technical details. http://www.symantec.com/security_response/writeup.jsp?docid=2011-121306-2304-99&tabid=2. Visited: June 2014.
[SysSec 2012]
SysSec: Deliverable D7.2: Intermediate Report on Cyberattacks on Ultra-portable Devices. Seventh Framework Programme – Information & Communication Technologies – Trustworthy ICT –Network of Excellence. http://www.syssec-project.eu/m/page-media/3/syssec-d7.3-CyberattacksLightweightDevices.pdf. Visited: July 2015.
[Travis Credit Union 2009]
Travis Credit Union: Phishing Scam targeting Android-based Mobile Devices. https://www.traviscu.org/news.aspx?blogmonth=12&blogyear=2009&blogid=112.
Visited: March 2014.
[Trend Micro 2014]
Trend Micro: TrendLabs 2Q 2013 Security Roundup.

http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/reports/rpt-2q-2013-trendlabs-security-roundup.pdf. Visited: July 2014.

[Trend Micro 2015]
Trend Micro: Trojanized apps root Android devices.
http://blog.trendmicro.com/trojanized-apps-root-android-devices/. Visited: July 2015.
[Valdés-Valdivieso/Penteriani/Lyons/Phillips 2012]
Valdés-Valdivieso, L., G. Penteriani, P. Lyons, and T. Phillips: *Sub-Saharan Africa Mobile Observatory 2012*.
http://www.gsma.com/publicpolicy/wp-content/uploads/2012/03/SSA_FullReport_v6.1_clean.pdf. Visited: July 2015.
[VirusTotal 2015]
VirusTotal: *VirusTotal - Free Online Virus, Malware and URL Scanner*. https://www.virustotal.com. Visited: January 2015.
[VisionMobile 2013]
VisionMobile: *Developer Economics Q3 2013, State of the Developer Nation*.
http://www.developereconomics.com/reports/q3-2013/#. Visited: December 2014.
[Waqas 2010]
Waqas, A.: *Root any Android Device and Samsung captivate with super one-click app*.
http://www.addictivetips.com/mobile/root-any-android-device-and-samsung-captivate-with-super-one-click-app/. Visited: December 2014.
[Woods 2010]
Woods, B.: *Researchers expose Android Webkit Browser Exploit*.
http://www.zdnet.co.uk/news/security-threats/2010/11/08/researchers-expose-android-webkit-browser-exploit-40090787/. Visited : December 2014.
[Zdnet 2013]
Zdnet: *Google intègre un système de détection des malwares à l'Android Market*.
http://www.zdnet.fr/actualites/google-integre-un-systeme-de-detection-des-malwares-a-l-android-market-39768195.htm. Visited : January 2015.

Image Sources

[Android Developers 2015]
Android Developers: *Android Architecture*.
http://developer.android.com/images/system-architecture.jpg. Visited: December 2015.

Terms and Acronyms

AAPT: Android Asset Packaging Tool.

ACC: Accuracy, a term referring to the sum of correct predictions divided by the total number of predictions respectively.

ADT: Android Development Tool.

ADB: Android Debug Bridge.

API: Application Programming Interface.

APK: Android Package.

ANTIC: Agence Nationale des Technologies de l'Information et de la Communication.

ARFF: Attribute-Relation File Format, referring to the format of data used in WEKA.

ARM: Architecture of RISC processors used for Android devices.

AUC: Area Under the Curve, used to characterise the performance of a classification model.

AVD: Android Virtual Device.

CA: Certificate Authority.

C&C: Command and Control.

DASAMoid: Detecting and Alerting System for Android Malware.

DEX: Dalvik Executable.

DDMS: Dalvik Debug Monitor Service.

DM: Discriminating Metrics.

DVM: Dalvik Virtual Machine.

ERR: Prediction Error, a term referring to the sum of all false predictions divided by the number of total predictions.

FP: False Positive.

FP: False Negative.

FPR: False Positive Rate.

FNR: False Negative Rate.

GPL: Google Protection Levels.

GPS: Global Positioning System. It is a space-based satellite navigation system that provides precise positional and velocity data and global time synchronization for air, sea, and land travel.

H-IDS: Host Intrusion Detection System.

HTTP: Hypertext Transfer Protocol.

ICT: Information and Telecommunication Technology.

IDE: Integrated Development Environment.

IDS: Intrusion Detection System.

I/O: Input/Output.

IoT: Internet of things.

IPS: Intrusion Prevention System.

JDK: Java Development Kit.

JVM: Java Virtual Machine.

K-IDS/IPS-K: Kernel IDS / IPS Kernel. Types of detection systems.

MD: Malevolent Developer.

MIPS: Microprocessor without Interlocked Pipeline Stage.

MMS: Multimedia Message Service.

