A framework for detection of malicious software in Android handheld systems using Machine Learning techniques.

Blas Torregrosa

Universitat Oberta de Catalunya

Master Program in Security of Information and Communication Technologies (MISTIC)

Valladolid / Barcelona 2015

A framework for detection of malicious software in Android handheld systems using Machine Learning techniques

Master Program in Security of Information and Communication Technologies (MISTIC)

Author: Blas Torregrosa García
Consultor: Carles Estorach Espinós

Acknowledgements

I would like to express my warm thanks to Dr. Andrew Ng, Associate Professor at Stanford University, for his inspiring lectures on Machine Learning published on Coursera.

I cannot find the words to express my gratitude towards Dr. Luis Floria for his tireless encouragement and assistance.

My sincere thanks to Dr. Pablo de la Fuente for leading my work on diverse exciting projects.

I'd like to seize this opportunity to thank all my friends at Coffee's House for providing very friendly environment and unforgettable coffee breaks: Miguel, Jesús, Jorge, José Luis and Ceci.

Last but not least, I'd like to thank my mates at ATDP C.G. for sharing their time with me and making me feel as a part of the group: Fer, Albert, Raúl, Jonathan, Rodri, Nacho, Ana, Esther, David, Carlos and the newbies Víctor and Álvaro.

Blas Torregrosa, 2015/01/02

Abstract

Nowadays the Android platform is the fastest growing handheld operating system. As such, it has become the most coveted and viable target of malicious applications.

The present study aims at designing and developing new approaches to detect malicious applications in Android-based devices. More precisely, **MaLDroide** (**Ma**chine **L**earning–based **D**etector for And**roid** malwar**e**), a framework for detection of Android malware based on Machine Learning techniques, is introduced here. It is devised to identify malicious applications.

To start with, features of real–world known-benign and known–malicious applications are extracted, gathering 299 features grouped into 8 categories: stats, requested permissions, used permission, API calls, Intents, risks, system calls, and content access. MaLDroide uses *static analysis* in order to automatically extract these features.

As an essential part of the work, a mechanism for *dimensionality reduction*, using Principal Component Analysis (PCA), was applied, the original 299 features are reduced to 27, retaining 90% of the variance, and dramatically reducing time and space consumption.

Once determined the appropriate data set, a *training process* of the seven most widely used Machine Learning algorithms (Naïve Bayes, Decision Tree, Random Forest, k-Nearest Neighbor, Support Vector Machine, Multi-Layer Perceptron, and AdaBoost) is undertaken.

After an evaluation process with 436 normal applications and 2295 malware samples, and on the basis of the evaluation results, one can conclude that *Support Vector Machine classifier produces the most accurate predictions, and identifies malware with an accuracy of 99.8%*.

The evaluation also shows that our framework constitutes a valuable tool which is effective in detecting malware for Android devices.

Keywords: Mobile Malware, Android, Android Malware Detection, Static Analysis, Security, Machine Learning, Dimensionality Reduction.

Resumen

En la actualidad la plataforma Android es el sistema operativo de mayor crecimiento entre los dispositivos móviles. Por ello, también se ha convertido en el objetivo más codiciado y viable para las aplicaciones maliciosas.

Este trabajo tiene como objetivo el diseño y el desarrollo de nuevas formas de detección de estas aplicaciones maliciosas en los dispositivos basados en Android. Más concretamente, en este trabajo se presenta **MaLDroide** (**Ma**chine **L**earning–based **D**etector for And**roid** malwar**e**), un marco de trabajo para la detección de malware en Android utilizando técnicas de Aprendizaje Automatizado.

Para empezar, se extraen una serie de atributos de aplicaciones recientes y reales, tanto legítimas como malware. Exactamente se recogen 299 atributos agrupados en 8 categorías: estadísticas, permisos solicitados, permisos utilizados, llamadas a la API, Intents, riesgos, llamadas al sistema y acceso a contenidos. MaLDroide utiliza *análisis estático* para extraer estos atributos de forma automatizada.

Como parte esencial del trabajo, se ha utilizado un mecanismo que permite *reducir la dimensión* del problema utilizando Análisis de Componentes Principales (PCA, por sus siglas en inglés), pasando de las 299 iniciales a las 27 finales, manteniendo un 90 % de la varianza y reduciendo espectacularmente el tiempo y el espacio necesarios.

Una vez determinado el conjunto de datos adecuado, se procede al *adiestramiento de los clasificadores* mediante los siete algoritmos de Aprendizaje Automatizado más ampliamente utilizados: Naïve Bayes, Decision Tree, Random Forest, k-Nearest Neighbor, Support Vector Machine, Multi-Layer Perceptron y AdaBoost.

Finalmente, se realiza una evaluación usando 436 aplicaciones legítimas y 2296 casos de malware. Y a la vista de los resultados obtenidos, se puede concluir que *el clasificador Support Vector Machine obtiene más predicciones correctas, con una precisión del 99,8 %*.

A la vista de estos resultados queda patente que el marco de trabajo presentado es una herramienta muy eficaz para detectar malware en los dispositivos con Android.

Palabras clave: Malware en móviles, Android, Detección de malware para Android, Análisis Estático, Seguridad, Aprendizaje Automatizado, Reducción de Dimensión.

Resum

Actualment, la plataforma Android és el sistema operatiu de major creixement entre els dispositius mòbils. Per això, també ha esdevingut l'objectiu més cobejat i viable per a les aplicacions malicioses.

Aquest treball té com a objectiu el disseny i el desenvolupament de noves formes de detecció d'aquestes aplicacions malicioses en els dispositius basats a l'Android. Més concretament, en aquest treball es exposa el **MaLDroide** (**M**achine **L**earning–based **D**etector for And**roid** malwar**e**), un marc de treball per a la detecció de malware a l'Android utilitzant tècniques d'Aprenentatge Automatitzat.

Per començar, s'extreuen una sèrie d'atributs d'aplicacions recents i reals, tant legítimes com malware. Exactament es recullen 299 atributs agrupats en 8 categories: estadístiques, permisos sol·licitats, permisos utilitzats, cridades a la API, Intents, riscos, cridades al sistema i accés a continguts. El MaLDroide utilitza *anàlisi estàtica* per extreure aquests atributs d'una manera automatitzada.

Com a part essencial del treball, s'ha utilitzat un mecanisme que permet *reduir la dimensió* del problema utilitzant Anàlisi de Components Principals (PCA, per les seves sigles en anglès), passant dels 299 atributs inicials als 27 finals, mantenint un 90% de la variància i reduint espectacularment el temps i l'espai necessaris.

Una vegada determinat el conjunt de dades més adequat, es procedeix a l'*ensinistrament dels classificadors* amb els set algorismes d'Aprenentatge Automatitzat més àmpliament utilitzats: Naïve Bayes, Decision Tree, Random Forest, k-Nearest Neighbor, Support Vector Machine, Multi-Layer Perceptron i AdaBoost.

Finalment, es realitza una avaluació usant 436 aplicacions legítimes i 2296 casos de malware. I a la vista dels resultats obtinguts, es pot concloure que *el classificador Support Vector Machine obté més prediccions correctes, amb una precisió del 99,8%*.

A la vista d'aquests resultats queda patent que el marc de treball presentat és una eina molt eficaç per detectar malware als dispositius amb l'Android.

Paraules clau: Malware en dispositius mòbils, Android, Detecció de malware per a l'Android, Anàlisi Estàtica, Seguretat, Aprenentatge Automatitzat, Reducció de la Dimensió.

Contents

List of Figures

List of Tables

*"Did I ever tell you what the definition of insanity is? Insanity is doing the exact . . . same f*****g thing . . . over and over again expecting . . . shit to change . . . That. Is. Crazy."*

Vaas, in *"Far Cry 3"*

1 Introduction

Malware is an umbrella term, a short form for *malicious software*, actually covering a generic definition for all kinds of computer threats. *Malware* is defined as software designed to infiltrate or damage a computer system without the owner's informed consent. Malicious mobile applications can steal user's information, make premium calls, and send SMS advertisement spams without the user's permission.

Android is the most popular handheld operating system (OS) in the most important markets in America, Europe and Asia, outperforming its rivals iOS and Windows Phone, according to the Kantar Worldpanel ComTech's *"Smartphone OS market share data"*, October 2014. In 2014, research firm IDC stated that more than a billion smartphones will be sold and global market share was 85% for Android, 11% for iOS, 3% for Windows Phone and remaining 1% for all other platforms (Fig. 1.1).

Speaking of malware that infects mobile devices amounts to speaking of *Android malware*. Kaspersky Labs estimates that Android is the target of more than 98% of the mobile malware[10]. Due to its popularity, inevitably Google's Android has attracted many malware writers' attention, and more and more Android malicious applications have been discovered recently. From a purely quantitative point of view, malware designed for Android has the greatest opportunities of success. Android is also an open platform, and the compatibility of Android causes it to become more liable to be attacked by malicious applications than other mobile operating systems [77], for instance iOS or Windows Phone.

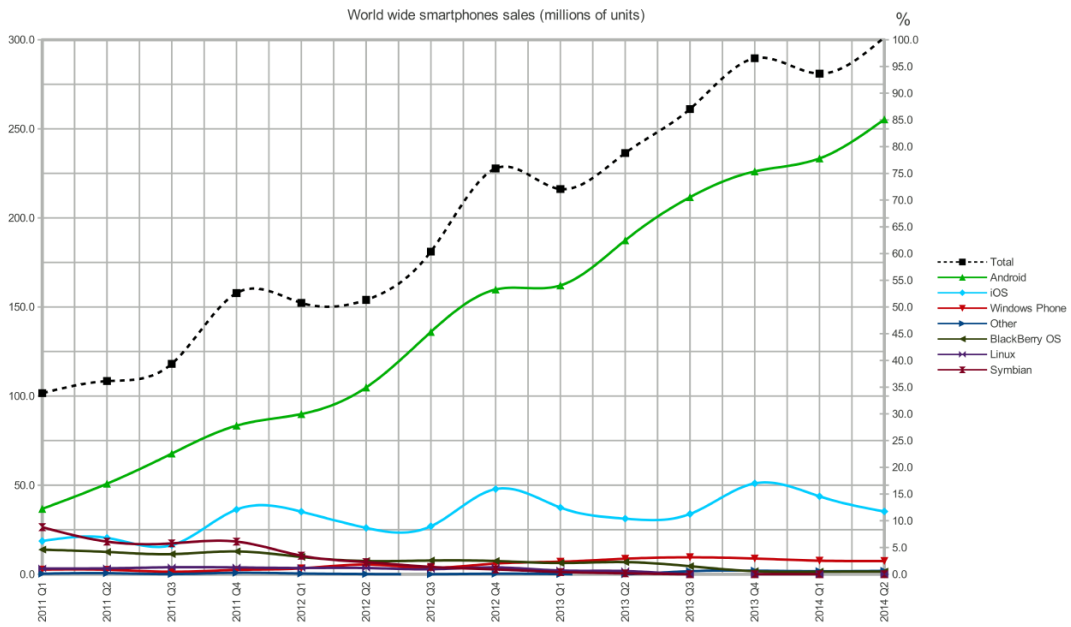


Figure 1.1: World-Wide Smartphone Shipments (Millions of Units — % of Smartphones). Source: wikipedia.

1.1 Motivation

The amount of malware on mobile devices has been growing significantly since it was first discovered in 2010 (Fig. 1.2). With the aim of detecting malware, mobile devices have adopted traditional approaches such as the antivirus. This strategy is not the most efficient against mobile malware, particularly when most malware is evolving rapidly to evade detection by traditional signature-based scanning. Furthermore, an antivirus needs to monitor a device's activities closely to detect malicious behavior. These procedures demand excessive memory and power usage, which becomes a drawback in handheld devices, degrading user experience.

Moreover, malware writers know that the best way to infect as many devices as possible is to attack central app markets. Like Apple, Google also provides a centralized market for Android applications called Google Play Store. However, Android has the ability to install apps from third-party sources: some of them are well-known (such as Amazon or Samsung), but others are not, such as web pages (mostly in Russia, India or China) and memory sticks.

Due to the openness of the Android platform, however, applications may also be installed from other sources.

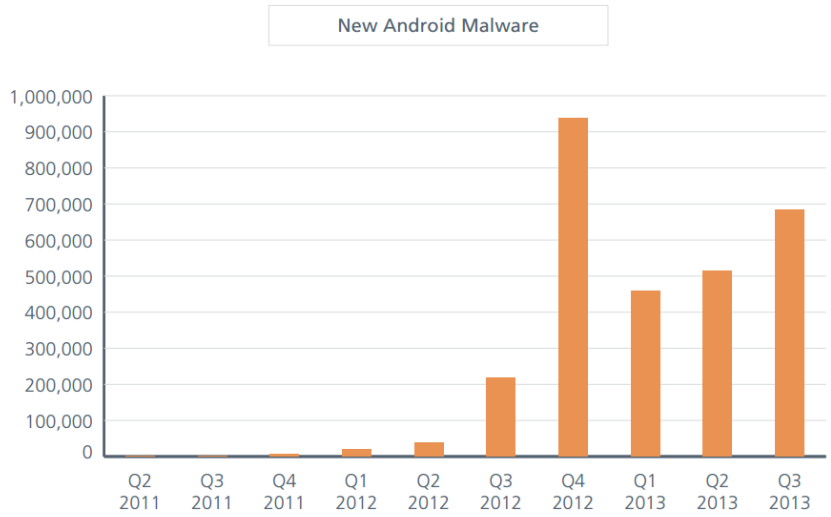


Figure 1.2: McAfee Labs Threats Report: Third Quarter 2013. Source: McAfee Labs.

1.2 Objetives

The work reported here aims to study the best classifier for detecting Android malware using Machine Learning classifiers in order to confront the rapid growth of malware. My interest has been focused, mainly, on the identification of malware, instead of studying it.

Machine Learning classifiers have played a relevant role in the development of Artificial Intelligence (AI) and detection systems. For this reason, my purpose in this work is to evaluate Machine Learning classifiers to provide an alternative solution to the malware detection problem.

In this work, **MaLDroide** (**Ma**chine **L**earning–based **D**etector for And**roid** malwar**e**) will be presented. MaLDroide aims to become a lightweight framework for detection of Android malware that *infers a model that enables identifying malware*. MaLDroide performs a static analysis, gathering as many features from an application's code and manifest as possible.

An important key element in malware detection is a collection of specific attributes by means of which malware could be characterized. In this concern, I will identify the variables (features) that should allow classifiers to determine whether an application is malware or not.

In so doing, I intend to contribute some significant aid in Android malware detection, and defend final users against it.

1.3 Structure of Contents

The materials contained in the present document are organized into five chapters, two appendices, and a bibliographical section.

The why, the aims and scope of the work carried out in this study are stated in this **first chapter**. A glance at the current state of the problem addressed in the study, as well as at some ways in which that problem has been faced out, are offered in **Chapter 2**. The idea and way a proceeding underlying the construction of the framework proposed here for detection of malicious software for Android devices are expounded in detail in **Chapter 3**, and the results obtained after the experiments previously devised are described and analyzed in **Chapter 4**. Finally, a summary of conclusions and open problems appears in **Chapter 5**.

To put the problem in an adequate context, **Appendices A** and **B** provide the reader with some background on the subject. The bibliographical section at the end of the document presents a wide and recent list of sources dealing with the matter treated in the study.

"We've always defined ourselves by the ability to overcome the impossible. And we count these moments. These moments when we dare to aim higher, to break barriers, to reach for the stars, to make the unknown known. We count these moments as our proudest achievements. But we lost all that. Or perhaps we've just forgotten that we are still pioneers. And we've barely begun. And that our greatest accomplishments cannot be behind us, because our destiny lies above us."