NIDS: Network Intrusion Detection System.

OS: Operating System.

PID: Process Identifier.

PLP: Principle of Least Privilege.

PRE: Precision, a term referring to proportion of the examples, which truly have class X among all those, which were classified as class X.

REC: Recall, a term referring to TPR.

ROC: Receiver Operator Characteristics. They are useful graphs to select classification models based on their performance with respect to the False Positive and True Positive Rates.

SDK: Software Development Kit.

SEN: Sensitivity, a term referring to TPR.

SIM: Subscriber Identity Module.

SMS: Short Message Service.

SPC: Specificity, a term referring to TNR.

Su: Set User.

TLMDASA: Three-Layered Malware Detection and Alerting System for Android.

TN: True Negative.

TNR: True Negative Positive.

TOCTOU: Time of Check to Time of Use.

TP: True Positive.

TPR: True Positive Rate.

UI: User Interface.

UID: User Identifier.

URI: Uniform Resource Identifier.

URL: Uniform Resource Locator.

WEKA: Waikato Environment for Knowledge Analysis.

XML: eXtensible Markup.

Part I

Appendices

Appendix A

Permissions Weights

Permissions	Perm11	Perm22	GPL
ACCESS_ALL_EXTERNAL_STORAGE	0	0	2
ACCESS_CACHE_FILESYSTEM	9	9	3
ACCESS_CHECKIN_PROPERTIES	0	0	3
ACCESS_COARSE_LOCATION	2	2	1
ACCESS_CONTENT_PROVIDERS_EXTERNALLY	0	0	2
ACCESS_DOWNLOAD_MANAGER	0	4	
ACCESS_DOWNLOAD_MANAGER_ADVANCED	2	4	
ACCESS_DRM	9	9	
ACCESS_FINE_LOCATION	1	1	1
ACCESS_GPS	9	9	
ACCESS_LOCATION	2	5	0
ACCESS_LOCATION_EXTRA_COMMANDS	2	4	1
ACCESS_MOCK_LOCATION	0	0	1
ACCESS_MTP	0	0	3
ACCESS_NETWORK_STATE	0	1	0
ACCESS_NOTIFICATIONS	0	0	3
ACCESS_SURFACE_FLINGER	0	0	2
ACCESS_WIFI_STATE	2	7	0
ACCESS_WIMAX_STATE	9	9	0
ACCOUNT_MANAGER	0	0	2
ADD_SYSTEM_SERVICE	9	9	
ADD_VOICEMAIL	0	0	1
ALLOW_ANY_CODEC_FOR_PLAYBACK	0	0	3
ASEC_ACCESS	0	0	2
ASEC_CREATE	0	0	2
ASEC_DESTROY	0	0	2
ASEC_MOUNT_UNMOUNT	0	0	2
ASEC_RENAME	0	0	2

Permissions	Perm11	Perm22	GPL
AUTHENTICATE_ACCOUNTS	0	0	1
BACKUP	0	0	3
BATTERY_STATS	0	0	1
BILLING	0	0	
BIND_ACCESSIBILITY_SERVICE	0	0	2
BIND_APPWIDGET	0	0	3
BIND_DEVICE_ADMIN	0	0	2
BIND_DIRECTORY_SEARCH	0	0	3
BIND_INPUT_METHOD	0	0	2
BIND_KEYGUARD_APPWIDGET	0	0	3
BIND_NOTIFICATION_LISTENER_SERVICE	0	0	2
BIND_PACKAGE_VERIFIER	0	0	2
BIND_REMOTEVIEWS	0	0	3
BIND_TEXT_SERVICE	0	0	2
BIND_VPN_SERVICE	0	0	2
BIND_WALLPAPER	0	0	3
BLUETOOTH	0	0	1
BLUETOOTH_ADMIN	0	0	2
BLUETOOTH_STACK	0	0	1
BRICK	5	0	2
BROADCAST_PACKAGE_REMOVED	1	3	2
BROADCAST_SMS	0	0	2
BROADCAST_STICKY	0	0	0
BROADCAST_WAP_PUSH	1	0	2
C2D_MESSAGE	0	0	2
CALL_PHONE	1	3	1
CALL_PRIVILEGED	0	0	3
CAMERA	0	0	3
CAMERA_DISABLE_TRANSMIT_LED	0	0	1
CHANGE_BACKGROUND_DATA_SETTING	0	0	2
CHANGE_COMPONENT_ENABLED_STATE	0	0	3
CHANGE_CONFIGURATION	1	4	3
CHANGE_NETWORK_STATE	0	4	0
CHANGE_WIFI_MULTICAST_STATE	0	0	1
CHANGE_WIFI_STATE	1	4	1
CHANGE_WIMAX_STATE	9	9	1
CHECK_LICENSE	0	1	
CLEAR_APP_CACHE	0	0	1
CLEAR_APP_USER_DATA	0	0	2
CONFIGURE_WIFI_DISPLAY	0	0	2

Permissions	Perm11	Perm22	GPL
CONFIRM_FULL_BACKUP	0	0	2
CONNECTIVITY_INTERNAL	0	0	3
CONTROL_LOCATION_UPDATES	3	0	3
CONTROL_WIFI_DISPLAY	0	0	2
COPY_PROTECTED_DATA	0	0	2
CRYPT_KEEPER	0	0	3
DELETE_CACHE_FILES	4	4	3
DELETE_PACKAGES	4	5	3
DEVICE_POWER	1	3	2
DIAGNOSTIC	0	0	2
DISABLE_KEYGUARD	2	6	1
DUMP	0	0	1
EXPAND_STATUS_BAR	1	3	0
FACTORY_TEST	5	0	2
FILTER_EVENTS	0	0	2
FLASHLIGHT	0	0	0
FORCE_BACK	0	0	2
FORCE_STOP_PACKAGES	0	0	2
FREEZE_SCREEN	0	0	2
GET_ACCOUNTS	0	0	0
GET_APP_OPS_STATS	0	0	3
GET_DETAILED_TASKS	0	0	2
GET_PACKAGE_SIZE	1	3	0
GET_TASKS	0	3	1
GET_TOP_ACTIVITY_INFO	0	0	2
GLOBAL_SEARCH	9	9	2
GLOBAL_SEARCH_CONTROL	9	9	3
GRANT_REVOKE_PERMISSIONS	0	0	2
HARDWARE_TEST	1	1	2
INJECT_EVENTS	0	0	2
INSTALL_DRM	9	9	
INSTALL_LOCATION_PROVIDER	0	0	3
INSTALL_PACKAGES	4	5	3
INSTALL_SHORTCUT	0	3	
INTERACT_ACROSS_USERS	0	0	2
INTERACT_ACROSS_USERS_FULL	0	0	3
INTERNAL_SYSTEM_WINDOW	0	0	2
INTERNET	1	1	1
KILL_BACKGROUND_PROCESSES	0	0	0
LOCATION_HARDWARE	0	0	3

Permissions	Perm11	Perm22	GPL
LOOP_RADIO	0	0	0
MAGNIFY_DISPLAY	0	0	2
MANAGE_ACCOUNTS	0	0	1
MANAGE_APP_TOKENS	0	0	2
MANAGE_NETWORK_POLICY	0	0	3
MANAGE_USB	0	0	3
MANAGE_USERS	0	0	3
MASTER_CLEAR	0	0	3
MODIFY_APPWIDGET_BIND_PERMISSIONS	0	0	3
MODIFY_AUDIO_SETTINGS	0	0	0
MODIFY_NETWORK_ACCOUNTING	0	0	3
MODIFY_PHONE_STATE	0	4	3
MOUNT_FORMAT_FILESYSTEMS	0	0	2
MOUNT_UNMOUNT_FILESYSTEMS	0	4	2
MOVE_PACKAGE	0	0	3
NET_ADMIN	0	0	2
NET_TUNNELING	0	0	2
NFC	0	0	1
PACKAGE_USAGE_STATS	0	0	3
PACKAGE_VERIFICATION_AGENT	0	0	3
PERFORM_CDMA_PROVISIONING	0	0	3
PERSISTENT_ACTIVITY	0	0	0
PROCESS_OUTGOING_CALLS	1	6	0
READ_CALENDAR	0	0	1
READ_CALL_LOG	0	0	1
READ_CELL_BROADCASTS	0	0	1
READ_CONTACTS	2	4	1
READ_DREAM_STATE	0	0	2
READ_EXTERNAL_STORAGE	0	4	0
READ_FRAME_BUFFER	2	0	3
READ_GMAIL	0	0	
READ_HISTORY_BOOKMARKS	2	4	1
READ_INPUT_STATE	0	0	2
READ_LOGS	1	4	3
READ_NETWORK_USAGE_HISTORY	0	0	3
READ_OWNER_DATA	0	0	
READ_PHONE_STATE	8	7	1
READ_PRIVILEGED_PHONE_STATE	0	0	3
READ_PROFILE	0	0	1
READ_SECURE_SETTINGS	0	0	