Cooper, in *"Interstellar"*

2 State of the Art

Android was released in 2008, and since then a number of studies focusing on analyzing Android security mechanisms have appeared [81].

This chapter contains an overwiew of different recent approaches that have been taken in the field of Android security,

2.1 Android Malware Categories

Most Android-targeted malware can be grouped into the categories of trojan, spyware, root permission acquisition (exploit), installer (dropper), and botnet. Next, I will briefly describe the most common malware in Android handheld system:

1. **Trojan**. It is a harmful piece of software that is intended to appear as legitimate. Users are typically led to download and execute the trojan on their devices. Trojans are also known to create backdoors through which malicious users can get access to the system. Unlike viruses and worms, trojans do not self-replicate; and they do not infect files.

2. **Spyware**. A compound word formed from *spy* and *software*. It is a type of malware that is secretly installed on a device to collect information. It is frequently used for commercial purposes, such as repeatedly opening pop-up advertisement or redirecting users to a particular site. Spyware causes inconvenience by changing a device's settings or becoming difficult to be removed.

3. **Root permission acquisition** (exploit). It uses unknown vulnerabilities or 0-day attacks: it is a new vulnerability that has not been discovered yet, and therefore the said vulnerability has not been patched for. It is a kind of malware that acquires root permission to clear security settings and then makes additional attacks on the Android platform.

4. **Installer** (dropper). It hides malware in a program and guides users to run malware and spyware. The malicious code can be either contained inside the dropper or downloaded by the dropper to the target device.

5. **Botnet**. It is a network of infected devices under the command of an attacker. The attacker, known as the *botmaster*, is able to control the bots and command them to perform malicious activities such as stealing data, intercepting messages and making phone calls without users' knowledge. An tainted device and the botmaster communicate through a rendez-vous point that is called *command and control (C&C) server*.

2.2 Attack Vectors

Because of the nature of mobile devices, they are also open to new types of attack. Malware commonly use three types of penetration techniques for installation, activation, and running on Android platform:

- **Downloading** (Drive-by Download) is the traditional attack technique: malware developers need enticing users to download apps that might appear to be interesting and attractive.

- **Repackaging** is one of the most common techniques that malware developers use to install malicious applications on Android platform. These types of approaches normally start from popular legitimate apps that have been modified (repackaged) by injecting malicious logic or obfuscated program segments so that they have the same structure as the original application, but contain malicious code. Thus, determining whether an application contains a repackaged or obfuscated malware, or whether an application is legitimate, becomes very challenging.

- **Updating** technique is more difficult for detection. Malware developers may still use repackaging but, instead of enclosing the affected code to the app, they include an update component which is devised to download malicious code at runtime.

2.3 Malware Detection

2.3.1 Detection Analysis

For malware detection, there are two common practices: static and dynamic analysis of software. Both have advantages and disadvantages, and numerous approaches to both static and dynamic analysis paradigms do exist.

2.3.1.1 Static Analysis

Static analysis involves various binary forensic techniques, including decompilation, decryption, pattern matching and static analysis of system calls. All of these techniques have in common that the software is not being executed.

Many static analysis techniques construct *graphs based on code structure*. One of these graphs is the **data flow graph**. Elish et al. [21] use data flow as a source of features by determining if risky API calls are connected in some way to data input by a user. For example if an application sends a text message containing information provided by the user, then it is probably safe. If a text message is sent containing information generated without the user's input, it will probably be malicious.

A second useful graph is the **control flow graph**. It maps blocks of a program to nodes, and possible paths of execution are the edges. *Woodpecker* [33] is a tool for detecting security leaks in Android devices. It uses control flow graphs to search for paths of execution that allow the use of dangerous API calls without requesting permissions.

The last graph is the **dependence graph**. In this kind of graph, nodes represent components of the program and the edges represent some sort of dependence. Walenstein et al. [90] use such a class dependence graph to detect isolated classes within repackaged applications. Isolated classes are more likely to contain malicious code that might have been injected. It is possible to detect isolated nodes in a graph with centrality metrics, like *betweenness*[1] and *closeness*[2]. A node with high closeness and low betweenness can be considered isolated.

Opcodes[3] are often used as features in static analysis. Many malicious activities have characteristic opcode patterns that can be detected by classifiers. Opcodes are also useful in detecting similarities between parts of programs. *DroidMOSS* [100] uses a sliding window to compare large sections of opcodes in an unverified application to opcodes of other known applications.

[1] Betweenness centrality measures how often a node appears on shortest paths between nodes in the graph.

[2] Closeness centrality of a node refers to the average distance from that node to all other nodes in the graph.

[3] An *opcode* is the part of a machine language instruction that determines the type of operation to be performed.

2.3.1.2 Dynamic Analysis

Dynamic analysis, also called **taint analysis**, is a set of techniques which involve running an application in a controlled environment and monitoring its behavior. Various heuristics have been used to best capture and analyze dynamic behavior of applications, such as monitoring file changes, network activity, processes and system call tracing.

A **sandbox** is a protected area for running untrusted applications that does not allow malicious behavior to influence anything outside the protected area. For example, an isolated virtual machine can act as a sandbox for testing malicious applications. Researchers can observe malicious activity without risking infection of other devices or leaking sensitive information.

Since February 2012, Google uses a verification service named **Bouncer**. Bouncer uses a sandbox to automatically scan applications submitted to the Android Play Store for malware. As soon as an application is uploaded, Bouncer checks it and compares it to the behavior of previous malicious apps. Unfortunately, as Oberheide and Miller [62] demonstrated, there are many ways to circumvent this sandbox testing.

Building a sandbox that is indistinguishable from a physical device is one of the great subjects of current research.

A significant challenge for dynamic analysis is **input generation**. Many malicious applications initiate suspicious activity without regard for user interaction, but more stealthy attacks could wait for a user to perform a predefined action before activating. To detect this type of malware, systems that can simulate human input are needed. Input generation systems are evaluated by the amount of code coverage[4] that they achieve.

The Monkey[5] is a system that helps developers to test applications by producing pseudo-random input during program execution. *The Monkey* does not simulate system events such as receiving a phone call. Just as malware could wait for a user action, it can also respond to system events. *The Monkey* achieved code coverage of 53%.

Dynodroid [56] is a system for input generation. *Dynodroid* simulates system events and allows a human to assist with input generation during testing. It uses an observer to determine which events are relevant.*Dynodroid* achieved code coverage of 55%.

2.3.2 Detection Methodologies

Commonly, Android malware detection techniques can be classified into three detection techniques, namely signature-based detection, anomaly-based detection and behavior-based

[4] **Code coverage** refers to the amount of source code that has been executed. For example, a program that has been tested with 100% code coverage will have executed every piece of code in the program.

[5] http://developer.android.com/tools/help/monkey.html

detection [52]:

Signature-based detection technique is a traditional method used to detect malware in personal computer environment. Signature based methods [45], introduced in the mid90s, detect the presence of a virus by using short identifiers called *signatures*, which consist of sequences of bytes in the code of the infected program. The major drawback of this approach is that it cannot detect metamorphic or unknown malware.

Static and dynamic methods can be used to define signatures. Static analysis targets the code without actually running a program. Dynamic analysis is a method of searching for certain patterns in memory leakage, traffic and data flow while running the program.

Anomaly-based detection monitors regular activities in the devices [74], and looks for any behavior that deviates from the normal pattern . Zhou et al. [102] have applied both the signature-based and anomaly-based detection technique in their Android malware detection system.

Behavior-based detection technique [9] is a method of detection focused on analyzing the behavior of a program to conclude whether it is malicious or not. Behavior-based technique usually applies Machine Learning approach for learning patterns and behaviors from known malware, and then to predict the unknown one.

Apart from traditional approaches, other researchers resort to Machine Learning techniques. Instead of using predefined signatures for malware detection, Machine Learning techniques provide an effective way to dynamically extract malware patterns of behavior. Gavrilut et al. [31] applied Machine Learning tecniques to identify patterns of behavior for viruses in Windows executable files, by using a simple multi-stage combination (cascade) of different versions of the perceptron algorithm.

2.4 Related Work

Although Android malware is a relatively new interesting topic, there has already been a plethora of research in this field. Just after Android was released in 2008, the contributed paper [75] was one of the first studies focused on analyzing Android security mechanisms (actually, on its Linux side).

In the context of Machine Learning methods, the first major contributions for detecting malware were proposed in [76, 49, 42]. These authors tried to detect unknown computer

viruses by using classifiers like Decision Trees, Bayesian Networks, Support Vector Machine (SVM), and Boosting, among others.

Enck et al.[25] was one of the first studies concerning Android permission model, internal components and their interactions. Schmidt et al.[73] perform a static analysis of the executables to extract function calls in Android environment. Function call lists are compared with malware executables for classifying the function calls. Polla et al.[66] survey some security solutions for mobile devices.

Kirin [24] is a logic–based tool for Android that performs a lightweight certification of applications to mitigate malware at install time.

SCanDroid [30] presents a methodology for static analysis of Android applications to identify privacy violations. It extracts security specifications from manifest, and checks whether all data flows through that application are consistent with the aforesaid specifications.

Bläsing et al.[7] describe another dynamic analysis system for Android, focused on classifying applications as malicious (or not), by scanning those applications according to criteria establishing potential danger.

TaintDroid [22] monitors applications for the leakage of sensitive information, implemented upon the Dalvik Virtual Machine. Private data are considered tainted, and anything that reads them will also be considered tainted, and then tracked. If private data leave the Android device, the user is provided with a report. *DroidBox* [51] uses *TaintDroid* to detect privacy leaks, but also comes with a patched Dalvik Virtual Machine to monitor the Android API and to report on file system and network activity, the use of cryptographic operations and cell phone usage. It then provides a timeline view of the monitored activity. *DroidBox* is useful for manually identifying malware by viewing its observed behavior.

ComDroid [14] performs static analysis of decompiled bytecode of Android applications to find Android Intents sent with weak permissions.

Crowdroid [12] is a dynamic analysis and Machine Learning-based framework that recognizes Trojan-like malware on Android smartphones, by analyzing the number of times that each system call has been issued by an application during the execution of an action that requires user interaction. Collected observations are classified using K-Means, showing a detection rate of 100%. The main difference with other approaches is that authors analyze the monitored features in the cloud.

In 2012, researchers at the North Carolina State University published a data sample, called **Android Malware Genome Project**, that consists of 49 different Android malware families with a total of 1,260 malware cases [101].

There are also studies [27, 4, 16, 65] on the use of static-based detection methods relying on permissions which require a small amount of resources and enable a quick detection. However, these methods have a relatively low detection rate. In case of dynamic detection methods [96, 32], they exhibit a high detection rate, but require a lot of resources, and detecting malicious behaviors takes them a long time.

DroidRanger [102] uses imported packages and other features extracted from the application in order to achieve malware classification, and also extracts system calls made by native code and looks for attempts to hide this code.

DroidMat [94] claims to outperform *Androguard* [17] in accuracy and efficiency. DroidMat uses several features including permissions, deployment of components, intent messages, and API calls. It also performs different types of clustering to classify applications as benign or malicious.

Andrubis [53, 91] is an extension of the Anubis service: a platform for analyzing unknown applications. Andrubis was the first dynamic analysis platform for Android available as a web application since May 2012. Users can submit Android applications and receive a detailed report including a maliciousness rating when analysis is finished.

MADAM[18] (Multi-level Anomaly Detector for Android Malware) uses 12 features to detect Android malware at both kernel and user level. Together with the K-Nearest Neighbors (KNN) classifier, MADAM successfully obtained a 93% accuracy rate for 10 malwares. However, it is incapable of detecting malware that avoids the system call with root permission, for example SMS malware.

Andromaly[80] is a host-based malware detection system that continuously monitors smartphone features and events. Andromaly relies on Machine Learning techniques monitoring both the smartphone and user's behaviors by observing 88 features to describe these behaviors, ranging from activities to CPU usage. Andromaly identified the best classification method out of six classifiers (DTJ48, NB, Bayesian Networks, k-Means, Histogram and Logistic Regression), and achieved a 99.9% accuracy rate with the decision tree (J48) classifier. Although its authors achieved great accuracy, they had used self-written malware to test their framework.

RobotDroid[99] is based on the SVM classifier to detect unknown malware in mobile devices. The focus was on privacy information leakage and hidden payment services. The authors evaluated three malware types (Gemini, DroidDream and Plankton). This framework is limited to a few malware types.

In their turn, Sanz et al.[71] worked out PUMA (Permission Usage to detect Malware in Android), a framework to detect malicious Android applications through Machine Learning techniques by analyzing the extracted permissions.

Kim et al.[46] proposed a malicious application detection framework on Android market performing both static and dynamic detection methods using Machine Learning. Saranya et al.[88] not only check permissions but their treatment also involves a feature selection method that finds the best performance in detecting new malware.

TStructDroid [82] constitutes a real-time malware detection system by using theoretical analysis, time-series feature logging, segmentation and frequency component analysis of data, and a learned classifier to analyze monitored data. *TStructDroid* shows a 98% accuracy and less than 1% false alarm rate.

MADS [72] (Malicious Android applications Detection through String analysis) extracts the strings contained in applications to build Machine Learning classifiers and detect malware. Huang et al.[44] used static analysis and evaluated four classifiers (AdaBoost, NB, DT48 and SVM) on permissions to detect anomalies. Then, this authors claimed that permission-based analysis is a quick filter for identifying anomalous applications.

STREAM [2] uses an Android emulator to collect selected features such as battery, memory, network and permission. It then applied several Machine Learning classifiers to the collected data. Liu's approach [54] is a malware detection method based on the Multiple Classifier System (MCS) [92], wherein each base classifier (SVM) is responsible for detecting only one kind of malware.

Yerima et al. [97] present an approach based on Bayesian classification models obtained from static code analysis. The models were built from a collection of code and app characteristics that provide indicators of potential malicious activities. This models were evaluated with real malware samples. Canfora et al.[13] compute the occurrences of a set of system calls invoked by the application under analysis. The fundamental hypothesis is that different malicious applications can share common patterns of system calls that are not present in normal applications, since these common patterns are recurrently used to implement malicious actions.

In [26], Feizollah et al. evaluate 5 Machine Learning classifiers, and results were validated using malware data samples from the Android Malware Genome Project. These authors consider only three network features: connection duration, TCP size and number of GET/POST parameters. Narudin et al.[61] proposed an alternative solution combining the anomaly-based approach with Machine Learning classifiers, evaluating that solution on two data sets, namely MalGenome[101] and a self-collected private data set. In 2013, Han and Choi [37] published their study in which they applied Machine Learning to selected features, evaluating 5 malware families.

A5 [89], short form for Automated Analysis of Adversarial Android Applications, is an automated system to process Android malware, combining static and dynamic malware

analysis techniques. Main innovation in A5 consists of platforms, both virtual and physical, to capture behavior that could otherwise be missed.

Dendroid [86, 87] is an approach, based on Text Mining and Information Retrieval techniques, to analyze and classify code structures in Android malware families. These authors use those code structures in order to investigate hierarchical relationships among malware families.

DroidDolphin [95] is a dynamic analysis framework based on Big Data and Machine Learning to detect malicious Android applications. Liu and Liu [55] propose a scheme for detecting malicious applications using a decision tree classifier. In contrast to other previous researches, they consider requested permission pairs as an additional condition, and also consider used permissions to improve detection accuracy.

Protsenko and Müller [68] proposed a static detection method based on software complexity metrics (involving control and data flow metrics as well as object-oriented design metrics); such a method shows resilience against common obfuscation transformations.

Finally, DREBIN [3] performs a broad analysis of Android applications to detect on-device malware by using Machine Learning based on features like requested hardware components, permissions, names of app components, intents, and API calls. However, it is unable to detect malwares that use obfuscation technique and dynamically loaded code technique.