Permissions	Perm11	Perm22	GPL
READ_SETTINGS	0	4	
READ_SMS	6	8	1
READ_SOCIAL_STREAM	0	0	1
READ_SYNC_SETTINGS	0	0	0
READ_SYNC_STATS	0	0	0
READ_USER_DICTIONARY	0	0	1
REBOOT	0	0	3
RECEIVE_BOOT_COMPLETED	5	6	0
RECEIVE_DATA_ACTIVITY_CHANGE	0	0	3
RECEIVE_EMERGENCY_BROADCAST	0	0	3
RECEIVE_MMS	3	5	1
RECEIVE_SMS	6	5	1
RECEIVE_WAP_PUSH	4	5	1
RECORD_AUDIO	0	1	1
REMOTE_AUDIO_PLAYBACK	0	0	2
REMOVE_TASKS	0	0	2
REORDER_TASKS	0	0	3
RESTART_PACKAGES	3	4	0
RETRIEVE_WINDOW_CONTENT	0	0	2
RETRIEVE_WINDOW_INFO	0	0	3
SEND_DOWNLOAD_COMPLETED_INTENTS	1	2	
SEND_RESPOND_VIA_MESSAGE	0	0	3
SEND_SMS	8	5	1
SERIAL_PORT	0	0	3
SET_ACTIVITY_WATCHER	0	0	2
SET_ALARM	0	0	0
SET_ALWAYS_FINISH	0	0	3
SET_ANIMATION_SCALE	0	0	3
SET_DEBUG_APP	0	0	3
SET_KEYBOARD_LAYOUT	0	0	2
SET_ORIENTATION	0	0	2
SET_POINTER_SPEED	0	0	2
SET_PREFERRED_APPLICATIONS	1	2	2
SET_PROCESS_LIMIT	0	0	3
SET_SCREEN_COMPATIBILITY	0	0	2
SET_TIME	0	0	0
SET_TIME_ZONE	0	0	3
SET_WALLPAPER	2	3	3
SET_WALLPAPER_COMPONENT	0	0	0
SET_WALLPAPER_HINTS	1	3	0

Permissions	Perm11	Perm22	GPL
SHUTDOWN	0	0	3
SIGNAL_PERSISTENT_PROCESSES	0	0	3
START_ANY_ACTIVITY	0	0	2
STATUS_BAR	1	2	2
STATUS_BAR_SERVICE	0	0	0
STOP_APP_SWITCHES	0	0	3
SUBSCRIBED_FEEDS_READ	0	0	0
SUBSCRIBED_FEEDS_WRITE	0	0	1
SYSTEM_ALERT_WINDOW	0	0	1
TEMPORARY_ENABLE_ACCESSIBILITY	0	0	2
UPDATE_APP_OPS_STATS	0	0	3
UPDATE_DEVICE_STATS	2	0	3
UPDATE_LOCK	0	0	3
USE_CREDENTIALS	0	0	1
USE_SIP	0	0	1
VIBRATE	0	2	0
WAKE_LOCK	0	2	0
WRITE_APN_SETTINGS	4	5	3
WRITE_CALENDAR	0	0	1
WRITE_CALL_LOG	0	0	1
WRITE_CONTACTS	2	5	1
WRITE_DREAM_STATE	0	0	2
WRITE_EXTERNAL_STORAGE	1	4	1
WRITE_GSERVICES	0	0	3
WRITE_HISTORY_BOOKMARKS	2	5	1
WRITE_MEDIA_STORAGE	0	0	3
WRITE_OWNER_DATA	1	3	
WRITE_PROFILE	0	0	1
WRITE_SECURE_SETTINGS	2	0	3
WRITE_SETTINGS	0	2	0
WRITE_SMS	4	6	1
WRITE_SOCIAL_STREAM	0	0	1
WRITE_SYNC_SETTINGS	0	0	0
WRITE_USER_DICTIONARY	0	0	0

Table A.1: Permissions with GPL and DM

Permissions	Perm11	Perm22	GPL
ACCESS_WIMAX_STATE	9	9	0
RECEIVE_BOOT_COMPLETED	5	6	0

Permissions	Perm11	Perm22	GPL
RESTART_PACKAGES	3	4	0
ACCESS_LOCATION	2	5	0
ACCESS_WIFI_STATE	2	7	0
EXPAND_STATUS_BAR	1	3	0
GET_PACKAGE_SIZE	1	3	0
PROCESS_OUTGOING_CALLS	1	6	0
SET_WALLPAPER_HINTS	1	3	0
ACCESS_NETWORK_STATE	0	1	0
BROADCAST_STICKY	0	0	0
CHANGE_NETWORK_STATE	0	4	0
FLASHLIGHT	0	0	0
GET_ACCOUNTS	0	0	0
KILL_BACKGROUND_PROCESSES	0	0	0
LOOP_RADIO	0	0	0
MODIFY_AUDIO_SETTINGS	0	0	0
PERSISTENT_ACTIVITY	0	0	0
READ_EXTERNAL_STORAGE	0	4	0
READ_SYNC_SETTINGS	0	0	0
READ_SYNC_STATS	0	0	0
SET_ALARM	0	0	0
SET_TIME	0	0	0
SET_WALLPAPER_COMPONENT	0	0	0
STATUS_BAR_SERVICE	0	0	0
SUBSCRIBED_FEEDS_READ	0	0	0
VIBRATE	0	2	0
WAKE_LOCK	0	2	0
WRITE_SETTINGS	0	2	0
WRITE_SYNC_SETTINGS	0	0	0
WRITE_USER_DICTIONARY	0	0	0

Table A.2: Permissions of GPL 0

Appendix B

F-SECURE Classification

Many privacy issues (6)	AVG
	McAfee Security
	Android system ✓
	Dendroid ✓
	FlvPlayer✓
	SPL Meter Free ✓
Some privacy issues (8)	41CA3EFDE1FB6228A3EA13DB67BD0722_Trahanie_ofisnyih_rabot.apk ✓
	ING Bank N.V ✓
	Jelly Matching ✓
	Mobile Security ✓
	Safety Care
	Sound Hound ✓
	Who's calling
	Fsecure Mobile Security
Few Privacy issues (16)	AlfSafe ✓
	Avast Mobile Security
	BaseApp ✓
	Battery Doctor ✓
	Black Market Alpha ✓
	com.Android.tools.system ✓
	CSipSimple
	German
	o5Android ✓
	PronEnabler ✓
	Radardroid Pro ✓
	SberSafe ✓
	System Service ✓
	Talking Ben
	VkSafe ✓

	ThreatJapan_D09A1FF8A96A6633B3B285F530E2D430_News
	Androidnocode.apk ✓
No privacy issues (33)	100 % applivre
	Alphabets & Numbers Writing
	Apk Extractor
	AppPermissionWatcher
	AppPermissions
	AweSome Jokes ✓
	Baby Ninja Dance
	Badoink ✓
	Battery Improve ✓
	Business Calendar Pro ✓
	Candy Crush Saga
	Chibi Fighter ✓
	DASAMoid
	Détecteur de Carrier IQ ✓
	File Manager
	Important Dates
	Install ✓
	Kids Songs
	Learn Numbers in French Lang
	Malware Tracker
	Malware Tracker
	My Permissions
	Noms Abc
	Permission Friendly Apps
	Permission Monitor Free
	Polaris Viewer 4
	Pregnancy Tracker
	Screenshot Easy
	Se-Cure Mobile AV ✓
	Smartworld
	Test
	sb.apk ✓
✓indicates a malware sample	

Table B.1: Testing dataset

Appendix C

Resources Associated to Risks

AAPT: Android Asset Packaging Tool.

N°	Resources	Cij	Combinations	DM	R_1	R_2	R_3	R_4	W
1	MESSAGES	C1,1	SEND_SMS	8	1	0	1	1	3
		C2,1	RECEIVE_SMS	6	1	0	0	0	1
		C3,1	RECEIVE_MMS	3	1	0	0	0	1
		C4,1	READ_SMS	6	1	0	0	0	1
		C5,1	WRITE_SMS	4	0	1	0	0	1
		C6,1	SEND_SMS RECEIVE_SMS		1	0	1	1	3
		C7,1	SEND_SMS READ_SMS		1	0	1	1	3
		C8,1	SEND_SMS WRITE_SMS		0	1	1	1	3
		C9,1	RECEIVE_SMS READ_SMS		1	0	0	0	1
		C10,1	RECEIVE_SMS WRITE_SMS		1	1	0	0	2
		C11,1	READ_SMS WRITE_SMS		1	1	0	1	3
		C12,1	SEND_SMS RECEIVE_SMS READ_SMS		1	0	1	1	3
		C13,1	SEND_SMS RECEIVE_SMS WRITE_SMS		1	1	1	1	4
		C14,1	SEND_SMS READ_SMS WRITE_SMS		1	1	1	1	4
		C15,1	WRITE_SMS READ_SMS RECEIVE_SMS		1	1	0	0	2