The aforementioned approaches, summarized in Tables 2.1 and 2.2, demonstrate the logic behind applying Machine Learning classifiers to detect Android malware.

	Year	Detection Approach	Features	Observations
Kirin[24]	2009	Static.	Subset of the requested permissions.	Lightweight certification of apps at install time: Kirin recommends against the installation of apps when certain permission combinations appear.
SCanDroid[30]	2009	Static.	Security specifications from manisfest.	SCanDroid is built using Ruby.
TaintDroid[22]	2010	Dynamic.	Variables, methods, files, and messages.	Tracking of variables, methods, files and IPC via dynamic tainting, and enforced by the user.
Crowdroid[12]	2011	Dynamic. Clustering.	System calls per application.	Clustering with k–means.
DroidRanger[102]	2012	Dynamic and Static.	Two schemes of detection: static analysis of the manifest and dynamic monitoring of imported packages and native code through heuristics.	Detecting malware in Android markets. About 0.1% of the 204.040 analyzed apps are malicious.
MADAM[18]	2012	Dynamic. ML.	Kernel: system calls, processes, free memory, and CPU usage. User: user–state, key strokes, called numbers, SMS, and Bluetooh/Wifi.	K-NN (K=1) classifier. 10 malicious apps and 50 benign. 93% detection rate and 5% FP
RobotDroid[99]	2012	Machine Learning	Intent issued and system resources accessed.	This framework is based on the SVM classifier. It focused on privacy information leakage.
Andromaly[80]	2012	Dynamic. ML.	Detection Method: monitorization of a subset of 88 initial features.	Classification with labelled data. Experimental evaluation.
PUMA[71]	2012	Static. ML	Permissions	PUMA analyzes the extracted permissions through Machine Learning techniques. Accuracy: 88.24%.
DroidMat [94]	2012	Static and dynamic. Clustering.	Permissions, deployment of components, intent messages, and API calls.	Detection rate: 97.87%.
DroidMOSS [100]	2012	Dynamic.	Fingerprints	It detects repackaged apps in markets using fuzzy hashing; however,it depends on the existence of the corresponding original app. Not designed for general malware detection.

Table 2.1: Malware detection systems (1/2)

	Year	Detection Approach	Features	Observations
Andrubis [53]	2012	Static and dynamic. ML	File and network operations, data leaks, crypto operations, classes and native libraries loaded, SMS and phone calls, and started services.	The analysis provides a rating on a scale 1-10 (10 being likely malicious)
TStructDroid[82]	2013	Dynamic. ML.	99 preliminary parameters.	Accuracy: 98%.
MADS[72]	2013	Static.Text Mining & ML.	Permissions and string constants	Evaluation over 2 data sets: one with 357 benign and 249 malware, the other with 333 benign and 333 malware. It evaluated 6 algorithms. Accuracy of 92.04% obtained the RF (100 trees).
Yerima et al.[97]	2013	Static. ML.	Permissions, APIs, and commands	Bayesian classification. Their experiments had 100, 250, 500, 1,000 and 1,600 samples. The highest detection rate is 90.60%
STREAM[2]	2013	Dynamic. ML.	Battery, memory, network and permissions	It applies several Machine Learning classifiers.
DroidAPIMiner [1]	2013	Static. ML.	API calls and requested permissions	DroidAPIMiner is built upon Androguard, evaluates different classifiers, and achieves an accuracy 99% using KNN classifier
DenDroid[87]	2014	Static. Text Mining & Information Retrieval.	Code structures, a high-level representation of the Control Flow Graph (CFG)	Automatic classification of unknown malware. The classification is done using Text Mining and Information Retrieval techniques.
DroidDolphin[95]	2014	Dynamic. ML and Big Data.	Log files contain tagged events: API calls and activities.	It applies the SVM classifier.
DREBIN[3]	2014	Static. ML.	Permissions, hardware access, API calls, and network addresses	It trains an SVM to find a hyperplane separating benign apps from malware. Detecting 94% of malware.

Table 2.2: Malware detection systems (2/2)

"Knowledge is useless if it is not used."

Gixx, in *"Guild Wars 2"*

3 Methodology

The Android malware detection method proposed in the present work is discussed in detail in this chapter. This chapter contains the overall work flow of the experiments. Figure 3.1 illustrates the experiment work flow structure consisting of four phases. The first stage is **data collection**, which collects normal and malicious applications. In the second phase, which is **feature selection and extraction**, features selected are extracted, labelled and stored to be applied in the next phase. The **Machine Learning classifiers** entail the third phase, whereby the stored information trains the Machine Learning classifiers to produce several detection models. The last phase is the **evaluation** and choice of a classifier based on empirical data obtained, in order to build our framework.

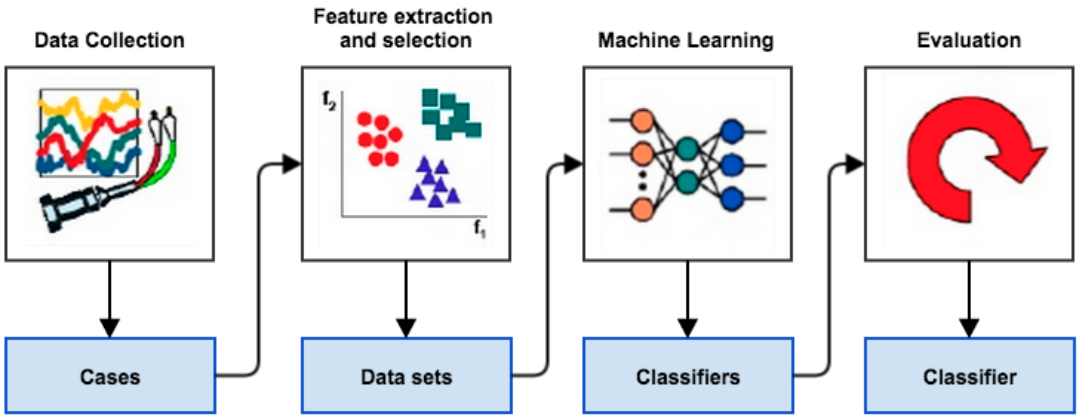


Figure 3.1: The experiment work flow structure

The following sections describe each phase in detail.

3.1 Data Collection Phase

3.1.1 Benign Applications

The top successful applications at Google Play[1] Store for Android are selected as benign applications. The benign cases (436) consist of apps available on Google Play in November 2014. Although there is a relatively small number of benign cases vs. that of the malicious ones, I have supposed apps to be mostly malicious.

The most popular applications from the Android official market have been included, and I have also decided to include some "scam apps" (i.e. wifi boosters, battery doctor, photo galleries, ...) because those kinds of apps usually hide malware.

3.1.2 Malicious Applications

Several sources were considered for use as a malicious data set. Our malware cases consist of 2295 malicious apps grouped into different families. A malware family is basically a collection of malware presenting a similar behavior. The selected malicious applications belong to trojans, adware, rootkits, bots and backdoors categories.

Among the malicious apps, some 1900 apps are taken from DREBIN[3], 331 from contagio [64], and the rest of malicious apps are collected from the Andriod Malware Genome Project [101]:

- MalGenome data set. The Android Malware Genome Project is a malware repository that covers the majority of malware families for Android (Table 3.1). This repository contains 1247 malicious apps grouped into 49 different families. These samples included specimens with a variety of infection techniques (repackaging, update attacks, and drive-by-download) and payload functionalities (privilege escalation, remote control, financial charge, and private information leakage).

- DREBIN data set. The data set contains 5,560 applications from 179 different malware families. The samples have been collected in the period of August 2010 to October 2012.

- Contagio Mobile Dump. Contagio Mobile is a public malware repository managed by a group of independent security researchers and counts on a great support within

[1] https://play.google.com/store

Malware	Description
SMS	The malware sends Premium SMS messages charged against the user.
Banking	The malware is designed specifically to intercept Banking messages.
Root	The malware uses a rootkit as a method of attack.
Info	The malware uploads personal information to a remote server, without notifying the user.
Spyware	The malware remains on in the background, or has the capability of remotely monitoring the smartphone
Botnet	The malware exhibits a behavior associated with botnets, e.g. accepts remote commands.
Market	The malware was spotted in the official Google Play Store

Table 3.1: Malware Genome Project's malware classes.

the malware community. This data set contains 331 samples, some of which are very recent (November 2014).

3.1.3 Samples Description

Our samples consisted of a total of 2731 applications split into 2295 malicious apps and 436 clean apps. There were no duplicate programs in the cases and every sample is labelled as either malicious or benign.

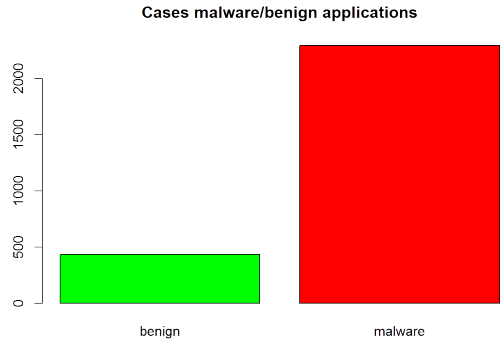


Figure 3.2: The distribution of applications (malware/benign) in the data set

Once the data collection phase was finished, I ended up with a less benign cases than I would have expected, as seen in Fig. 3.2. However, this should not affect the further analysis. After verification of the samples, the next step of this method was devoted to extract features from the collected applications.

3.2 Feature Extraction and Selection Phase

In the feature extraction phase, MalDroide statically examines the collected benign and malware apk samples to determine and extract the necessary features. Besides, it analyzes

the decompiled source code of an application, by identifying used permissions required by the application, API calls, as well as other features for malware detection.

Behavior of both malicious and benign applications are profiled with a set of feature vectors [2]. In order to apply any Machine Learning technique, relevant features from every application have been extracted. Android applications are delivered as apk files. I decide to use an open source project, namely *Androguard* [17], as support to process these files and extract features.

Analysts [94] have observed that malicious applications have significantly more permissions than benign ones. This is expected since permissions allow applications to perform actions that can potentially harm the user. Malicious applications also tend to request unusual permissions when compared to legitimate ones [5].

Let an application feature F_i, obtained from digging in the apk by MaLDroide, be defined as:

$$F_i = \begin{cases} 1, \text{if discovered by MaLDroide,} \\ 0, \text{ otherwise.} \end{cases}$$

Every application is assigned a vector defined by $f = (f_1, f_2, \ldots, f_n)$, where f_i is the result of the ith variable F_i.

MalDroide uses the information extracted with the help of Androguard to construct the following feature categories:

Stats: MaLDroide retrieves the following statistical information from the apk, when available. MaLDroide gathers 11 statistical features:

- Number of classes, packages, permissions, activities, services, receivers, and providers.
- Minimum and target SDK version.
- If apk contains URL's or IP's hardcoded.

Requested Permissions: One of the most important security mechanisms introduced in Android is the permission system. Every Android application must include, in a manifest file, a list of request permissions to access certain restricted elements, such as hardware devices (camera, GPS, file system, ...), sensitive features of the system (contacts, bookmarks, ...), and access to certain exposed parts of other applications. Malicious software tends to request certain permissions more often than innocuous applications [55, 16, 65]. For example, INTERNET requests the right to access the internet, READ_CONTACTS requests the right to access the users contact list, and a

great percentage of current malware sends premium SMS messages and thus requests the SEND_SMS permission.

MaLDroide gathers 107 permissions both dangerous and harmless.

Used Permissions: This feature set is a list of all used dangerous permissions. There is not a file that describes the permissions actually used by an app. Therefore, MaLDroide needs to analyze the dex file to identify which permissions are actually used by the app. MaLDroide selects a total number of 62 used permissions.

API Calls: Certain API calls allow to gain access to sensitive data or resources, and are frequently found in malware samples. MaLDroide collects the following types of API calls:

- API calls for accessing sensitive data, such as `getSubscriberId()`, `getDeviceId()`, `getLine1Number()`, `getSimSerialNumber ()`, `getNetworkOperator()`, and `getCellLocation()`.
- API calls for sending and receiving SMS messages/phone calls, like `sendTextMessage()`.
- API calls for execution of external commands, such as `ClassLoader.loadClass()` and `System.loadLibrary()`.
- API calls for network communication, like `HttpURLConnection()` and `Socket()`.
- API calls frequently used for encrypting, like `Cipher.getInstance()`.

MaLDroide gathers 56 potentially dangerous API calls.

Intents: Inter-process communication on Android is mainly performed through intents. MaLDroide collects a set of suspicious intents as another feature set. A typical example of an intent message involved in malware is `BOOT_COMPLETED`, which is used to trigger malicious activity directly after booting the device. MaLDroide selects 27 suspicious Intents.

Risks: Androrisk is another utility of Androguard tool set, which identifies "red flags" based on permissions, shared libraries, and other risk factors [20]. MaLDroide checks the following risk factors (5):

- Dynamic Code: DexClassLoader is a class loader that loads classes from .jar and .apk files containing a classes .dex entry. This can be used to execute code not installed as a part of an app.
- Native Code: Android apps can load code written in C/C++ by using the Native Development Kit (NDK). Native code can be used to boost application performance for resource intensive applications. Nevertheless, the use of native code increases the risk of the app.
- Java Reflection: Java Reflection is used to examine or modify the runtime behavior. Using this characteristic, Android applications can load Java classes at runtime, which may be used to circumvent API restrictions.
- Crypto: Androrisk searches for the use of cryptographic libraries that may obscure part of the code that avoids a static analysis.
- Score: Using Fuzzy Logic, Androrisk calculates a score (0-10, 10 meaning the most risky score) of risk factors.

System Calls: Linux kernel is executed in the lowest layer of Android architecture (Fig. A.1). Calls to system commands, like `chmod`, `rm`, `mount`, or `su`, are registered. In total, MaLDroide collects 19 different Linux system calls.

Content Access: Content providers manage to access to central repositories of data in Android system. A special form of URI, which starts with `content://`, is assigned to each content provider. Any app can query the inbox from the device using its URI `content://sms/inbox`, although `READ_SMS` permission must be required in order to access. MalDroide locates access via URI to 12 content providers.

First, MaLDroide decompress and decompiles the apk file to retrieve the content. MaLDroide uses Androguard tools to extract the information from the apk files. Androguard can turn them into Java's bytecode. Next, it processes the Android manifest file to extract data (mainly, requested permissions). MaLDroide looks up into the bytecode identifying selected API calls, dangerous used permissions, suspicious system calls and content access (Table 3.2).

Finally, MaLDroide stores a vector of features similar to Fig. 3.3.

Not all of the collected information items will be used in the analysis, but it is kept in the data set in case of future study.

Lastly, I need to gather all generated vectors to shape the data set. This is simply done with a script in R to construct the full data set.

Category	Feature	Comments
Stats	stat__NUMBER_OF_CLASSES	Number of classes implemented by application.
	stat__MIN_SDK_VERSION	The minimum system API with which it is compatible.
Permissions	permission__android.permission.READ_PHONE_STATE	Access to unique identification numbers of your device (IMEI, IMSI Google ID).
	permission__android.permission.BRICK	Can remotely disable the device!!
	permission__android.permission.DELETE_PACKAGES	Can delete any installed apps.
	permission__android.permission.PROCESS_OUTGOING_CALLS	Can monitor what phone numbers call and collect other personal info!
Used Permissions	used_permission__android.permission.MODIFY_PHONE_STATE	Access to power and mmi state modifications.
	used_permission__android.permission.SYSTEM_ALERT_WINDOW	Allows an app to show a popup window on top of other apps.
	used_permission__android.permission.READ_SOCIAL_STREAM	Allow an app to read updates from social networking apps like Google+, Twitter, and Facebook.
API calls	api_call__Landroid/telephony/TelephonyManager...textgreatergetCallState	Application reads the phone's current state.
	api_call__Landroid/location/LocationManager...textgreatergetProviders	Application reads location information from all available providers (WiFi, GPS etc.)
	api_call__Landroid/telephony/TelephonyManager...textgreatergetSimCountryIso	Application reads the ISO country code equivalent to the SIM provider's country code.
Intents	intent__android.intent.action.AIRPLANE_MODE_CHANGED	Application receives this intent when device changes to "airplane mode".
	intent__android.intent.action.BOOT_COMPLETED	Application receives a "boot completed" when device restarts.
System calls	system_call__rm	Application calls rm (remove file) linux command.
	system_call__mount	Application calls mount linux command.
Risks	risk__REFLECTION	Application uses Java's Reflection.
Content	content__content://sms	Application reads the SMS inbox .

Table 3.2: A non-exhaustive list of some of the features extracted by MaLDroide.

```
com.whatsapp:<1,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,0,1,1,0,1,1,1,0,1,1,1,1,1,1,1,
1,1,1,1,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,0,1,0,0,1,1,1,0,0,
1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,0,1,1,0,0,0,0,1,1,0,1,1,1,0,0,
1,1,1,1,0,0,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,1,0,0,1,0,0,
0,1,0,1,1,1,0,1,0,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,0,1,1,1,
1,1,1,0,1,1,7,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,0,1,0,0,
1,0,0,1,1,0,1,1,0,0,1,1,0,0,0,0,1,1,1,0,0,0,0,1,1,0,1,0,0,1,
1,0,0,0,0,0,0,1,0,0,1,1,1,1,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,
1,0,0,1,1,0,benign>
```

Figure 3.3: An example of feature vector for the WhatsApp application retrieved with MaLDroide

3.2.1 Android App Reverse Engineering: Androguard

Android applications are written in Java and compiled by the Android SDK tools with any data and resource files into an Android package.

Androguard [17] is a popular static analysis tool for Android applications. It can disassemble and decompile Dalvik bytecode. It can also compute a similarity measure to detect repackaged apps or known malware. It also has modules that can parse and retrieve information from the AndroidManifest.xml. Due to its flexibility, it is used by some other analysis frameworks that need to perform some static analysis.

This tool has three different decompilers: DAD, DED and JAD. The one used by default is DAD, which is also the fastest one, due to the fact that it is a native decompiler. Additional features, such as visualizing the application as a graph and permissions examination, are available as separate scripts.

To facilitate the Machine Learning detection approaches in the present work, I have implemented an Androguard-based Android analysis tool for automated reverse engineering of the apk files, named `MaLDroide.py`.

3.2.2 Feature Selection

In Machine Learning applications, a large number of extracted features, some of which redundant or irrelevant, present several problems such as misleading the learning algorithm, over-fitting, reducing generality, and increasing model complexity and run-time.

Applying the most suitable feature selection at a preparatory stage enables to use our malware detector more efficiently, with a faster detection cycle. Nevertheless, a reduction of the amount of features should be performed in such a way that a high level of accuracy is preserved.

Full generated data set contains 299 features, most of which may be redundant and do not contribute significantly to classification. Accordingly, it is necessary to reduce the number of features to an appropriate level. In the present work, the feature reduction method has been *Principal Component Analysis* (PCA).

Principal Component Analysis (PCA) [85] creates a number of *principal components* which is less than or equal to the total number of available variables. Each principal component (PC) should exhibit a high variance with other components. Top principal components are orthogonal to each other, and have minimal correlation. A principal component can be expressed as a linear combination of original features (Fig. 3.4), strategically chosen so as to produce a reduction of dimensionality.

```
PC1: 0.102*permission__android.permission.RECORD_AUDIO +
     0.102*permission__android.permission.UPDATE_DEVICE_STATS +
     0.098*api_call__Landroid.app.ActivityManager...
        restartPackage
PC2: 0.177*used_permission__android.permission.NFC +
     0.177*permission__android.permission.NFC +
     0.177*used_permission__android.permission.
        AUTHENTICATE_ACCOUNTS
    ...
PC27: 0.207*permission__android.permission.MODIFY_AUDIO_SETTINGS+
      0.185*api_call__Landroid.telephony.TelephonyManager...
         getNeighboringCellInfo-
      0.175*api_call__Ljava.lang.ClassLoader...loadClass...
```

Figure 3.4: Top Principal Components.

Data Set	Number of Features	Number of Cases	Comments
Full	299	2731	
PCA5	43	2731	95% of variance preserved.
PCA3	27	2731	90% of variance preserved.

Table 3.3: Data sets used in the experiments

In Table 3.3, a description of the data sets used in the experiments of this study can be found.

3.2.3 Data Set Discussion

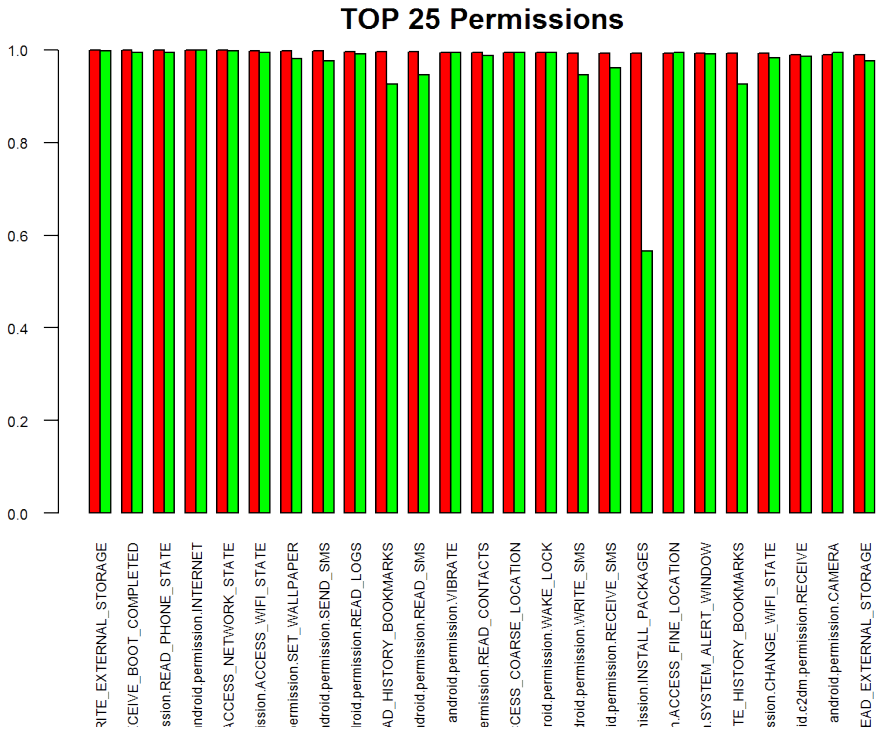
Before digging into next phases, I am going to have a quick look at the most frequently used permissions in Android applications. Fig. 3.5 shows that malware requires relatively more permissions with a fewer implemented classes. An important point to be taken into account is that not all the permissions requested by a malicious application are necessarily required for it to do its malicious work [8]. For example, when a trojan is attached to a legitimate application (repackaging), the resulting application requires the permissions of the original application in addition to the permissions that it requires for its own malicious purposes.

Figure 3.5: Number of classes vs number of permissions in malware/benigns applications.

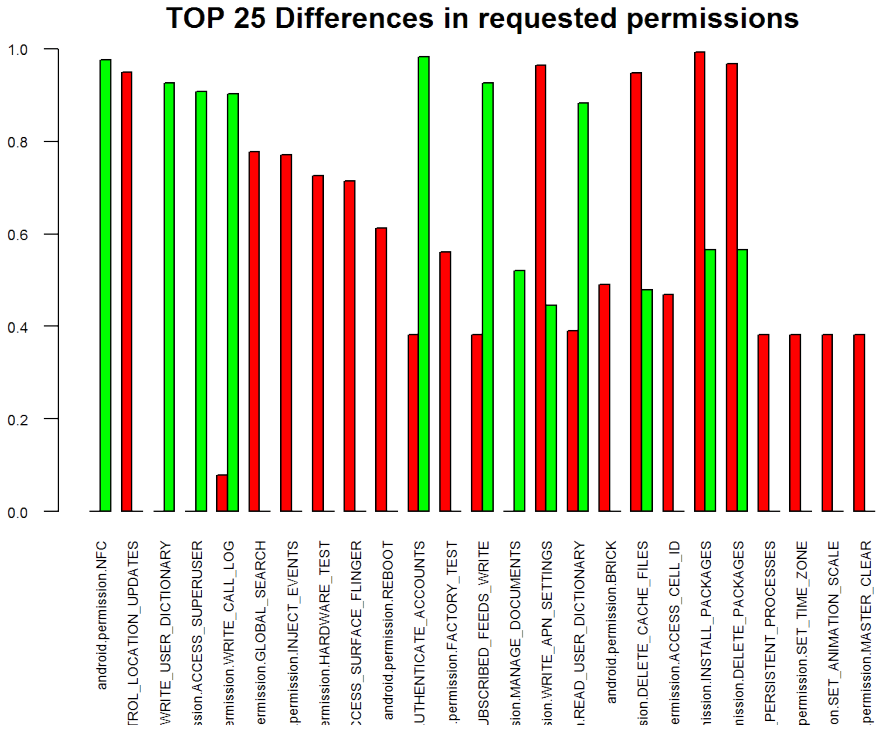
Fig. 3.6a shows the top 25 permissions requested by both malicious and benign applications in the data set.

We can see that the permission `INTERNET` is requested by every application (100%), either malware or legitimate. Therefore, this is a feature in the data set which will provide us very little information, since it presents no variability.

As a guiding principle, I am mainly interested in such permissions for which there is a substantial difference between malware and legitimate application requests. Fig. 3.6b shows the major differences between the most requested permissions, depending on legitimate applications or malware.



(a) Top 25 requested permissions.



(b) Top 25 major differences in requested permissions between malware and benign applications.

3.3 Machine Learning Phase

The third phase of this methodology, just the one truly involving a Machine Learning approach, includes the training of the classification models from the collected data.

The experiments described in this work compare runs of Machine Learning algorithms with and without feature selection from the data sets described in the preceeding section.

I have resorted to WEKA (Waikato Environment for Knowledge Analysis) [93, 36] to train and evaluate different algorithms used in the experiments. WEKA is a collection of Machine Learning algorithms for Data Mining tasks.

As above described, our data sets have 2731 cases. Typically, in a Machine Learning approach, not all cases will be used to train the classifier. Moreover, only a relatively small percentage of the data set (called the **training set**) will be used for training the classifier; most cases (the so–called **test set**) in the data set will be used to verify the classifier. Accordingly, the classifiers are trained on the training set, and then the generated classifiers are verified with the help of data in the test set.

3.3.1 Machine-Learning Algorithms

A huge diversity of classifiers can be used for Machine Learning techniques. After extensive study of existing Machine Learning approaches, one can highlight disadvantages and advantages of the following algorithms, which I consider to seem to be the most appropriate for the task of malware detection:

- **Naïve Bayes** (NB) [48] assumes the features to be 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.

- **Decision Tree** (J48) [50] is a tree relying on the feature values to classify instances. A decision tree consists of nodes and leaves. Nodes perform evaluations of features, and leaves contains the reached conclusion (namely, whether the app under considetation is classified as malware or benign).

- **Ramdom Forest** (RF) [11] combines decision trees made up by the independent random features to draw a conclusion, and achieves a relatively high detection rate. Random Forest is a Machine Learning classifier frequently used in malware detection studies in the Android environment [47, 37].

- **K-Nearest Neighbor** (KNN). Although it is a very simple algorithm (an instance of the so-called "lazy algorithms") and exhibits fast performance, it becomes inappropriate when the training set suffers from noise or outliers

- **Support Vector Machine** (SVM) has a robust theoretical and conceptual background, thanks to which its performance regarding classification results is generally better than that achieved by other algorithms. However, it is conceptually complex, hard to interpret, CPU– and memory–intensive, and performs well on linearly-separable data; otherwise, it requires involved non-linear transformations by means of kernels. In the present work, the linear kernel has been chosen.

- **Neural networks** (NN) [34] is another Machine Learning technique inspired by the human brain. However, since neural networks technique takes more time than other classifiers when training [60], it is considered difficult to apply to any malware detection system in which real time is a constraint. The Multi-Layer Perceptron (MLP) is a type of artificial neural network consisting of a network of neuron layers. It has been widely employed among researchers in various fields such as banking, defence, and electronics. MLP has medium-level complexity. In this work, one hidden three–node layer was chosen. This classifier is flexible and supports high degree of complexity, but it seems complex and hard to interpret.

- **AdaBoost** (AB) [28] is an iterative classifier that runs other algorithms multiple times in order to reduce the error. For the first iteration all of 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.

In this study I have used the algorithms and the parameters specified in Table 3.4.

Algorithm	Used configuration
Naïve Bayes (NB)	N/A
Decission Tree (J48)	N/A
Random Forest (RF)	Number of trees: 10
k-Nearest Neighbors (KNN)	K=1
Support Vector Machine (SVM)	Kernel: linear
Multi-Layer Perceptron (MLP)	Hidden layer:1, nodes in hidden layer: 3
AdaBoost (AB)	Base classifier: SimpleLogistic

Table 3.4: Machine Learning classifiers used in the experiments.

3.4 Evaluation Phase

In order to evaluate each algorithm, three *validation methods* were chosen. These methods are known as 10% split cross–validation, 33% split cross–validation, and 5–fold cross–validation.

The first method, 10% split cross–validation, is defined as using 10% of a data set for training purposes and remaining 90% for testing. Similarly, the second method, 33% split cross–validation, uses 33% of the data set for training purposes and the remainder for testing. Finally, as a case of k–fold cross–validation method, a 5–fold option was used, which is described as applying the classifier to data 5 times, and each time with a 80-20 configuration, i.e. 80% of data for training and 20% for testing; the final model is the average of these 5 iterations.

The benefit of split methods is that they take much less time as compared to the k-fold method, since the process is carried out only once, whereas the same process is to be done k times for the k–fold method. *Over–fitting* is a drawback of the 33% split method, and it occurs when a classifier *memorizes* the training set instead of getting trained. In the majority of the experiments, the k-fold method generally produces better results than the split methods, which can be understood as a sign of over–fitting..

The WEKA software has been preferred for this study, due to its simplicity and user-friendly interface. More specifically, the algorithms used in WEKA are displayed in Table 3.4. In those cases in which no configuration parameters are specified, the configuration used was the default.

In order to evaluate the performance of algorithms, I have chosen the following standard *metrics*:

- **True Positive Rate** (TPR), which is the proportion of correctly classified instances.
- **False Positive Rate** (FPR) is the proportion of incorrectly classified instances.
- **Precision**, which is the number of true positives divided by the total number of elements labelled as belonging to the positive class.
- **Area under Curve** (AUC) provides the relation between false negatives and false positives.

Different algorithms were trained with three alternative data sets (Table 3.3): the full data set , the PCA data set with a linear combination of 5 parameters per Principal Component

(denoted PCA5 data set), and the most compressed data set (namely PCA3 data set) obtained after applying PCA to generte a linear combination of 3 parameters per PC.

To sum up: I have built *three data sets* (Full, PCA5, and PCA3), then I have chosen *seven Machine Learning algorithms* (NB, J48, RF , KNN, SVM, MLP, and AB), I have selected *four metrics* (TPR, FPR, Precision, and AUC), and *three validation methods* (10% split, 33% split, and 5-fold cross-validation). All these "ingredients" have been combined together to perform a series of experiments in the WEKA environment (Fig. 3.7). The results of these experiments will be shown, analyzed and interpreted in next chapter.