		C16,1	READ_SMS SEND_SMS RECEIVE_SMS WRITE_SMS		1	1	1	1	4
2	CONTACTS	C1,2	READ_CONTACTS	2	1	0	0	0	1
		C2,2	WRITE_CONTACTS	2	0	1	0	0	1
		C3,2	MANAGE_ACCOUNTS	0	1	1	0	0	2
		C4,2	READ_CONTACTS WRITE_CONTACTS		1	1	0	0	2
		C5,2	READ_CONTACTS MANAGE_ACCOUNTS		1	1	0	0	2
		C6,2	MANAGE_ACCOUNTS WRITE_CONTACTS		1	1	0	0	2
		C7,2	READ_CONTACTS WRITE_CONTACTS MANAGE_ACCOUNTS		1	1	1	0	3
3	CALLS	C1,3	PROCESS_OUTGOING_CALLS	1	1	0	1	1	3
		C2,3	READ_CALL_LOG	0	1	0	0	0	1
		C3,3	WRITE_CALL_LOG	0	0	1	1	0	2
		C4,3	CALL_PHONE	1	1	0	1	1	3
		C5,3	CALL_PRIVILEGED	0	1	0	1	1	3
		C6,3	PROCESS_OUTGOING_CALLS READ_CALL_LOG		1	0	1	1	3
		C7,3	PROCESS_OUTGOING_CALLS WRITE_CALL_LOG		1	1	1	1	4
		C8,3	PROCESS_OUTGOING_CALLS CALL_PHONE		1	0	1	1	3
		C9,3	PROCESS_OUTGOING_CALLS CALL_PRIVILEGED		1	0	1	1	3
		C10,3	READ_CALL_LOG WRITE_CALL_LOG		1	1	0	0	2
		C11,3	READ_CALL_LOG CALL_PHONE		1	0	1	1	3
		C12,3	READ_CALL_LOG CALL_PRIVILEGED		1	0	1	1	3
		C13,3	WRITE_CALL_LOG CALL_PHONE		1	1	1	1	3
		C14,3	WRITE_CALL_LOG CALL_PRIVILEGED		1	1	1	1	3
		C15,3	CALL_PHONE CALL_PRIVILEGED		1	0	1	1	3

		C16,3	PROCESS_OUTGOING_CALLS, READ_CALL_LOG WRITE_CALL_LOG		1	1	1	1	4
		C17,3	PROCESS_OUTGOING_CALLS, READ_CALL_LOG CALL_PHONE		1	0	1	1	3
		C18,3	PROCESS_OUTGOING_CALLS, READ_CALL_LOG, CALL_PRIVILEGED		1	0	1	1	3
		C19,3	PROCESS_OUTGOING_CALLS, WRITE_CALL_LOG, CALL_PHONE		1	1	1	1	4
		C20,3	PROCESS_OUTGOING_CALLS, WRITE_CALL_LOG, CALL_PRIVILEGED		1	1	1	1	4
		C21,3	PROCESS_OUTGOING_CALLS, CALL_PHONE, WRITE_CALL_LOG		1	1	1	1	4
		C22,3	PROCESS_OUTGOING_CALLS, CALL_PHONE, CALL_PRIVILEGED		1	0	1	1	3
		C23,3	READ_CALL_LOG, WRITE_CALL_LOG, CALL_PHONE		1	1	1	1	4
		C24,3	READ_CALL_LOG, WRITE_CALL_LOG, CALL_PRIVILEGED		1	1	1	1	4
		C25,3	READ_CALL_LOG, CALL_PHONE, CALL_PRIVILEGED		1	0	1	1	3
		C26,3	WRITE_CALL_LOG, CALL_PHONE, CALL_PRIVILEGED		1	1	1	1	4
		C27,3	PROCESS_OUTGOING_CALLS, READ_CALL_LOG, WRITE_CALL_LOG, CALL_PHONE		1	1	1	1	4
		C28,3	PROCESS_OUTGOING_CALLS, READ_CALL_LOG, WRITE_CALL_LOG, CALL_PRIVILEGED		1	1	1	1	4
		C29,3	PROCESS_OUTGOING_CALLS, READ_CALL_LOG, CALL_PHONE CALL_PRIVILEGED		1	0	1	1	4
		C30,3	PROCESS_OUTGOING_CALLS, WRITE_CALL_LOG, CALL_PHONE, CALL_PRIVILEGED		0	1	1	1	3