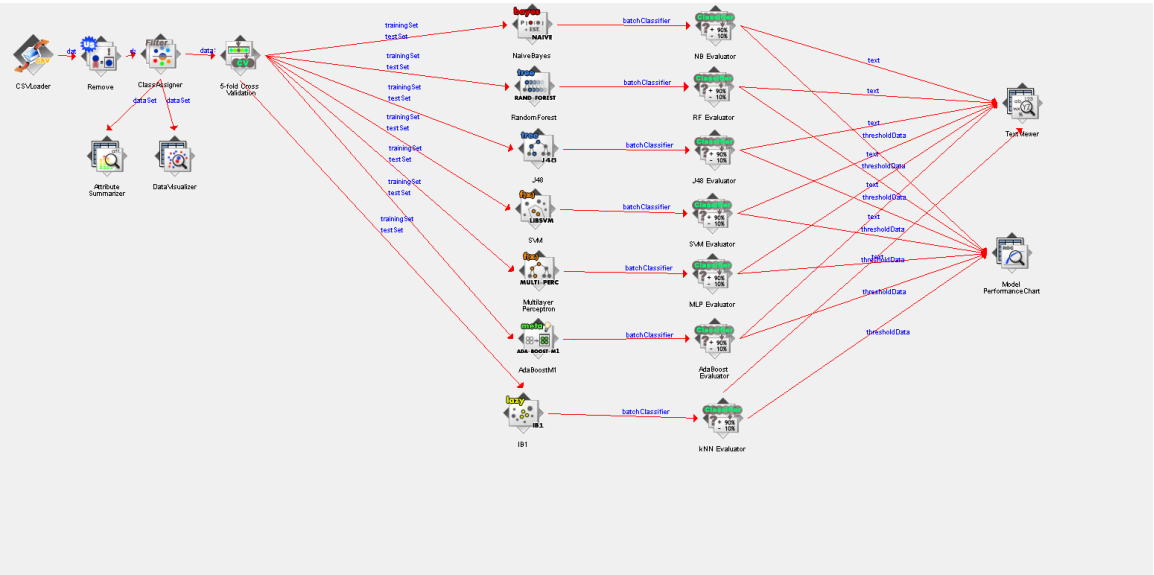


Figure 3.7: Example of an experiment in WEKA

"We're made up of thousands of parts with thousands of functions all working in tandem to keep us alive. Yet if only one part of our imperfect machine fails, life fails. It makes one realize how fragile... how flawed we are..."

Ingun Black–Briar, in *"The Elder Scrolls V: Skyrim"*

4
Experimentation Results

As I have already stated, the present work addresses a study and evaluation of various Machine Learning classifiers for Android malware detection in order to face the problem of its rapid growth.

MaLDroide experiment results are presented and interpreted in this chapter.

The empirical results of experiments are presented in Table 4.1. Details concerning the experiments have already been described in the preceding chapter.

In view of the results summarized in Table 4.1, one may conclude that all these classifiers show a high effectiveness rate. But we cannot see the wood for the trees!.

A thorough examination of these results is required. For instance, one might be led to conclude that a 96% true positive rate (which is the lowest value) reached by Naïve Bayes algorithm is an excellent output; however, one should have in mind that this implies that 105 cases have been misclassified (104 are malware misclassified as benign, and 1 benign sample has been misclassified as malware). Accordingly, apparently *negligible variations in FPR involve hundreds of cases that become misclassified*, which is not acceptable in this context.

I would like to put special emphasis on the fact that the dimensionality reduction process applied to the full data set by **PCA does not entail a significant loss in the different performance measures**. This circumstance becomes clear by examining and comparing the successive rows in Table 4.1. Taking this fact into account, as well as the results shown in Table 4.2, the said dimensionality reduction has been worth.

		10% split				33% split				5-fold cross-validation			
Data Set	Algorithm	TPR	FPR	Prec.	AUC	TPR	FPR	Prec.	AUC	TPR	FPR	Prec.	AUC
	NB	0.995	0.009	0.995	0.997	0.995	0.021	0.995	0.995	0.996	0.010	0.996	0.997
	J48	0.996	0.021	0.996	0.987	0.997	0.018	0.997	0.99	0.999	0.004	0.999	0.999
	RF	0.996	0.017	0.996	1.0	0.999	0.006	0.999	1.0	0.999	0.004	0.999	1.0
Full	**KNN**	0.998	0.011	0.998	0.994	0.999	0.006	0.999	0.997	1.0	0.0	1.0	1.0
	SVM	0.997	0.013	0.997	0.992	0.999	0.006	0.999	0.997	1.0	0.0	1.0	1.0
	MLP	0.998	0.013	0.998	1.0	0.999	0.006	0.999	1.0	1.0	0.0	1.0	1.0
	AB	0.996	0.021	0.996	0.987	0.997	0.018	0.997	0.99	1.0	0.0	1.0	1.0
	NB	0.988	0.006	0.989	0.993	0.981	0.006	0.983	0.99	0.96	0.01	0.968	0.981
	J48	0.995	0.017	0.995	0.989	0.994	0.018	0.994	0.988	0.998	0.006	0.968	0.995
	RF	0.989	0.047	0.989	1.0	0.996	0.015	0.996	1.0	0.999	0.006	0.999	1.0
PCA5	**KNN**	0.993	0.034	0.994	0.98	0.996	0.024	0.996	0.986	0.999	0.004	0.999	0.998
	SVM	0.998	0.011	0.998	0.994	0.999	0.006	0.999	0.997	1.0	0.002	1.0	0.999
	MLP	0.996	0.019	0.996	0.995	0.999	0.006	0.999	0.993	0.999	0.004	0.999	0.995
	AB	0.995	0.025	0.995	0.993	0.994	0.032	0.994	0.991	0.999	0.004	0.999	1.0
	NB	0.995	0.007	0.995	0.992	0.986	0.005	0.987	0.992	0.962	0.009	0.969	0.986
	J48	0.995	0.017	0.995	0.989	0.994	0.018	0.994	0.988	0.998	0.006	0.998	0.995
	RF	0.995	0.027	0.995	1.0	0.997	0.015	0.997	1.0	0.999	0.004	0.999	1.0
PCA3	**KNN**	0.998	0.011	0.998	0.994	0.999	0.006	0.999	0.997	0.999	0.004	0.999	0.998
	SVM	0.998	0.011	0.998	0.994	0.999	0.006	0.999	0.997	1.0	0.002	1.0	0.999
	MLP	0.996	0.021	0.996	0.995	0.999	0.006	0.999	0.993	0.999	0.004	0.999	0.995
	AB	0.995	0.025	0.995	0.993	0.994	0.032	0.994	0.991	0.999	0.004	0.999	0.998

Table 4.1: The performance of classifiers on malware detection

For my purposes, the most significant part of Table 4.1 is left–bottom block (corresponding to experiments conducted with PCA3 data set and 10%–split cross-validation); in this block the performance measures under the most restrictive conditions have been collected. Consequently, this block contains the results to which special attention must be paid. Important as they are also, the remaining blocks in Table 4.1 provide useful information for comparison tasks.

AUC values confirm that the Machine Learning techniques under consideration provide compelling results regarding malicious application detection in Android.

As expected, the values obtained in the 5-fold cross–validation reveal rates that are generally higher than those provided by the other validation methods, as can be inferred from the 5-fold cross–validation columns in Table 4.1. This fact points out that a certain **risk of over–fitting** in training process can have occurred.

Apart from considering the effectiveness of the algorithms, I am also interested in their computational performance, since both on-device and conventional computer executions become a fundamental parameter to be considered. To this end, I will compare the execution times in the course of training and test processes. These execution times can be found in Table 4.2.

	Algorithm						
Data Set	**NB**	**J48**	**RT**	**KNN**	**SVM**	**MLP**	**AB**
Full	0.11 / 0.11	0.17 / 0.00	0.11 / 0.00	0.01 / 4.79	12.64 / 0.17	34.40 / 0.05	13.56 / 0.00
PCA5	0.02 / 0.01	0.18 / 0.00	0.29 / 0.00	0.00 / 0.67	0.43 / 0.04	5.51 / 0.00	3.20 / 0.00
PCA3	0.01 / 0.01	0.13 / 0.00	0.29 / 0.00	0.00 / 0.57	0.28 / 0.03	3.81 / 0.00	2.46 / 0.00

Table 4.2: Elapsed time (in seconds) for training and testing.

To start with, it is seen in Table 4.2 that the execution time dramatically decreases when using reduced data set (PCA5 and PCA3), as compared to the full data set. This is especially evident in the cases of the most time-consuming algorithms: Multi–Layer Perceptron (MLP), AdaBoost (AB), and Support Vector Machine (SVM). The training time is reduced to one tenth (from 34.4 sec. to 3.81 sec. in the case of MLP).

Moreover, the case of KNN algorithm shows that training time is almost zero, while the test time becomes relatively high. This is due to the algorithm nature (it is called a "lazy algorithm"): in the training process, a lazy algorithm does not do much work, while in test stage KNN must calculate all the Euclidean distances for all of the cases. For this reason, this algorithm is not especially effective as an on-device classifier.

Decision Tree-based algorithms (say J48 and Random Forest) demonstrate their simplicity and test speed, their training and test times being almost the best ones.

As a consequence of the above considerations and results, I am now in a position to decide on a classifier for the framework of Android malware detection proposed in the present work.

From Table 4.1, we can observe that SVM algorithm has a slightly better classification performance than other methods. As the table illustrates, Support Vector Machines (SVM) classifier provides the best TPR value with 99.8%, which means nearly perfect prediction. Regarding TPR and FPR, k-Nearest Neighbor (KNN) clasiffier is tied with SVM. The remaining classifiers reach slighly lower rates.

Narrowly, SVM outperforms the other classifiers if the whole Table is taken into consideration. Other classifiers, such as KNN, MLP, and AB classifiers attain excellent rates in malware detection. Accordingly, any of them might be accepted as a good candidate for malware classifier in this framework.

The empirical results provide an initial roadmap to investigate selected Machine Learning algorithms for Android malware classification, and represent a resource for validating the future improvements of new malware classification approaches.

"Big things have small beginnings."

David, in *"Prometheus"*

5 Conclusions

This present work is devoted to the study and evaluation of Machine Learning-based techniques to effectively detect Android malware by selecting and collecting appropriate features. In the light of several measure values, an "ideal" classifier is proposed.

5.1 Contributions and Conclusions

This trained classifier is an essential part of the framework designed in this work. This framework has been named **MaLDroide** (**Ma**chine **L**earning–based **D**etector for And**roid** malwar**e**).

The major contributions and conclusions in this work are as follows:

- In this study I have use 2295 samples of recent (as in November 2014), real–world malicious applications extracted from the Android Malware Genome Project, the DREBIN Project, and contagio mobile dump. For comparison purposes, I have resorted to 436 cases of recent and free, real-world legitimate applications borrowed from Google Play Store.
- I have devised an method for selecting and extracting 299 features that allow me to characterize the above applications according to the main criteria related to security and malicious activity in Android handheld system.
- After a dimensionality reduction stage carried out by means of Principal Component Analysis (PCA), the initial 299 features have been compressed to 27 features, almost

completely preserving their original global variance. In so doing, a reduced data set has been determined.

- The best known and studied Machine Learning algorithms have been trained and evaluated in their performance over the chosen data set. Those algorithms are Naïve Bayes (NB), Decision Tree (J48), Random Forest (RF), k-Nearest Neighbor (KNN), Support Vector Machine (SVM), Multi-Layer Perceptron (MLP), and AdaBoost (AB).
- Four standard performance measures have determined a ranking of the aforesaid algorithms.

The final product resulting from this study is a framework for Android malicious application detection consisting of

- A procedure for features extraction from applications based on static analysis of Android manifest and source code,
- a dimensionality reduction mechanism for compressing the number of features without considerable accuracy loss,
- a Machine Learning algorithm, Support Vector Machine (SVM), already configured, trained and adjusted to be applied.

This framework leads to 99.8% accuracy, which means that it is capable to discriminate almost all the cases of malware existing in the considered data set.

5.2 Limitations

As I have demonstrated in Chapter 4, MaLDroide shows a high efficacy in detecting recent real-world malware on the Android platform. However, as any tool, MaLDroide can potentially limit its effectiveness as a framework for malware detection. For instance,

- MaLDroide's detection performance critically depends on the availability of representative malicious and benign applications.
- Machine Learning techniques can help us to recognize similar and quasi-clone malware but they cannot identify completely new brand malware.

- As a purely static method, MaLDroide suffers from the inherent limitations of static code analysis. Some attacks based on reflection and encryption become undetectable by MaLDroide. MaLDroide may also fail to detect malware that uses updating technique as attack vector, because only the main package is analyzed.

- MaLDroide depends upon tools such as Androguard. Thus, deficiencies and limitations in this tool may also manifest in MalDroide.

5.3 Future Work

This work simply represents the first step in a longer journey towards developing a practical Android handheld malware detection system.

I have in mind to develop an efficient and lightweight implementation of MaLDroide that can be embedded into an Android handheld device for real-time detection. Furthermore, the framework presented here may potentially be applied to develop a cloud-based malware detection system [83, 84].

Future studies will intend to face hardly detectable malware. Given that diverse variants and new types of malware are arising, additional research on Machine Learning techniques should be considered in order to detect future malware.

"A man chooses. A slave obey. OBEY."
Andrew Ryan, in *"Bioshock"*

A Android Platform

Andriod platform is an open source mobile operating system for mobile phones. However, it can be better described as a middleware running on top of embedded Linux. Each application is written in Java (possibly accompanied by native code), runs in its own virtual machine, and runs in a process with a low-privilege user ID. This design choice minimizes the effects of buffer overflows. Applications can access only to their own files by default

There are four layers in the Android stack (Fig. A.1):

- **Android Application Layer**,
- **Application framework**,
- **Android runtime**,
- **Linux kernel**, that acts as an abstraction layer between software stack and the hardware of device.

Android applications are developed using Java programming language along with the tools and APIs provided by the Android Software Development Kit.

A.1 Components of the Android Application

Android applications are composed of several standard components which are responsible for different parts of the application functionality. Android defines four types of application components:

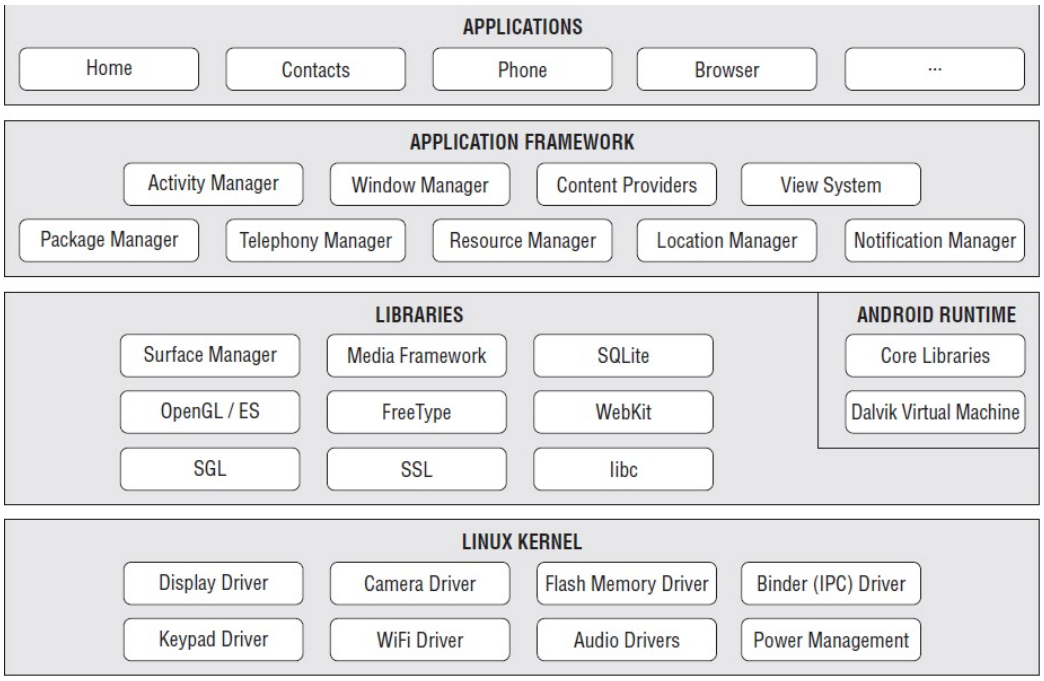


Figure A.1: Android Architecture

- **Activities** provide user interfaces. Activities are started with *Intents* , and they can return data to their invoking components upon completion. All visible portions of applications are Activities. The activity which is started at the application launch time is called the *main activity*. The life cycle of an activity includes several states. It begins from `onCreate()` and ends at the time when `onDestroy()` is called. After an activity has been created, `onStart()` is the point at which the activity becomes visible to users.

- **Services**, which are background processes for functionality not directly tied to the user interface. Services work quite similarly to activities, the only difference is that services usually perform a long term task. Services can be started in two different ways. Calling `startService()` allows us to run an independent task, and the service quits automatically when the task is finished. The other way to start a service is through application bindings, and thus the application has to decide when the service is activated and when it is killed.

- **Content Providers** are databases addressable by their application-defined URIs. The data in content providers can be shared across applications, but only when the access is allowed.

- **Broadcast Receivers**, which passively receive messages from the Android application

framework. Registering a broadcast receiver lets our application listen to a particular state of either the system or other applications.

A.2 Android Application Structure

In Android, applications are usually written in Java (less than 5% have native C components [102]), and are distributed as *apk* (Android package) files. These apk files are in fact zip files which contain everything that the application needs to run, including compiled Java classes (in Dalvik[1] DEX format[2]), icons, XML files specifying the user interface (UI), application data, and an AndroidManifest.xml binary XML file containing application meta-data.

The structure of Android applications and the Android security mechanisms have been well–documented [79] and many tools exist for creating and manipulating apks.

Typically, Android applications that have a user interface specify at least one Android Activity, whereas those that do not have a user interface specify at least one Service. These are classes that typically contain the core functionality of the mobile application, and are the primary method for executing application code.

A.2.1 Android Manifest

The AndroidManifest.xml file is the configuration file of the Android applications. The manifest file informs the Android framework of the application components and how to route intents between components. It also declares the specific screen sizes handled, available hardware and, most importantly for this work, the application's required permissions.

```
<manifest>
    <uses-permission />
    <permission />
    <permission-tree />
    <uses-sdk />
    ....
</manifest>
```

Figure A.2: Example of Android Manifest file.

[1] An alternative runtime environment called Android Runtime (ART) was included in Android 4.4. ART will replace Dalvik entirely in Android 5.0.

[2] The DEX bytecode format is independent of device architecture and needs to be translated into native machine code to run on the device. The most significant change from Dalvik to ART is that Dalvik is based on Just-in-Time (JIT) compilation, while ART is based on Ahead-of-Time (AOT) compilation.

A.3 The Android Security Model

The Android system uses several methods to secure the devices [23]. In next sections, the security features that affect applications directly will be described.

A.3.1 Permissions

Android security model highly relies on permission–based mechanisms. The Android platform has about 130 application permissions with four protection levels that govern access to resources. The protection level is a parameter of a permission and needs to be specified when defining our own permissions. Each level of protection enforces a different security policy. From weak to strong, we have following levels:

Normal permission: includes lower–risk permissions which control access to API calls that are not particularly harmful. The system automatically grants this type of permission to a requesting application at installation, without asking for the user's explicit approval. For example, `VIBRATE` and `SET_WALLPAPER` are permissions that are not considered to have any danger associated with them.

Dangerous permission: regulates access to potential harmful API calls that would give access to private user data. The dangerous protection level will cause warnings to be displayed to the user before installation, and requires the user's explicit approval to be granted. For example, permissions to read the location of a user `ACCESS_FINE_LOCATION` or `WRITE_CONTACTS` are classified as dangerous.

Signature permission: only granted if the requesting application is signed by the same certificate as the certificate that was used to sign the application that defined the permission.

SignatureOrSystem permission: only granted to applications installed in the system applications folder (a part of the system image) or that the application is signed by the same certificate as the one used to sign the version of the Android system installed on the device. Applications from the Android Market cannot be installed into the system applications folder. System applications must be pre–installed by the device manufacturer or manually installed by an advanced user.

A small number of permissions are enforced by Unix groups, rather than the Android permission validation mechanism. In particular, when an application is installed with the

INTERNET, WRITE_EXTERNAL_STORAGE, or BLUETOOTH permissions, it is assigned to a Linux group that has access to the pertinent sockets and files.

Android uses a static permission approach for its security, all permissions are requested and granted at install-time. Once being granted, they cannot be changed and will be valid throughout the lifetime of this application.

A.3.2 Application Programming Interface (API)

The public API describes 8,648 methods[3], some of which are protected by permissions. There is no centralized policy for checking permissions when an API is called.

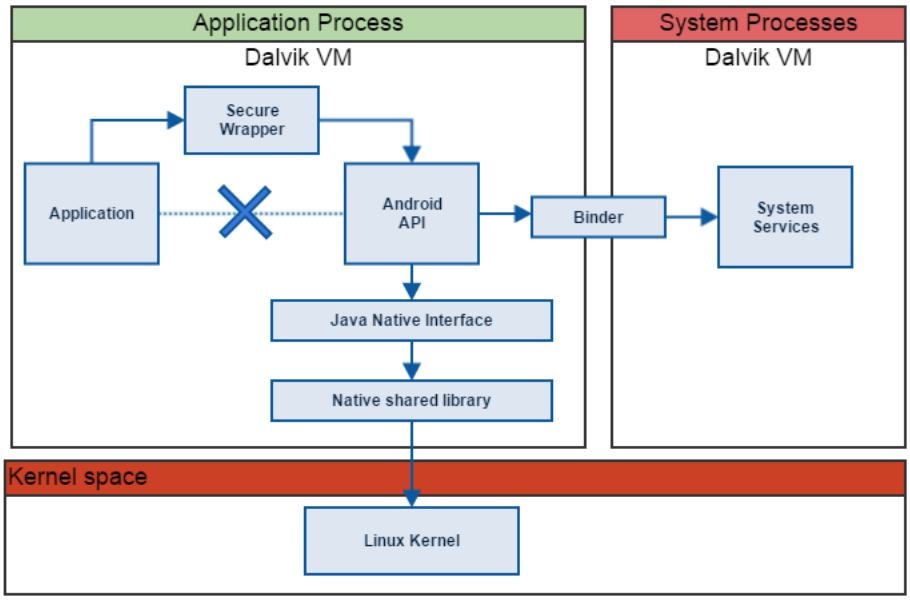


Figure A.3: Android API Architecture

The Android API framework is composed of two parts: a library that resides in each application's virtual machine and an implementation of the API that runs in the system process. The API library runs with the same permissions as the application it accompanies, whereas the API implementation in the system process has no restrictions. API calls are handled in three steps (Fig. A.3):

1. the application invokes the public API in the library,

2. he library invokes a private interface (an RPC stub) also in the library,

[3]Android developers reference:http://developer.android.com/reference/

3. the RPC stub initiates an RPC request with the system process that asks a system service to perform the desired operation.

Permission checks are placed in the API implementation in the system process. When necessary, the API implementation calls the permission validation mechanism to check that the invoking application has the necessary permissions. Permission checks therefore should not occur in the API library. Instead, the API implementation in the system process should invoke the permission validation mechanism.

An application can use Java Reflection to access all of the API library's hidden and private classes and methods [57]. Some private interfaces do not have any corresponding public API; anyway, applications can still invoke them using reflection. Java code running in the system process is in a separate virtual machine and therefore immune to reflection.

A.3.3 Content Providers

Content providers are used to share data between multiple applications. For example, contact information is stored in a content provider so that other applications can query it when necessary.

System Content Providers are installed as standalone applications, separate from the system process and API library.

User data are stored in Content Providers, and permissions are required for operations on some system Content Providers. For example, applications must hold the `READ_CONTACTS` permission in order to execute read queries on the Contacts Content Provider.

A.3.4 Intents

In order to communicate and coordinate between components, Android provides a message routing system based on URIs. Activities, services and broadcast receivers are activated by an asynchronous message called an intent. An *Intent* is a message that declares a recipient and optionally includes data. Intents notify applications of events, such as a change in network connectivity. Additionally, some Intents sent by the system are delivered only to applications with appropriate permissions. Furthermore, to prevent applications from mimicking system Intents, Android restricts who may send certain Intents. All Intents are sent through the ActivityManagerService (a system service), which enforces this restriction.

For activities and services, intents define an action that should be performed (start or stop). Some intents may include additional data that specify what to act on. For broadcast receivers, the intent simply defines the current announcement that is being broadcast. For

example, for an incoming SMS text message, the additional data field will contain the content of the message and the sender's phone number.

To Receiver	`sendBroadcast(Intent i)`
	`sendBroadcast(Intent i, String rcvrPermission)`
	`sendOrderedBroadcast(Intent i, ...)`
	`sendOrderedBroadcast(Intent i, String recvrPermission)`
	`sendStickyBroadcast(Intent i)`
	`sendStickyOrderedBroadcast(Intent i, ...)`

To Activity	`startActivity(Intent i)` `startActivityForResult(Intent i, int requestCode)`
To Service	`startService(Intent i)` `bindService(Intent i, ServiceConnection conn, int flags)`

Table A.1: A non-exhaustive list of Intent-sending mechanisms

Intents can include URIs that reference data stored in an application's Content Provider. If the Provider has allowed URI permissions to be granted (in the manifest), this will give the Intent recipient the capacity to read or write the data at the content. If a malicious code intercepts the Intent, it can access the data URI contained in the Intent.

A malicious application can launch an Intent spoofing attack by sending an Intent to an exported component that is not expecting Intents from that application. If the victim application takes some action upon receipt of such an Intent, the attack can trigger that action.

"You have to learn the rules of the game. And then you have to play better than anyone else."

Albert Einstein, Physician.

B

An Brief Overview of Machine Learning

In 1959, Arthur Samuel defined **Machine Learning** as a field of study that gives computers the ability to learn without being explicitly programmed. This is an informal definition, but also an old one. A slightly more recent definition was proposed by Tom Mitchell [59], who states that a computer program is said to **learn from experience** E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

Machine Learning algorithms can commonly be divided into three different types depending on the training data:

- supervised learning,
- unsupervised learning,
- semi-supervised learning.

For *supervised* learning algorithms, the training data set must be labelled (e.g., malware and benign). For *unsupervised* learning algorithms, data do not need to be labelled and try to determine how data are organised into different groups called *clusters*. Finally, *semi-supervised* algorithms use a mixture of both labelled and unlabelled data in order to build models, improving the accuracy of solely unsupervised methods.

B.1 Classifiers

Classification is a fundamental issue in Machine Learning and Data Mining. In classification, the goal of a learning algorithm is to construct a classifier given a set of training examples with class labels.

A case E is represented by a tuple of attribute values $(x_1, x_2, \ldots, x_n)$, where x_i is the value of attribute X_i. Let C represent the classification variable, and let c be the value of C. In this work, there are only two classes: MALWARE (the *positive* class) or BENIGN (the *negative* class) is assumed.

A **classifier** is a function that assigns a class label to a case.

B.1.1 Bayesian Classifiers

Bayesian Networks (BN) are defined as probabilistic models based on the Bayes Theorem for multivariable analysis. BN are *directed acyclic graphs* (DAG) that have an associated probability distribution function: *nodes* represent problem variables, and the e*edges* represent conditional dependencies between such variables.

The most important capability of Bayesian Networks is their ability to determine the probability that a certain hypothesis is true (e.g., the probability of an app to be malware) given a historical data set.

According to Bayes Rule, the probability of a sample $S = (x_1, x_2, \ldots, x_n)$ being class c is $p(c|S) = \frac{p(S|c)p(c)}{p(E)}$. Assuming that all attributes are independent, given the value of the class variable, $p(S|c) = p(x_1, x_2, \ldots, x_n|c) = \prod_{i=1}^{n} p(x_i|c)$.

B.1.1.1 Naïve Bayes

Naïve Bayes (NB) is the simplest form of Bayesian Network wherein given a class variable, all attributes are assumed to be independent [98].

The algorithm is able to classify by calculating the maximum likelihood of the attributes belonging to a certain class. Even with the interaction of certain attributes, the Naïve Bayes assumption does not lose predictive accuracy even if the actual probabilities are different.

In Naïve Bayes, the acyclic graph becomes the simpliest possible one, and each attribute node has no parent except the class node.

Naïve Bayes classifiers have high accuracy on the data sets in which strong dependencies exist among attributes. Surprisingly it makes an assumption that is almost always violated in real applications: given the class value, all attributes are independent. The reason is that NB does not penalize inaccurate probability estimation as long as the maximum probability is assigned to the correct class [19].

B.1.2 Decision Trees

Decision Tree (DT) classifiers are a type of Machine Learning classifiers that can be graphically represented as trees [78, 40]. A decision tree consists of three building blocks: internal nodes, edges and leaves. *Internal nodes* represent conditions regarding the variables of a problem, whereas *leaves nodes* represent the ultimate decision of the algorithm.

Different training methods are typically used for learning the graph structure of these models from a labelled data set; special attention will be paid here to the C4.5 algorithm and the Random Forest algorithm.

B.1.2.1 C4.5

C4.5 [69, 70] belongs to a succession of decision tree learners. Although C4.5 has been superseded by C5.0, a commercial system from RuleQuest Research, this description will focus on C4.5 since its source code is readily available. As a particular case, J48 is an open source Java-based implementation of the C4.5 Decision Tree algorithm present in WEKA [93].

C4.5 adopts a greedy approach in which decision trees are constructed in a top–down recursive divide–and–conquer manner. Most algorithms for decision tree induction also follow a top–down approach, which starts with a training set and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built. By selecting the attributes according to the gain ratios criterion, C4.5 builds up a decision tree where each path from the root to a leaf represents a specific classification rule.

B.1.2.2 Random Forest

Collection of trees are called forest, and **Random Forest** (RF) [43, 11] uses many classification trees to be able to classify a class based on the majority vote of classification generated by the trees.

The algorithm generates K trees independently, which makes it very easily parallelizable. For each tree, RF algorithm constructs a full binary tree of a given depth. The features used in each tree are selected randomly (meaning that the same feature can appear multiple times). The resulting classifier is a *voting* of the K random trees.

This algorithm tends to work best when all the features are relevant, since the features selected for any given tree are small.

B.1.3 k-Nearest Neighbor

The **k-Nearest Neighbor** algorithm is based on learning by analogy, that is, by comparing a given test example with training examples that are similar to it. Each training example represents a point in an n-dimensional space.

The k-Nearest Neighbor algorithm is amongst the simplest of all Machine Learning algorithms: an example is classified by a majority vote of its neighbors, with the example being assigned to the most common class amongst its k nearest neighbors (where k is a positive integer, typically small). If $k = 1$, then the example is simply assigned to the class of its nearest neighbor. *Closeness* is defined in terms of a distance metric, such as the Euclidean distance.

At training time, k-Nearest Neighbor algorithm simply stores the entire training set. At test time, to predict a class, k-Nearest Neighbor algorithm finds the training example that is the most similar one. In particular, the algorithm finds the training example that minimizes the Euclidean distance between both cases. The algorithm predicts the same class for the k-nearest training cases.

Despite its simplicity, k-Nearest Neighbor classifiers are astonishingly effective.

B.1.4 Support Vector Machines

Support Vector Machine (SVM) [15] is a set of *supervised* learning method used for classification, regression and outlier detection.