		C31,3	READ_CALL_LOG, WRITE_CALL_LOG, CALL_PHONE, CALL_PRIVILEGED		1	1	1	1	4
		C32,3	PROCESS_OUTGOING_CALLS, READ_CALL_LOG, WRITE_CALL_LOG, CALL_PHONE, CALL_PRIVILEGED		1	1	1	1	4
4	CALENDAR	C1,4	READ_CALENDAR	0	1	0	0	0	1
		C2,4	WRITE_CALENDAR	0	0	1	1	0	2
		C3,4	READ_CALENDAR WRITE_CALENDAR		1	1	1	0	3
5	LOCATION	C1,5	ACCESS_FINE_LOCATION	1	1	0	1	0	2
		C2,5	ACCESS_COARSE_ LOCATION	2	1	0	1	0	2
		C3,5	INSTALL_LOCATION_ PROVIDER	0	1	0	1	0	2
		C4,5	LOCATION_HARDWARE	0	1	0	1	0	2
		C5,5	ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION		1	0	0	0	1
		C6,5	ACCESS_FINE_LOCATION, INSTALL_LOCATION_PROVIDER		1	0	0	0	2
		C7,5	ACCESS_FINE_LOCATION, LOCATION_HARDWARE		1	0	0	0	1
		C8,5	ACCESS_COARSE_ LOCATION, INSTALL_LOCATION_ PROVIDER		1	0	0	0	1
		C9,5	ACCESS_COARSE_ LOCATION, LOCATION_HARDWARE		1	0	0	0	1
		C10,5	INSTALL_LOCATION_ PROVIDER, LOCATION_HARDWARE		1	0	0	0	1
		C11,5	ACCESS_FINE_LOCATIO ACCESS_COARSE_LOCATION, INSTALL_LOCATION_PROVIDER		1	0	0	0	1
		C12,5	ACCESS_FINE_LOCATION ACCESS_COARSE_ LOCATION, LOCATION_HARDWARE		1	0	0	0	1
		C13,5	ACCESS_FINE_LOCATION INSTALL_LOCATION_PROVIDER, LOCATION_HARDWARE		1	0	0	0	1
		C14,5	ACCESS_COARSE_ LOCATION, INSTALL_LOCATION_ PROVIDER, LOCATION_HARDWARE		1	0	0	0	1

		C15,5	ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, INSTALL_LOCATION_PROVIDER, LOCATION_HARDWARE		1	0	0	0	1
6	WIFI	C1,6	ACCESS_WIFI_STATE	2	1	0	0	0	1
		C2,6	CHANGE_WIFI_STATE	1	0	1	0	0	1
		C3,6	CHANGE_WIFI_MULTICAST_STATE	0	0	1	0	0	1
		C4,6	ACCESS_WIFI_STATE, CHANGE_WIFI_STATE		1	1	0	0	2
		C5,6	ACCESS_WIFI_STATE, CHANGE_WIFI_MULTICAST_STATE		1	1	0	0	2
		C6,6	CHANGE_WIFI_STATE CHANGE_WIFI_MULTICAST_STATE		0	1	0	0	1
		C7,6	ACCESS_WIFI_STATE, CHANGE_WIFI_STATE CHANGE_WIFI_MULTICAST_STATE		1	1	0	0	2
7	BLUETOOTH	C1,7	BLUETOOTH	0	1	1	1	0	3
		C2,7	BLUETOOTH_ADMIN	0	1	1	1	0	3
		C3,7	BLUETOOTH, BLUETOOTH_ADMIN		1	1	1	0	3
8	NETWORK	C1,8	INTERNET	1	1	0	1	0	2
		C2,8	ACCESS_NETWORK_STATE	0	1	0	0	0	1
		C3,8	CHANGE_NETWORK_STATE	0	0	1	0	0	1
		C4,8	INTERNET, ACCESS_NETWORK_STATE		1	1	1	0	3
		C5,8	INTERNET, CHANGE_NETWORK_STATE		1	1	1	0	3
		C6,8	ACCESS_NETWORK_ STATE, CHANGE_NETWORK_ STATE		1	1	0	0	2
		C7,8	INTERNET, ACCESS_NETWORK_STATE, CHANGE_NETWORK_ STATE		1	1	1	0	3
9	TELEPHONY	C1,9	MODIFY_PHONE_STATE	0	0	1	0	0	1
		C2,9	READ_PHONE_STATE	8	1	0	0	0	1
		C3,9	READ_PHONE_STATE, MOD- IFY_PHONE_STATE		1	1	0	0	2
10	WEBTRACE	C1,10	WRITE_HISTORY_ BOOKMARKS	2	0	1	0	0	1
		C2,10	READ_HISTORY_ BOOKMARKS	2	1	0	0	0	1

		C3,10	READ_HISTORY_ BOOKMARKS, WRITE_HISTORY_ BOOKMARKS		1	1	0	0	2

Table C.1: Resources and risks generated

Appendix D

Scripts for Analysis

```
1  #!/bin/bash
2  chemin=santoku
3  for i in $(ls Cop/)
4  do
5      filename=$i
6      unzip Cop/$i -d Un/$filename
7      java -jar $chemin/axmlprinter2/AXMLPrinter2.jar Un/$filename/
           AndroidManifest.xml > Mani/$filename.txt
8  done
```

Listing D.1: Decompiling packages

```
1  #!/bin/bash
2  for app in $(ls PERS1/)
3  do
4  filename=${app%.*}
5  echo  PERS/$filename.per
6  while read perm
7  do
8  r=$(grep -c "$perm" PERS1/$app)
9  if [ $r -gt 0 ]; then
10 echo $perm  PERS/$filename.per
11 fi
12 done < permissions
13 done
```