SVM is effective in high dimensional spaces and also when the number of dimensions is greater than the number of samples. SVM is very memory efficient because it uses a subset of training points (formed by the so–called *support vectors*) in the decision function. In general, the number of support vectors is far smaller than the number of training samples.

Given a training set, SVM tries to find a **decision boundary** that maximizes the (geometric) margin. This will result in a classifier that separates the positive and the negative training examples by means of a "gap" (*geometric margin*). We will assume that we are given a training set that is *linearly separable*[1]. This is a way of setting up an *optimization problem* that attempts to find a separating hyperplane with as large a margin as possible.

Let D be some training data, that is, a set of n points of the form

$$D = \left\{ \left(x^{(i)}, y^{(i)} \right) \mid x^{(i)} \in \mathbb{R}^m,\, y^{(i)} \in \{-1, 1\} \right\}_{i=1}^{n} ,$$

where the y_i are either -1 or 1, indicating the class to which the point x_i belongs, and each $x^{(i)}$ is an m-dimensional vector of real numbers.

[1] It is possible to separate the positive and negative examples using some separating hyperplane.

In this optimization approach, SVM algorithm is trying to find parameters that *maximize the margin*, denoted γ, subject to the *constraint* that all training examples are correctly classified. This can be formulated as a **constrained optimization problem**:

$$\min_{w,b} \frac{1}{\gamma(w,b)}, \tag{B.1}$$

$$\text{such that} \quad y^{(i)}(wx^{(i)} + b) \geq 1, \forall i = 1, \ldots, n. \tag{B.2}$$

The goal of this objective function is to ensure that all points are correctly classified. The difficulty with the optimization problem is what happens with data that are not linearly separable. Into the bargain, a *"penalty" constraint* is added if a point cannot be correctly classified.

The original algorithm was modified [15] to introduce *nonlinear classifiers* by applying a kernel function. This allows the algorithm to fit the maximum-margin hyperplane in a transformed feature space.

Different *kernel functions* can be specified for the decision function. The basic idea is to map the given data into some feature space, using some nonlinear mapping, and then to learn a linear classifier in the new space. A *kernel function* is a symmetric function K that computes a dot product in some space:

$$K(x, z) = \phi(x)^T \phi(z)\,.$$

Some examples of kernel functions can be adduced:

- Linear kernel $K(x, z) = x^T z$
- Polynomial kernel $K(x, z) = \left(1 + x^T z\right)^p$, where $p = 2, 3, \ldots$. It encompasses all polynomial terms up to degree p.
- Radial basis kernel $K(x, z) = e^{-\frac{1}{2}x-z^2}$.
- Gaussian kernel $K(x, z) = e^{(-\frac{x-z^2}{2\sigma^2})}$.

B.1.5 Neural Networks

A **Neural Network** is a fine–grained, parallel, distributed computing model characterized by:

- a large number of very simple, neuron-like processing elements called *units*, or *nodes*,

- a large number of weighted (positive or negative real values), directed connections between pairs of units,
- local processing in that each unit computes a function based on the outputs of a limited number of other units in the network,
- each unit computes a simple function of its input values, which are the *weighted outputs* from other units. If there are n inputs to a unit, then the *unit's output*, or *activation*, is defined by $a_i^{(k)} = g\left(\sum_{j=0}^{n} \Theta_{i,j}^{(k)} x_j\right)$, where $a_i^{(k)}$ is the activation of unit i in layer k, and $\Theta^{(k)}$ is a matrix of weights controlling function mapping from layer k to $k+1$. Thus each unit computes a (simple) function g of a linear combination of its inputs[2].

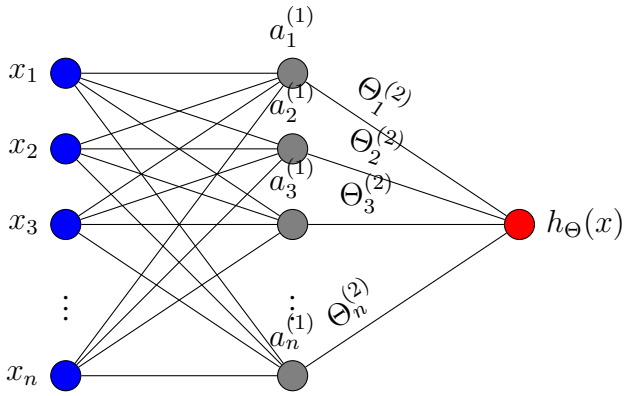


Figure B.1: Artificial Neural Network with 1 hidden layer

Multi-Layer Perceptron (MLP) [63] is an artificial neural network model that consists of multiple layers of nodes that interact via weighted connections (Fig. B.1).

B.1.5.1 Backpropagation Algorithm

The neural networks can be consider as an acyclic directed graph of units based on backpropagation algorithm. Intuitively, the backpropagation algorithm calculates the error of node j in layer l, denoted by $\delta_j^{(l)}$. The activation a_j^l of node j in layer l is a totally calculated value, so we expect there to be some error compared to the "real" value, and the δ term captures this error. The only "real" value that we have available is our actual classification y, so let us first calculate $\delta^{(L)} = a^{(L)} - y = h_\Theta(x) - y$. With $\delta^{(L)}$ calculated, we can determine the error terms for the other layers as follows: $\delta^{(L-1)} = \left(\Theta^{(L-1)}\right)^T \delta^{(L)}\, g\left(z^{(L-1)}\right)$.

[2]Originally the *neuron output function* g in original model was proposed as threshold function. However linear, ramp and sigmoid are also widely used output functions.

Algorithm 1 Back Propagation Algorithm

Given training data $\left(x^{(i)}; y^{(i)}\right), \quad i = 1, \ldots, m.$

Initialize $\Delta_{ij}^{(l)} = 0$ for all $i, j, l.$

for $i = 1, \ldots, m$ **do**

Perform forward propagation to compute $a^{(l)}$ for each layer $(l = 1, 2, \ldots, L)$.

Calculate $\delta^{(L)} = a^{(L)} - y.$

Move back through the network from layer $L - 1$ down to layer 1 (**back propagation**).

Use Δ to accumulate the partial derivative terms $\left(\Delta_{ij}^{(l)} \leftarrow \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}\right).$

end for

Compute $D_{ij}^{(l)} = \dfrac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)}$ if $j \neq 0$, and $D_{ij}^{(l)} = \dfrac{1}{m}\Delta_{ij}^{(l)}$ otherwise.

Calculate the partial derivative for each parameter $\dfrac{\partial}{\partial\Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}.$

B.1.6 Boosting

Boosting is a method to improve the performance of classifiers [28], such as Decision Trees or Naïve Bayes. Boosting is a method for *combining multiple classifiers*. Studies have shown that ensemble methods often improve performance over single classifiers. Boosting produces a set of *weighted models* by iteratively learning a model from a weighted data set, evaluating it, and reweighting the data set based on the model's performance.

The underlying idea of *Boosting* [29, 58] is to combine simple "rules" to form an ensemble such that the performance of the single ensemble member is improved (*"boosted"*). Let $h_1, h_2, \ldots, h_T$ be a set of *hypotheses*, and consider the composite ensemble hypothesis

$$h(x) = \operatorname{sign}\left(\sum_{t=1}^{T} \alpha_t \, h_t(x)\right),$$

where each $h_t(x)$ is a classifier producing values ± 1, and α_t denotes the coefficient with which the member h_t is combined. Both α_t and the hypothesis h_t are to be learned within the boosting procedure.

At each iteration, the boosting procedure updates the weight of each sample such that the misclassified ones get more weighting for the next iteration. *Boosting hence focuses on the examples that are hard to classify*.

The **AdaBoost (Adaptive Boosting)** algorithm [41, 6, 93], the most commonly known boosting algorithm, is described in Algorithm 2.

Algorithm 2 AdaBoost Algorithm

Given training data $\left(x^{(i)}; y^{(i)}\right), \quad y^{(i)} \in \{-1, +1\}, \quad i = 1, \ldots, m$.
Initialize the weights assigned to each sample, $\alpha_i = \dfrac{1}{m}, \quad i = 1, \ldots, m$.
loop
  Apply learning algorithm to weighted data set and store resulting model.
  Compute error err of model on weighted data set and store error.
  if $err = 0$ or $err \geq 0.5$ **then**
    Terminate model generation.
  end if
  for $j = 1, \ldots, m$ **do**
    if $\left(x^{(j)}; y^{(j)}\right)$ classified correctly by model **then**
      Multiply weight of instance α_j by $-\log\left(err/(1 - err)\right)$.
    end if
    Normalize weight of all instances.
  end for
end loop

B.2 Evaluation of Classifiers

The simplest way of measuring the perfomance of a classifier is to count correct decissions.

B.2.1 Confusion Matrix and Related Measures

Table B.1 describes a **confusion matrix** for computing the evaluation indicators.

		Predicted class	
		Positive	Negative
Actual class	Positive	TP (true positive)	FN (false negative)
	Negative	FP (false positive)	TN (true negative)

Table B.1: Confusion matrix

The following entries are shown in a confusion matrix:

- **True Positives** (TP): positive instances predicted to be positive.
- **True Negatives** (TN): negative instances predicted to be negative.
- **False Positives** (FP): negative instances predicted to be positive.
- **False Negatives** (FN): positive instances predicted to be negative.

On the basis of the aforementioned confusion matrix, the following indicators are given by Eqs. (B.3):

$$TPR = \frac{TP}{TP + TN} \tag{B.3a}$$

$$FPR = \frac{FP}{FP + TN} \tag{B.3b}$$

$$Precision = \frac{TP}{FP + FP} \tag{B.3c}$$

$$Recall = \frac{TP}{TP + FN} \tag{B.3d}$$

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{B.3e}$$

$$F_1 - score = 2 \times \frac{Precision * Recall}{Precision + Recall} \tag{B.3f}$$

True positive rate (TPR) means the proportion of correctly identified normal applications. *False positive rate* (FPR) represents the proportion of malware-containing applications incorrectly identified as benign. *Precision* is an indicator representing an *error of the decision value*, which represents the proportion of correctly diagnosed normal applications. *Accuracy* is an indicator representing the system's accuracy, expressed in the proportion of correctly identified normal applications and malware, among the results. Finally, F_1–*score* is also called F–*measure* and means accuracy in the aspect of decision results, where an F_1-score reaches its best value at 1 and worst score at 0.

These measures focus only on the positive examples and predictions, although overall they capture some information about the rates and kinds of errors made. However, *neither of them captures any information about how well the model handles negative cases* [67].

B.2.2 ROC Curves

A **Receiver Operating Characteristic (ROC) curve** is a graphical plot that illustrates the performance measure of a binary classifier determined by the true-positive rate versus the false one. ROC is mainly used for visualizing, organizing and selecting the optimal classifier based on the varying performances of classification algorithms.

ROC curves allows us evaluating and comparing algorithms. ROC curves signify the tradeoff between false positive and true positive rates, which means that any increase in the true positive rate is accompanied by a decrease in the false positive rate.

Misclassification rate is often a poor criterion by means of which to assess the performance of classification rules. Because of these problems, one of the most frequently adopted

measures of performance for the binary classifiers is the *area under the ROC curve* (AUC) [39, 38]. The area below the ROC curve, known as AUC, is widely utilized for measuring classifier performance with the following defined levels:

- 1.0: perfect prediction,
- 0.9: excellent prediction,
- 0.8: good prediction,
- 0.7: mediocre prediction, and
- $\leq$ 0.6: poor prediction.

B.2.3 Over–fitting and Under–fitting

Roughly speaking, **under–fitting** is when you had the opportunity to learn something but did not do it. **Over–fitting** is when you pay too much attention to singularities of the training data, and are not able to generalize well. This often means that the model is fitting noise, rather than whatever it is supposed to have to fit.

B.2.4 Cross-validation

Cross-validation is a standard statistical method to estimate the generalization error of a predictive model. Cross-validation is a model validation method that divides data into two segments and one of them used to train the Machine Learning classifier and the other segment is used to test it.

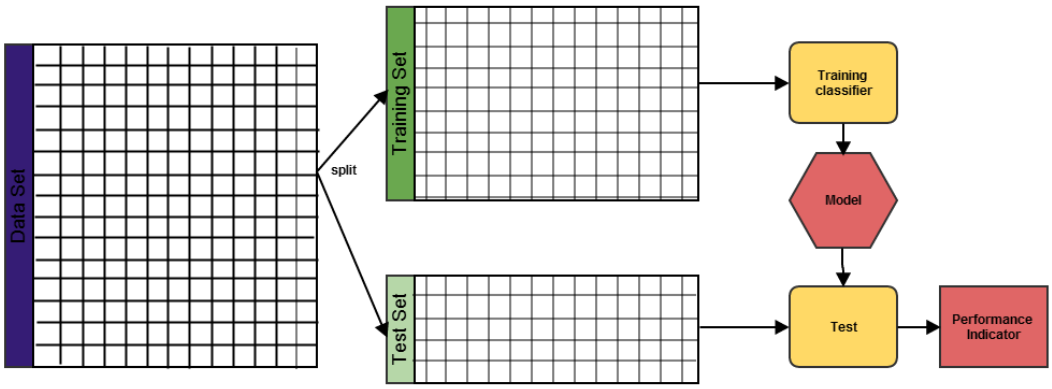


Figure B.2: Cross-validation using a training set and a test set.

B.2.4.1 k-fold cross-validation

In **k-fold cross-validation** a training set is divided into k equal-sized subsets. Then the following procedure is repeated for each subset: a model is built using the other $(k-1)$ subsets as the training set, and its performance is evaluated on the current subset. This means that each subset is used for testing exactly once. The result of the cross-validation is the average of the performances obtained from the k rounds.

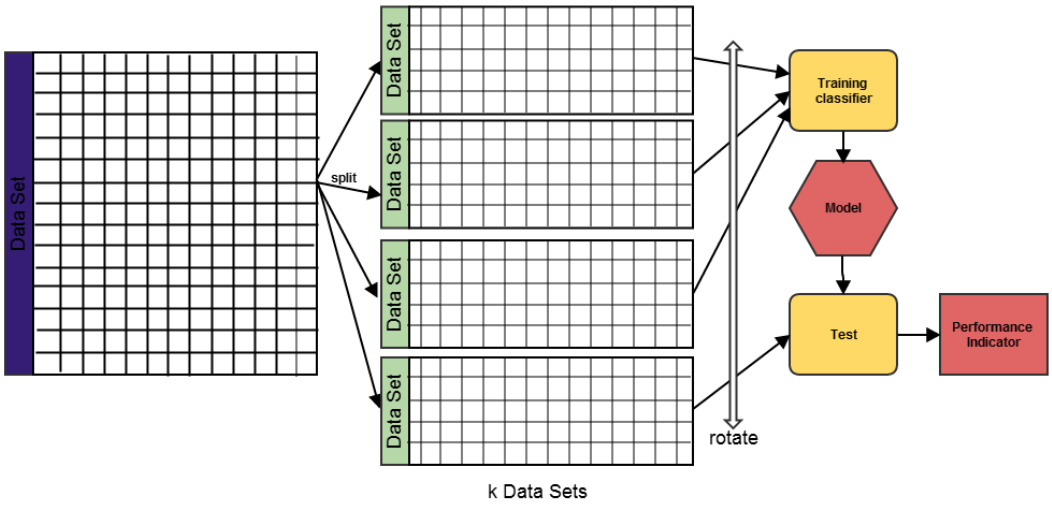


Figure B.3: k-fold cross-validation.

The *disadvantage* of this method is that the training phase has to be rerun from scratch k times, which means that it takes k times as many computations to perform an evaluation.

B.3 Feature Selection and Dimensionality Reduction

Dimensionality reduction tries to remove irrelevant, weakly relevant, and redundant attributes. Dimensionality reduction methods take an original data set and convert every instance from the original R^d space into an $R^{d'}$ space, where obviously $d' < d$.

Feature selection is considered successful if the dimensionality of the data is reduced and the accuracy of a learning algorithm improves or remains the same [35].

B.3.1 Principal Component Analysis (PCA)

Note that, strictly speaking, PCA is not a feature selection but *a feature extraction method*. The new attributes are obtained by means of a linear combination of the original attributes,

and dimensionality reduction is achieved by keeping the components with highest variance. Then, PCA tries to identify the components that characterize the data

Given a set of original variables $x_1, \ldots, x_n$ (the *attributes* of the problem), the PCA algorithm finds a set of vectors $z_1, \ldots, zp$ that are defined as linear combinations of the original variables and that are uncorrelated between each other; they are called the *principal components* (PC). Principal Components can be extracted using following steps:

- Each observation is subtracted from mean to produce a data set with zero mean values.
- Compute the covariance between dimensions. If the covariance is zero, it indicates that the dimensions are independent of each other.
- Compute eigen–vectors and eigen–values.
- Determine principal component and attributes corresponding to each principal component.

Algorithm 3 PCA algorithm

Given a data set X
$\mu = mean(X)$
$D = (X - \mu_1^T)^T (X - \mu_1^T)$ {compute covariance}
$\{\lambda_k, \mu_k\} \leftarrow$ top k eigen–values and eigen–vectors of D
return $(X - \mu_1)U$ {project data using U}

References

[1] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In *SecureComm*, volume 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 86–103. Springer, 2013.

[2] Brandon Amos, Hamilton A. Turner, and Jules White. Applying machine learning classifiers to dynamic android malware detection at scale. In *IWCMC*, pages 1666–1671, 2013.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. 2014.

[4] Zarni Aung and Win Zaw. Permission-based android malware detection, 2013.

[5] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil 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, pages 73–84, New York, NY, USA, 2010. ACM.

[6] C.M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[7] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Çamtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *MALWARE'10*, pages 55–62, 2010.

[8] Trond Boksasp and Eivind Utnes. Android apps and permissions: Security and privacy risks, 2012.

[9] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th International Conference on*

Mobile Systems, Applications, and Services, MobiSys '08, pages 225–238, New York, NY, USA, 2008. ACM.

[10] Tony Bradley. Huge spike in mobile malware targets android, especially mobile payments. `http://www.pcworld.com/article/2691668/report-huge-spike-in-mobile-malware-targets-android-especially-mobile-payments.html`, 2014.

[11] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[12] Iker Burguera, Urko Zurutuza, and Simin 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, pages 15–26, New York, NY, USA, 2011. ACM.

[13] G. Canfora, F. Mercaldo, and C.A. Visaggio. A classifier of malicious android applications. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 607–614, Sept 2013.

[14] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[15] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, September 1995.

[16] Bement Aberra Debelo, Wooguil Pak, and Young-June Choi. Sandroid: Simplistic permission based android malware detection and classification. In Roberto Saracco, Khaled Ben Letaief, Mario Gerla, Sergio Palazzo, and Luigi Atzori, editors, *IWCMC*, pages 1683–1687. IEEE, 2013.

[17] Anthony Desnos. Androguard. `https://code.google.com/p/androguard/`. Accessed: 2014-09-30.

[18] Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele 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: Computer Network Security*, MMM-ACNS'12, pages 240–253, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] Pedro Domingos and Michael Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, November 1997.

[20] Ken Dunham, Shane Hartman, Manu Quintans, Jose Andre Morales, and Tim Strazzere, editors. *Android Malware and Analysis*. Auerbach Publications, 2014.

[21] Karim O. Elish, Danfeng (Daphne) Yao, and Barbara G. Ryder. User-centric dependence analysis for identifying malicious mobile apps, 2013.

[22] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[23] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[24] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[25] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. Understanding Android Security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[26] Ali Feizollah, Nor Badrul Anuar, and Rosli Salleh. A Study Of Machine Learning Classifiers for Anomaly-Based Mobile Botnet Detection. *Malaysian Journal of Computer Science*, 26(4):251–265, 2013.

[27] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[28] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm, 1996.

[29] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28:2000, 1998.

[30] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park, November 2009.

[31] Dragos Gavrilut, Mihai Cimpoesu, Dan Anton, and Liviu Ciortuz. Malware detection using machine learning. In *IMCSIT*, pages 735–741. IEEE, 2009.

[32] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In Stefan Katzenbeisser, Edgar Weippl, L.Jean Camp, Melanie Volkamer, Mike Reiter, and Xinwen Zhang, editors, *Trust and Trustworthy Computing*, volume 7344 of *Lecture Notes in Computer Science*, pages 291–307. Springer Berlin Heidelberg, 2012.

[33] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[34] Martin T. Hagan, Howard B. Demuth, and Mark Beale. *Neural Network Design*. PWS Publishing Co., Boston, MA, USA, 1996.

[35] M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[36] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[37] Hyo-Sik Ham and Mi-Jung Choi. Analysis of Android malware detection performance using machine learning classifiers. In *ICT Convergence (ICTC), 2013 International Conference on*, pages 490–495, Oct 2013.

[38] David J. Hand. Measuring classifier performance: a coherent alternative to the area under the ROC curve. *Machine Learning*, 77(1):103–123, 2009.

[39] David J. Hand and Robert J. Till. A simple generalization of the area under the ROC curve for multiple class classification problems. *Machine Learning*, 45(2):171–186, 2001.

[40] Peter Harrington. *Machine Learning in Action*. Manning Publications Co., Greenwich, CT, USA, 2012.

[41] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[42] O. Henchiri and N. Japkowicz. A feature selection and evaluation scheme for computer virus detection. In *Data Mining, 2006. ICDM '06. Sixth International Conference on*, pages 891–895, Dec 2006.

[43] Tin Kam Ho. Random decision forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ICDAR '95, pages 278–, Washington, DC, USA, 1995. IEEE Computer Society.

[44] Chun-Ying Huang, Yi-Ting Tsai, and Chung-Han Hsu. Performance evaluation on permission-based detection for android malware. In Jeng-Shyang Pan, Ching-Nung Yang, and Chia-Chen Lin, editors, *Advances in Intelligent Systems and Applications - Volume 2*, volume 21 of *Smart Innovation, Systems and Technologies*, pages 111–120. Springer Berlin Heidelberg, 2013.

[45] Jeffrey O Kephart and William C Arnold. Automatic extraction of computer virus signatures. *4th virus bulletin international conference*, pages 178–184, 1994.

[46] Dong-uk Kim, Jeongtae KIm, and Sehun Kim. A malicious application detection framework using automatic feature extraction tool on android market. In *Proceedings of the 3rd International Conference on Computer Science and Information Technology*, ICCSIT 2013, 2013.

[47] Taehyun Kim, Yeongrak Choi, Seunghee Han, Jae Yoon Chung, Jonghwan Hyun, Jian Li, and J.W.-K. Hong. Monitoring and detecting abnormal behavior in mobile cloud infrastructure. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1303–1310, April 2012.

[48] Ron Kohavi. 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*, pages 202–207. AAAI Press, 1996.

[49] Jeremy Z. Kolter and Marcus A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the Tenth ACM SIGKDD International Conference on*

Knowledge Discovery and Data Mining, KDD '04, pages 470–478, New York, NY, USA, 2004. ACM.

[50] S. B. Kotsiantis. Supervised Machine Learning: A Review of Classification Techniques. In *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, pages 3–24, Amsterdam, The Netherlands, The Netherlands, 2007. IOS Press.

[51] P. Lantz, A. Desnos, and K. Yang. Droidbox: Android application sandbox. `https://code.google.com/p/droidbox/`.

[52] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Review: Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, January 2013.

[53] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[54] Wen Liu. Mutiple classifier system based android malware detection. In *ICMLC*, pages 57–62, 2013.

[55] Xing Liu and Jiqiang Liu. A Two-Layered Permission-Based Android Malware Detection Scheme. In *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on*, pages 142–148, April 2014.

[56] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA, 2013. ACM.

[57] Glen McCluskey. Using Java Reflection. `http://www.oracle.com/technetwork/articles/java/javareflection-1536171.html`. Accessed: 2014-11-30.

[58] Ron Meir and Gunnar Rätsch. Advanced Lectures on Machine Learning. chapter An Introduction to Boosting and Leveraging, pages 118–183. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[59] T.M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[60] Srinivas Mukkamala and Andrew H. Sung. Identifying Key Features for Intrusion Detection Using Neural Networks. In *Proceedings of the 15th International Conference on Computer Communication*, ICCC '02, pages 1132–1138, Washington, DC, USA, 2002. International Council for Computer Communication.

[61] Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar, and Abdullah Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 2014.

[62] J. Oberheide and C. Miller. Dissecting the android bouncer. At `https://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/`, 2012. In SummerCon 2012. Brooklyn, NY.

[63] Sankar K Pal and Sushmita Mitra. Multilayer perceptron, fuzzy sets, and classification. *IEEE Transactions on Neural Networks*, 3(5):683–697, 1992.

[64] M. Parkour. Contagio mobile. `http://contagiominidump.blogspot.com/`. Accessed: 2014-11-30.

[65] Naser Peiravian and Xingquan Zhu. Machine Learning for Android Malware Detection Using Permission and API Calls. In *ICTAI*, pages 300–305, 2013.

[66] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. A survey on security for mobile devices. *IEEE Communications Surveys and Tutorials*, 15(1):446–471, 2013.

[67] David M. W. Powers. Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. Technical Report SIE-07-001, School of Informatics and Engineering, Flinders University, Adelaide, Australia, 2007.

[68] Mykola Protsenko and Tilo Müller. Android Malware Detection based on Software Complexity Metrics. In DEXA Society, editor, *11th International Conference on Trust, Privacy & Security in Digital Business*, 2014.

[69] John Ross Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, March 1986.

[70] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[71] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo García Bringas, and Gonzalo Álvarez Marañón. PUMA: Permission usage to detect malware in Android. In *CISIS/ICEUTE/SOCO Special Sessions*, volume 189 of *Advances in Intelligent Systems and Computing*, pages 289–298. Springer, 2012.

[72] Borja Sanz, Igor Santos, Javier Nieves, Carlos Laorden, Íñigo Alonso-Gonzalez, and Pablo García Bringas. MADS: Malicious Android Applications Detection through String Analysis. In Javier Lopez, Xinyi Huang, and Ravi Sandhu, editors, *Network and System Security*, volume 7873 of *Lecture Notes in Computer Science*, pages 178–191. Springer Berlin Heidelberg, 2013.

[73] Aubrey-Derrick Schmidt, Rainer Bye, Hans-Gunther Schmidt, Jan Clausen, Osman Kiraz, Kamer A. Yüksel, Seyit A. Camtepe, and Sahin Albayrak. Static Analysis of Executables for Collaborative Malware Detection on Android. In *Proceedings of the 2009 IEEE International Conference on Communications*, ICC'09, pages 631–635, Piscataway, NJ, USA, 2009. IEEE Press.

[74] Aubrey-Derrick Schmidt, Frank Peters, Florian Lamour, Christian Scheel, Seyit Ahmet Çamtepe, and Sahin Albayrak. Monitoring smartphones for anomaly detection. *Mob. Netw. Appl.*, 14(1):92–106, February 2009.

[75] Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Jan Clausen, Kamer Ali Yüksel, Osman Kiraz, Ahmet Camtepe, and Sahin Albayrak. Enhancing Security of Linux-based Android Devices. In *Proceedings of 15th International Linux Kongress*. Lehmann, October 2008.

[76] M.G. Schultz, E. Eskin, E. Zadok, and S.J. Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49, 2001.

[77] Patrick Schulz. Android security-common attack vectors. *Rheinische Friedrich-Wilhelms-Universitat Bönn, Germany, Tech. Rep*, 2012.

[78] Giovanni Seni and John F. Elder. *Ensemble Methods in Data Mining: Improving Accuracy Through Combining Predictions*. Synthesis Lectures on Data Mining and Knowledge Discovery. Morgan & Claypool Publishers, 2010.

[79] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google Android: A Comprehensive Security Assessment. *IEEE Security and Privacy*, 8(2):35–44, March 2010.

[80] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. Andromaly: A Behavioral Malware Detection Framework for Android Devices. *Journal of Intelligent Information Systems*, 38(1):161–190, February 2012.

[81] Kaveh Shaerpour, Ali Dehghantanha, and Ramlan Mahmod. Trends in Android Malware Detection. *Journal of Digital Forensics, Security and Law*, 8(3), 2013.

[82] Farrukh Shahzad, M Akbar, S Khan, and Muddassar Farooq. Tstructdroid: Realtime malware detection using in-execution dynamic analysis of kernel process control blocks on Android. *National University of Computer & Emerging Sciences, Islamabad, Pakistan, Tech. Rep*, 2013.

[83] M. Shiraz, 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(3):1294–1313, Third 2013.

[84] Muhammad Shiraz, Saeid Abolfazli, Zohreh Sanaei, and Abdullah Gani. A study on virtual machine deployment for application outsourcing in mobile cloud computing. *The Journal of Supercomputing*, 63(3):946–964, 2013.

[85] Jonathon Shlens. A Tutorial on Principal Component Analysis. *CoRR*, abs/1404.1100, 2014.

[86] Guillermo Suarez-Tangil. *Mining structural and behavioral patterns in Smart Malware*. PhD thesis, Universidad Carlos III, Madrid, October 2014.

[87] Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Jorge Blasco Alís. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. pages 1104–1117, 2014.

[88] Saranya .T, Shalini .A.P, and Kanchana .A. Detection and Prevention for Malicious Attacks for Anonymous Apps. In *International Journal of Innovative Research in Computer and Communication Engineering, Vol 2, Issue 3,*, IJIRCCE, March 2014.

[89] T. Vidas, J. Tan, J. Nahata, C. Tan, N. Christin, and P. Tague. A5: Automated Analysis of Adversarial Android Applications. In *Proceedings of the 4th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM 2014)*, Scottsdale, AZ, November 2014.

[90] Andrew Walenstein, Luke Deshotels, and Arun Lakhotia. Program Structure-Based Feature Selection for Android Malware Analysis. In *Security and Privacy in Mobile Information and Communication Systems*, volume 107 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 51–52. Springer Berlin Heidelberg, 2012.

[91] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis: Android Malware Under The Magnifying Glass. Technical Report TR-ISECLAB-0414-001, Vienna University of Technology, 2014.

[92] Terry Windeatt. Diversity measures for multiple classifier system analysis and design. *Information Fusion*, 6(1):21–36, March 2005.

[93] I.H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2nd edition, 2005.

[94] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69, Aug 2012.

[95] Wen-Chieh Wu and Shih-Hao Hung. DroidDolphin: A Dynamic Android Malware Detection Framework Using Big Data and Machine Learning. In *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems*, RACS '14, pages 247–252, New York, NY, USA, 2014. ACM.

[96] Liang Xie, Xinwen Zhang, Jean-Pierre Seifert, and Sencun Zhu. pbmds: A behavior-based malware detection system for cellphone devices. In *Proceedings of the Third ACM Conference on Wireless Network Security*, WiSec '10, pages 37–48, New York, NY, USA, 2010. ACM.

[97] S.Y. Yerima, S. Sezer, G. McWilliams, and I. Muttik. A New Android Malware Detection Approach Using Bayesian Classification. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 121–128, March 2013.

[98] Harry Zhang. The Optimality of Naive Bayes. In Valerie Barr and Zdravko Markov, editors, *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004)*. AAAI Press, 2004.

[99] Min Zhao, Tao Zhang, Fangbin Ge, and Zhijian Yuan. RobotDroid: A Lightweight Malware Detection Framework On Smartphones. *JNW*, 7(4):715–722, 2012.

[100] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, New York, NY, USA, 2012. ACM.

[101] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

[102] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, February 2012.