Listing D.2: Removing duplicates

```
1 #!/bin/bash
2 for app in $(ls Mani/)
3 do
4 filename=${app%.*}
5 grep -R 'uses-permission
6 android:name="android.permission.' Mani/$filename.txt > PERS1/
    $filename.perm
7 done
```

Listing D.3: Extracting formatted permissions

```
#!/bin/bash
echo "app,zero,un,deux,trois,quatre,cinq,six,sept,huit,neuf,state">NOR1.csv
while read app
do
neuf=0
huit=0
sept=0
six=0
cinq=0
quatre=0
trois=0
deux=0
un=0
zero=0
while read per
do
r=$(grep -wr $per pacNew.csv | cut -d "," -f 2)
#9
if [ $r -eq 9 ]; then
neuf=$(($neuf+1))
fi
#8
if [ $r -eq 8 ]; then
huit=$(($huit+1))
fi
#7
if [ $r -eq 7 ]; then
sept=$(($sept+1))
fi
#6
if [ $r -eq 6 ]; then
six=$(($six+1))
fi
#5
if [ $r -eq 5 ]; then
cinq=$(($cinq+1))
fi
#4
if [ $r -eq 4 ]; then
quatre=$(($quatre+1))
fi
#3
if [ $r -eq 3 ]; then
trois=$(($trois+1))
fi
#2
if [ $r -eq 2 ]; then
deux=$(($deux+1))
fi
#1
if [ $r -eq 1 ]; then
un=$(($un+1))
fi
#0
if [ $r -eq 0 ]; then
zero=$(($zero+1))
fi
done < NORMAUX/$app
echo "$app,$zero,$un,$deux,$trois,$quatre,$cinq,$six,$sept,$huit,$neuf,normale">>NOR1.csv
done < NORMAUX/f
```

Listing D.4: Script to determine vectors of normal applications for the model 1

```
#!/bin/bash
while read per
do
n=0
p=0
for app in $(ls MALWARES/)
do
c=$(grep -c $per MALWARES/$app)
if [ $c -gt 0 ];then
n=$(($n+1))
fi
done
for ap in $(ls NORMAUX/)
do
t=$(grep -c $per NORMAUX/$ap)
if [ $t -gt 0 ];then
p=$(($p+1))
fi
done
echo $per,$n,$p  STATS.csv
done < permissions
```

Listing D.5: Determining frequency counts

```
#!/bin/bash
for app in $(ls DREBIN/)
do
#les initialisations
un=0
deux=0
trois=0
quatre=0
cinq=0
six=0
sept=0
vrai1=0
vrai2=0
vrai3=0
vrai4=0
vrai5=0
vrai6=0
vrai7=0
#1
while read line
do
per=$(grep -wr "$line" telephony/1 | cut -d ',' -f 1)
nbr=$(grep -wr "$line" telephony/1 | cut -d ',' -f 2)
r=$(grep -c $per DREBIN/$app)
if [ $r -eq 0 ]; then
vrai1=1
else
un=$(($un+$nbr))
fi
done < telephony/1
#2
while read line
do
per=$(grep -wr "$line" telephony/2 | cut -d ',' -f 1)
nbr=$(grep -wr "$line" telephony/2 | cut -d ',' -f 2)
r=$(grep -c $per DREBIN/$app)
if [ $r -eq 0 ]; then
vrai2=1
else
deux=$(($deux+$nbr))
fi
done < telephony/2
#3
while read line
do
per=$(grep -wr "$line" telephony/3 | cut -d ',' -f 1)
nbr=$(grep -wr "$line" telephony/3 | cut -d ',' -f 2)
r=$(grep -c $per DREBIN/$app)
if [ $r -eq 0 ]; then
vrai3=1
else
trois=$(($trois+$nbr))
fi
done < telephony/3
# the last conditions
if [ $vrai1 -eq 1 ]; then
un=0
fi
if [ $vrai2 -eq 1 ]; then
deux=0
fi
if [ $vrai3 -eq 1 ]; then
trois=0
fi
echo $app,$un,$deux,$trois  telephonyDB.csv
done
```

Listing D.6: Determining risks for the telephony resources

```
#!/bin/bash
while read app
do
#les initialisations
un=0
deux=0
trois=0
vrai1=0
vrai2=0
vrai3=0
#1
while read line
do
per=$(grep -wr "$line" internet/1 | cut -d ',' -f 1)
nbr=$(grep -wr "$line" internet/1 | cut -d ',' -f 2)
r=$(grep -c $per Normaux/$app)
if [ $r -eq 0 ]; then
vrai1=1
else
un=$(($un+$nbr))
fi
done < internet/1
# The last conditions
if [ $vrai1 -eq 1 ]; then
un=0
fi
echo $app,$un  internetN.csv
done < Normaux/f
```

Listing D.7: Determining risks for the network category